Korrekte Transformationsphase — der Kern korrekter Übersetzer

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

der Fakultät für Informatik

der Universität Karlsruhe (Technische Hochschule)

genehmigte

Dissertation

von

Andreas Heberle

aus Tübingen

Tag der mündlichen Prüfung: 4. Februar 2000
Erster Gutachter: Prof. Dr. Gerhard Goos
Zweiter Gutachter: Prof. Dr. Peter H. Schmitt

Inhaltsverzeichnis

Notationen

Schreibweise	Bedeutung
$\mathcal{A}$	Abstrakte Zustandsmaschine
Σ	Signatur
Ω	Signatur der beobachtbaren Funktionen
Q, $Alg(\Sigma)$, $Alg(\Omega)$	Zustandsmenge einer ASM
$\rightarrow$, $\rightarrow_i$, $\rightarrow_\Omega$	Zustandsübergangsrelation
I	Menge von Initialzuständen einer ASM
Init	Gleichungen zur Definition von Initialzuständen
$q, \overline{q}, q', q_i$	Zustand einer ASM
$[\![\mathit{value}]\!]_q$	Interpretation der Funktion *value* im Zustand q
$\Psi : Alg(\Sigma) \rightarrow Alg(\Omega)$	Restriktionsfunktor, der Zustände auf ihren beobachtbaren Anteil abbildet
σ	Signaturmorphismus
ρ, ρ_i	Relation zwischen Zuständen
$\overline{\rho}$, $\overline{\rho}_i$	Relation über dem beobachtbaren Verhalten von Zuständen
Sort	Sorte, Universum
$\mathsf{S}_1 < \mathsf{S}_2$	S_1 ist Untersorte von S_2
$\mathcal{T}(\Sigma)$	Termalgebra über Σ
s, t	Term
type, value	Funktion
S	Sprachkonzept bzw. Montage zu einem Sprachkonzept
$\mathsf{S}(S_1 < \mathcal{A}_1, \ldots, S_k < \mathcal{A}_k)$	generisches Sprachkonzept mit Minimalanforderungen
$\mathcal{C}$, $\mathcal{C}_i$, ϕ	Übersetzung
$\widetilde{f} : U^\infty \rightarrow V^\infty$	Punktweise Anwendung von $f : U \rightarrow V$ auf eine Folge
$\delta : U^\infty \rightarrow U^\infty$	δ löscht benachbarte gleiche Elemente aus einer Folge
$set((t_1, s_1), \ldots)$	Abkürzung für $set((t_1, s_1), set((t_2, s_2), set(\ldots)))$
$\lvert L \rvert$	Kardinalität von L
$abs(i)$	Betrag von i

1 Einleitung

Der Einsatz von Softwaresystemen in sicherheitskritischen Anwendungen stellt an die Korrektheit dieser Systeme erhöhte Anforderungen, denen durch Verifikation der Hardware, der Betriebssysteme und der Quellprogramme Rechnung getragen wird. Trotzdem bleibt eine sicherheitskritische Lücke, da auf einem Rechner die Maschinensprachprogramme ausgeführt werden, die ein Übersetzer aus gegebenen Quellprogrammen erzeugt. Um auf die Korrektheit des Gesamtsystems schließen zu können, muß daher zusätzlich nachgewiesen werden, daß die Übersetzung die Semantik erhält und korrekt implementiert wurde. Solange diese Lücke nicht geschlossen ist, werden Organisationen wie der TÜV und auch das Bundesamt für Sicherheit in der Informationsverarbeitung (BSI) für das Zertifizieren von Quellprogrammen weiterhin die Vorlage und Überprüfung des Maschinencodes verlangen (Bundesamt für Sicherheit in der Informationstechnik, 1994; Zentralstelle für Sicherheit in der Informationstechnik, 1990). Für große Programmsysteme ist dies aber kein praktikabler Weg.

Für die Korrektheit der Übersetzung eines Quellprogramms kommt es alleine darauf an, daß die Ausführung des erzeugten Zielprogramms bei gleichen Eingaben die gleichen beobachtbaren Effekte auf die Umgebung hat wie die Ausführung des Quellprogramms. Wegen der Endlichkeit von Betriebsmitteln wie Speicher oder Beschränkung von Zahlenbereichen erlauben wir allerdings, daß das Zielprogramm mit einem Ressourcenfehler endet, obwohl das Quellprogramm bei gleicher Eingabe fehlerfrei läuft. Ein Übersetzer transformiert Quellprogramme, die gewöhnlich als Text vorliegen, in Zielprogramme, die ebenfalls Texte darstellen. Die Korrektheit der Übersetzung bezieht sich auf die Interpretation dieser Texte. Diese Interpretationen sind durch je eine Spezifikation der Quellsprache und der Zielsprache definiert. Der Übersetzer selbst ist weder dafür verantwortlich, daß diese Spezifikationen korrekt sind, noch daß die Realisierung der Zielsprachspezifikation auf einem Mikroprozessor tatsächlich die im Prozessorhandbuch beschriebenen Zustandsänderungen zur Folge hat. Die Korrektheit des Betriebssystems und der Hardware wird vorausgesetzt.

Die traditionelle Übersetzerarchitektur kennt drei Phasen (Aho et al., 1988), die in Abbildung 1.1 inklusive der auftretenden Repräsentationen dargestellt sind. Die Analysephase prüft Programmtexte auf Legalität, die bedeutungstragenden Elemente werden identifiziert und Kontexteigenschaften überprüft und explizit gemacht. Das Ergebnis dieser Phase ist eine abstrakte Repräsentation, der abstrakte attributierte Strukturbaum. Über dieser abstrakten Repräsentation ist die Semantik einer Quellsprache definiert. Daher gibt es für den Programmtext selbst keine Interpretation, gegenüber der die Korrektheit der Analysephase verifiziert werden kann. Wir können die Korrektheit jedoch durch die Überprüfung des Ergebnisses zusichern (Heberle et al., 1999).

Den eigentlichen Übersetzungsschritt realisiert die Transformationsphase des Übersetzers. Mit ihr beschäftigt sich diese Arbeit. In der Transformationsphase werden den bedeutungstragenden Elementen der Quellsprache entsprechende Elemente der Zielsprache zugeordnet. Es werden die Daten und die Operationen der Quellsprache auf die Zielsprache abgebildet, Optimierungen durchgeführt, und unter Umständen werden Programme reorganisiert. Das Ergebnis sind Zwischensprachprogramme, die in der semantischen Welt der Zielsprache interpretiert

werden. Die Korrektheit der durchgeführten Transformationen muß gegenüber der Semantik der beteiligten Sprachen verifiziert werden.

An die Transformationsphase ist die Codeerzeugungsphase angeschlossen. Dort werden die Maschinenressourcen zugeteilt, Ausführungsreihenfolgen festgelegt und Befehle und Daten codiert. Die Korrektheit der Codeerzeugungsphase kann ebenfalls unter Anwendung von Programmprüfungstechniken sichergestellt werden (Gaul et al., 1999; Zimmermann und Gaul, 1997).

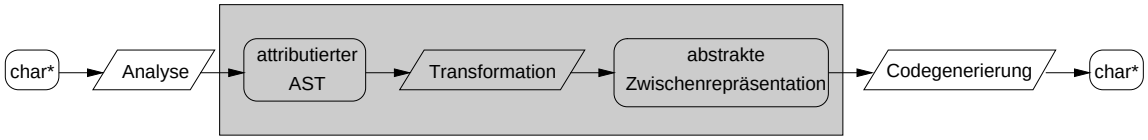


Abbildung 1.1: Phasen und Repräsentationen eines Übersetzers

Wir betrachten in dieser Arbeit imperative Programmiersprachen mit sequentieller und beschränkt indeterministischer Ausführungsreihenfolge, jedoch ohne Parallelität und ohne Echtzeiteigenschaften. Diese Sprachen stellen Basisdatentypen und Operationen zur Komposition von Datentypen zur Verfügung, mit denen strukturierte Datentypen erzeugt werden können. Die Operationen über Daten sind durch die Komposition elementarer Operationen aufgebaut. Weiterhin erlauben die Sprachen Fehlerbehandlung in Form von Ausnahmen. Üblicherweise existieren für die von uns betrachteten Sprachen nur informelle, aber keine formalen Spezifikationen.

1.1 Ziel der Arbeit

Unser Ziel ist die Konstruktion von Übersetzern, die sowohl realistisch als auch korrekt sind. Wir nennen einen Übersetzer *realistisch*, wenn er eine Quellsprache mit Umfang und Eigenschaften praxisrelevanter (realer) Quellsprachen (z. B. Pascal, C oder Java) in eine Zielsprache übersetzt, welche die Eigenschaften von Maschinensprachen typischer Prozessoren (z. B. Transputer oder DEC-Alpha) hat, und wenn der Übersetzer Maschinencode erzeugt, dessen Effizienz mit der von unverifizierten Übersetzern vergleichbar ist. Die Effizienz des Übersetzers selbst ist für diese Arbeit ein zweitrangiges Ziel. Korrekt ist ein Übersetzer, wenn sowohl die Spezifikation der Übersetzung, als auch die Implementierung dieser Spezifikation verifiziert sind. Wir beschränken uns in dieser Arbeit auf die Konstruktion einer korrekten Übersetzungsspezifikation und zeigen die Umsetzung der Spezifikation in eine Implementierung. Die Implementierungsverifikation kann dann mit den bekannten Techniken zur Programmverifikation durchgeführt werden.

Aus unseren Rahmenbedingungen und Zielen ergeben sich bestimmte Konstruktionskriterien, die eine Methode zur Erzeugung einer korrekten Transformationsphase erfüllen muß.

- Die Semantiken der beteiligten Sprachen müssen formal definiert sein, da nur dann formale Beweise geführt werden können. Der Formalismus sollte intuitiv und ohne großen Lernaufwand anwendbar sein, da vom Übersetzerbauer, der letztendlich der Anwender ist, kein Spezialwissen in formalen Methoden vorausgesetzt werden kann (Börger, 1999).

- Die Korrektheitsanforderungen an ein System sollten in den Konstruktionsprozeß integriert und Beweise sollten modular aufgebaut sein. Die Gesamtkorrektheit des Systems sollte aus lokalen Korrektheitseigenschaften und der Korrektheit der Konstruktion folgen (Reif, 1999).
- Die Wiederverwendung verifizierter Transformationen und ihrer verifizierten Implementierung für die Konstruktion neuer Übersetzer ist wichtig, da damit die Gesamtaufgabe erst handhabbar wird. Mit einer geeigneten Aufteilung des Übersetzungsprozesses ist es möglich, eine Quellsprache ohne zusätzlichen Aufwand in unterschiedliche Zwischensprachen zu übersetzen. Damit kann die Anzahl der Übersetzer, die für n Quellsprachen und m Zwischensprachen erzeugt werden müssen, von $n * m$ auf $n + m$ reduziert werden.
- Die traditionelle Übersetzerarchitektur sollte beibehalten werden, da erstens existierende Techniken und Werkzeuge verwendet werden können und zweitens bekannt ist, wie damit effizienter Maschinencode erzeugt werden kann. Der Ansatz sollte vor allem die Benutzung realer Zwischensprachen erlauben (Waite und Goos, 1984; Gaul, 1996).

Existierende Ansätze zur Erzeugung von (korrekten) Übersetzern erfüllen diese Kriterien nur teilweise, und die entstehenden Übersetzer sind entweder verifiziert oder realistisch.

1.2 Die Beschreibung der Lösung

In dieser Arbeit stellen wir einen Rahmen zur Konstruktion korrekter Transformationssysteme vor. Wir geben dem Übersetzerbauer einen Werkzeugkasten an die Hand, mit dem die Semantik von Sprachen spezifiziert, die Spezifikation und Verifikation der Übersetzung, sowie die Implementierung der Übersetzungsspezifikation durchgeführt werden können. Die Korrektheitsbeweise, die bei der Verifikation geführt werden müssen, sind modular, und strukturelle Induktion ist das bestimmende Beweisprinzip. Unser Ansatz erlaubt sowohl die Wiederverwendung der Semantikspezifikation von imperativen Quellsprachkonstrukten, als auch die Wiederverwendung von verifizierten Übersetzungsspezifikationen. Die Umsetzung der Spezifikationen in eine objektorientierte Implementierung ist einfach und die Implementierungsverifikation kann mit existierenden Techniken durchgeführt werden. Implementierungen von Transformationen definieren eine wiederverwendbare, objektorientierte Bibliothek. Wir können Quellsprachen auf beliebige Zwischensprachen abbilden, was eine wichtige Voraussetzung für die Erzeugung effizienten Maschinencodes ist.

1.2.1 Semantikspezifikation und Korrektheit von Übersetzungen

Die Semantik von (imperativen) Programmiersprachen ist kompositionell, d. h., die Bedeutung von Programmen und Abläufen ist aus der Bedeutung einzelner Sprachkonzepte zusammengesetzt. Sprachkonzepte sind durch die Produktionen der abstrakten Syntax vorgegeben. Eine Produktion definiert eine kontextfreie Repräsentation für ein Sprachkonzept. Die linke Seite der Produktion identifiziert das Sprachkonzept, die rechte Seite beschreibt die Teilkonzepte aus denen es aufgebaut ist. Die Kompositionalität der Semantik nutzen wir sowohl für eine modulare Sprachspezifikation, als auch zur Modularisierung der Übersetzungsverifikation aus. Wir definieren die Semantik einer Programmiersprache, indem wir jedem Sprachkonzept eine Konstruktionsvorschrift für einen Interpretierer zuordnen und damit die operationelle Semantik des Sprachkonzepts festlegen. Diese Konstruktionsvorschrift berechnet sich durch die Komposition der Konstruktionsvorschriften für die Teilkonzepte eines Sprachkonzepts. Formal

wird die Semantik durch eine Kombination der Technik von Montages (KUTTER und PIERANTONIO, 1997b) mit attributierten Grammatiken (PAAKKI, 1995) spezifiziert. Die entstehenden Interpretierer sind durch abstrakte Zustandsmaschinen (ASM) beschrieben (GUREVICH, 1995).

Für die Übersetzung eines Sprachkonzepts bzw. einer Sprache müssen wir trotz der modularen Spezifikation immer den kompletten Interpretierer betrachten. Wir haben also hinsichtlich der Übersetzungsverifikation noch keine Modularisierung gewonnen. Das erreichen wir erst, wenn wir in einem zusätzlichen Abstraktionsschritt die Eigenschaften der Komposition von den semantischen Details der Teilkonzepte separieren. Wir abstrahieren von der konkreten Semantik der Teilkonzepte und fordern stattdessen nur minimale Eigenschaften. Das führt uns zu generisch definierten Sprachkonzepten mit Minimalanforderungen für die Semantik ihrer Teilkonzepte. Durch Verwendung der Minimalanforderungen anstatt der konkreten Semantik der Teilkonzepte erreichen wir eine lokale Sicht auf die Semantik eines Sprachkonzepts. Dann können wir auch lokal über Übersetzungen argumentieren. Auf eine komplette Sprache angewendet ergibt sich die Korrektheit einer Übersetzung durch strukturelle Induktion über den Einzelübersetzungen von Sprachkonzepten und der zusätzlichen Beweisverpflichtung, daß die Teilkonzepte jedes einzelnen Sprachkonzepts auch tatsächlich die Minimalanforderungen erfüllen. Unsere Verifikationstechnik ist nicht auf die Ausnutzung der Modularisierung eingeschränkt. Ist es z. B. für Optimierungen notwendig oder günstiger, einen Simulationsbeweis über der kompletten operationellen Semantik eines Sprachkonzepts zu führen, dann ist dies ebenfalls möglich.

1.2.2 Ein Kalkül für korrekte Übersetzungen

Ein übersetzer transformiert Programmtexte nach bestimmten syntaktischen Regeln. Die Korrektheit der Übersetzung wird durch Äquivalenzbeweise gegenüber der Semantik von Programmen bewiesen. Wünschenswert wäre es, wenn es Schlußregeln auf der Semantik gäbe, die diese Äquivalenzbeweise repräsentieren. Allerdings stehen diesem Wunsch die Freiheiten entgegen, die bei der Spezifikation von Semantik mit abstrakten Zustandsmaschinen vorhanden sind.

Ein Programm definiert einen Zustand, der bei der Ausführung des Programms verändert wird. Die formale ASM-Semantik einer Sprache beschreibt sowohl den Zustand als auch die Zustandsübergänge. Unsere Untersuchung existierender Quell- und Zielsprachen bzw. Sprachspezifikationen hat ergeben, daß die Kernelemente des Zustandsraums immer gleich sind und Zustandsübergänge immer nach dem gleichen Ausführungsmodell ablaufen. Diese Tatsache versetzt uns in die Lage, die Spezifikationstechnik für unseren Zweck zu optimieren. Wir haben eine Sprache *AL* definiert, die speziell auf die Spezifikation der Semantik imperativer Sprachen zugeschnitten ist. Die Semantik von *AL* selbst ist operationell durch eine abstrakte Zustandsmaschine beschrieben. Mit der Verwendung von *AL* erreichen wir eine Normalisierung der Semantikbeschreibung, und wir können Transformationsregeln für unterschiedliche Spezifikationen dynamischer Semantik wiederverwenden. Damit haben wir einen Kalkül vorliegen, der bestimmte Annahmen über realistische Sprachen explizit macht. Korrekte Übersetzungen definieren Regeln des Kalküls und repräsentieren Äquivalenzbeweise auf der Semantik.

Wir verwenden die Sprache *AL* zur Spezifikation dynamischer Semantik. *AL* ist universell genug, um die Semantik beliebiger imperativer Sprachkonzepte zu beschreiben. Allerdings kann die Spezifikation aufwendig und schwierig sein. Aus diesem Grund benutzen wir eine Bibliothek vordefinierter Erweiterungen von *AL*. Jede dieser Erweiterungen definiert neue Sprachkonzepte zusammen mit Regeln, die Transformationen in Normalformen (die Terme von *AL*)

beschreiben. Diese Abbildungen definieren die Semantik der Erweiterung. Neben den Regeln für die Erweiterung von AL gibt es in unserem Kalkül Transformationsregeln, die konkreten Transformationen von Übersetzern entsprechen. In vielen Fällen sind die Erweiterungsregeln jedoch mit den Übersetzungsregeln identisch. Das ist nicht verwunderlich, da die Sprache AL die semantischen Konzepte von Zwischensprachen subsumiert.

1.2.3 Wiederverwendung

In unserem Übersetzerrahmen können sowohl Semantikspezifikationen von Sprachkonzepten, als auch verifizierte Übersetzungen zur Konstruktion von Übersetzern für unterschiedliche Quell- und Zielsprachen wiederverwendet werden. Verwenden wir die Sprache AL als konkrete Zwischensprache in einem Übersetzer, dann können wir alle Übersetzer mit dem Ziel AL als Vorderteil für einen Übersetzer in eine neue Zwischen- bzw. Zielsprache verwenden und umgekehrt alle Übersetzungen von AL als Backend für den Übersetzer einer neuen Quellsprache QS. Für die Konstruktion eines korrekten Übersetzers sind die folgenden Arten von Wiederverwendung allerdings wichtiger.

- Ausnutzung vordefinierter Transformationsmuster zur Verifikation neuer Übersetzungen.
- Konstruktion der formalen Semantik einer Sprache mit vordefinierten Sprachkonzepten und gleichzeitiger Wiederverwendung korrekter Transformationsspezifikationen und -implementierungen der beteiligten Sprachkonzepte.

In realen Übersetzern treten bestimmte Übersetzungsmuster unabhängig von den beteiligten Sprachen immer wieder auf, weil sie Übersetzungsaufgaben betreffen, die durch ein bestimmtes Schema gelöst werden. Solche Transformationsmuster haben wir identifiziert und die entsprechenden abstrakten Transformationen spezifiziert und verifiziert. Anstatt nun eine Transformation explizit zu verifizieren, reicht es aus zu zeigen, daß es sich um die Instanz eines Transformationsschemas handelt. Für eine neue Übersetzung ist dieser Nachweis im Normalfall einfacher zu führen als die Verifikation der Transformation selbst.

Mit der graphischen Notation Montages können wir Sprachen modular spezifizieren. Allerdings müssen wir beim Zusammenstecken der Spezifikation aus vordefinierten Sprachelementen verhindern, daß es dabei zu unerwünschten Effekten kommt, weil die Semantik der verwendeten Sprachkonzepte nicht orthogonal ist. Wir haben zwei Möglichkeiten identifiziert, solche Fehler auszuschließen. Zum ersten stellen wir an die Komposition von Sprachkonzepten bestimmte Bedingungen, die zur statischen Semantik der Sprache gehören und die in der semantischen Analyse überprüft werden. Die Komposition von Konzepten, die nicht orthogonal sind, führt dann zu einer widersprüchlichen statischen Semantik. Die zweite Möglichkeit zur Kontrolle der Komposition von Sprachkonzepten erhalten wir durch die Definition von generischen Sprachkonzepten mit Minimalanforderungen. Ein solches Konzept beschreibt eine ganze Klasse von Sprachkonzepten, die durch Instantiierung entstehen. Die Minimalanforderungen beschreiben semantische Eigenschaften (statisch und dynamisch), die von den Teilkonzepten erfüllt werden müssen.

Eine Sprachspezifikation ist genau dann wohlgeformt, wenn die statische Semantik widerspruchsfrei ist und für alle Sprachkonzepte gezeigt werden kann, daß die Teilkonzepte den Minimalanforderungen genügen.

1.3 Erzielte Ergebnisse

Wir haben in dieser Arbeit die Brücke geschlagen zwischen der Theorie, die zum Nachweis der Korrektheit von Übersetzungen angewendet werden muß, und dem ingenieurmäßigen, traditionellen Ansatz zur Konstruktion von Übersetzern. Zusammen mit der Programmprüfung der Analyse- und der Codeerzeugungsphase können wir damit realistische *und* korrekte Übersetzer konstruieren, die imperative Quellsprachen in Zielsprachen existierender Prozessoren übersetzen und Maschinencode erzeugen, der in der Effizienz der Ausführung nur unwesentlich langsamer ist als der erzeugte Code unverifizierter Übersetzer.

Wir haben die folgenden Resultate erzielt:

1. Definition einer einheitlichen Sprache zur operationellen Spezifikation der dynamischen Semantik von imperativen Programmiersprachen.
2. Modularisierung der Übersetzungsverifikation durch die Definition von Sprachkonzepten, die generisch über ihren Teilkonzepten definiert sind. An die Semantik der generischen Parameter werden nur Minimalanforderungen gestellt. Die Korrektheit einer Übersetzung kann lokal nachgewiesen werden. Die Konstruktion garantiert, daß aus der korrekten Übersetzung aller Konzepte einer Sprache die Gesamtkorrektheit der Übersetzung folgt.
3. Entwicklung eines Konzepts zur Wiederverwendung von Semantikspezifikationen und korrekten Transformationen auf der Basis generischer Sprachkonzepte mit Minimalanforderungen für ihre Teilkonzepte. Durch die statische Semantik und die Überprüfung von Minimalanforderungen kann die Konsistenz einer Sprachdefinition sichergestellt werden.
4. Definition und Verifikation von Transformationsmustern, die die Verifikation konkreter Transformationen vereinfachen.
5. Definition einer Bibliothek von Sprachkonzepten und verifizierten Transformationen, die schon zur Konstruktion der mathematisch verifizierten Übersetzung einer Pascal-ähnlichen Quellsprache in eine zur Codeerzeugung geeignete Zwischensprache eingesetzt wurde.

Mit unserem Ansatz können Übersetzer, die imperative Quellsprachen in frei wählbare Zwischensprachen übersetzen, nachweislich korrekt aus einer Bibliothek vordefinierter Komponenten zusammengesteckt werden. Falls die Quellsprache ein Konzept benutzt, das in der Bibliothek nicht vorhanden ist, dann muß dieses neue Konzept durch eine Abbildung in die Bibliothek von Hand definiert werden. Diese Abbildung muß gegenüber der intendierten Semantik validiert oder verifiziert werden, wobei maschinelle Beweisunterstützung durch PVS (Owre et al., 1992) möglich ist. Im Gegensatz zu anderen Ansätzen berücksichtigt die vorliegende Arbeit auch die korrekte Implementierung der Transformationsspezifikation in einer höheren Programmiersprache.

1.4 Aufbau der Arbeit

Im nächsten Kapitel führen wir die traditionellen Übersetzerbautechniken und die Grundlagen zur Konstruktion korrekter Übersetzer ein und stellen die unterschiedlichen Ansätze zur formalen Spezifikation von Semantik vor. Daraus ergeben sich feinere Anforderungen, die ein

Ansatz erfüllen muß, um die gesteckten Ziele zu erreichen. Anhand dieser Anforderungen bewerten wir existierende Arbeiten. In Kapitel 3 stellen wir unseren Ansatz zur Konstruktion korrekter Übersetzer im Überblick vor. Dann führen wir in Kapitel 4 den ASM-Formalismus im Detail ein, beschreiben unsere Notation für abstrakte Zustandsmaschinen und definieren einen Dialekt der graphischen Beschreibungstechnik Montages, den wir für unsere Spezifikationen verwenden. In Kapitel 5 leiten wir die Eigenschaften einer Sprache *AL* her, die speziell zur Spezifikation der dynamischen Semantik imperativer Programmiersprachen geeignet ist und definieren eine konkrete Ausprägung dieser Sprache. Die folgenden Kapitel beschreiben die einzelnen Teilschritte der Konstruktion. Kapitel 6 stellt den Zusammenhang zwischen der Semantikspezifikation und der Korrektheit der Übersetzungsspezifikation her und beschreibt unsere Technik zur Modularisierung der Verifikation. Dann beschreiben wir in Kapitel 7 die Verwendung der Sprache *AL* zur Konstruktion korrekter Übersetzer. Zusätzlich demonstrieren wir an ausgewählten Beispielen die Techniken zur Spezifikation und Verifikation von Sprachkonzepten und Übersetzungsregeln. In Kapitel 8 geben wir allgemeine Übersetzungsmuster an, die in jedem Übersetzer auftreten, und weisen deren Korrektheit nach. Am Beispiel der überladenen Zuweisung zeigen wir den Nutzen für die Verifikation. Eine Bibliothek von Sprachkonzepten und Transformationen wird in Kapitel 9 vorgestellt. Wir beschreiben unterschiedliche Arten der Wiederverwendung von Semantik und Transformationen innerhalb unseres Übersetzungsrahmens und gehen auf mögliche Fehler und ihre Vermeidung bei der Verwendung der Bibliothek ein. Zuletzt beschreiben wir in Kapitel 10 ein objektorientiertes Implementierungsschema für Sprachkonzepte und Transformationen. Dabei behandeln wir auch die Frage der Implementierungskorrektheit. Die Zusammenfassung und der Ausblick schließen diese Arbeit ab. Anhang A beschreibt technische Grundlagen, die zum detaillierten Verständnis der Arbeit benötigt werden, und Anhang B enthält eine ASM-Spezifikation der Semantik von *AL*.

2 Stand der Technik

In diesem Kapitel stellen wir die grundlegenden Techniken im Übersetzerbau vor, soweit sie für diese Arbeit wichtig sind. Wir beschreiben die einzelnen Komponenten eines Übersetzers und analysieren die Eigenschaften der am Übersetzungsprozeß beteiligten Sprachen. Die formale Verifikation eines Übersetzers ist nur möglich, wenn eine formale Spezifikation der Quell- und Zielsprachsemantik existiert. Daher untersuchen wir existierende Formalismen zur Spezifikation von Semantik und bewerten ihrer Eignung für unsere Zwecke. Anschließend legen wir den akzeptierten Stand der Technik zur Konstruktion korrekter Übersetzer dar und geben auf dieser Grundlage konkrete Anforderungen an, die zur praktikablen Konstruktion eines korrekten Übersetzer notwendig sind, der effizienten Maschinencode erzeugen kann[1]. Anhand des aufgestellten Anforderungskatalogs bewerten wir existierende Systeme zur semantikbasierten Übersetzerkonstruktion und identifizieren welche Kriterien von ihnen erfüllt werden und welche nicht. Im nächsten Kapitel stellen wir dann einen Ansatz vor, der alle unsere Anforderungen erfüllt.

2.1 Übersetzerbau-Grundlagen

In diesem Abschnitt vermitteln wir das notwendige Übersetzerbauwissen zum Verständnis der nachfolgenden Kapitel. Wir beschreiben die Aufgaben eines Übersetzers und stellen die traditionelle Übersetzerarchitektur vor. Wir identifizieren die Schnittstellen zwischen den einzelnen Phasen der Übersetzung und erläutern die Techniken, die zur Anwendung gelangen. Anschließend analysieren wir die Eigenschaften der Syntax, der statischen und der dynamischen Semantik von Programmiersprachen und beschreiben die Besonderheiten imperativer Quell- und Zielsprachen. Daraus leiten wir die konkreten Aufgaben der Transformationsphase ab. Mit dieser Hintergrundinformation wird einerseits die Bewertung der Methoden zur formalen Spezifikation von Semantik nachvollziehbar, andererseits benötigt man dieses Wissen zum Verständnis der in den folgenden Kapiteln vorgestellten Zerlegung der Transformations- und Verifikationsaufgabe.

2.1.1 Übersetzerarchitektur

Die traditionelle Übersetzung gliedert sich in drei Phasen, die Analysephase, die Transformations- und die Codeerzeugungsphase. In der Analysephase wird die Struktur des Programmtextes festgestellt, Kontextabhängigkeiten werden entdeckt und Konsistenzbedingungen werden überprüft. Der Eingabetext bekommt in dieser Phase erst Semantik zugeordnet. Die Korrektheit dieser Phase wird nachgewiesen, indem das Ergebnis, der attributierte abstrakte Strukturbaum, auf Konsistenz überprüft wird. In der anschließenden Transformationsphase wird die eigentliche Übersetzungsarbeit erledigt. Hier werden die Konzepte der Quellsprache auf die Konzepte der Zielmaschine abgebildet, es wird z. B. festgelegt, wie Datenobjekte im Speicher abgelegt werden oder wie die Parameterübergabe bei einem Funk-

1) Wenn wir im folgenden von effizienten Übersetzern sprechen, dann meinen wir Übersetzer, die effiziente Maschinenprogramme erzeugen.

tionsaufruf erfolgt. Die Transformation muß gegenüber den Semantiken der Quell- und der Zwischensprache verifiziert werden. An die Transformationsphase kann sich eine Optimierungsphase anschließen. In der Codeerzeugungsphase wird für die laufzeiteffiziente Codierung von Zwischensprachprogrammen durch Programme der Zielmaschine gesorgt. Dieser Schritt muß ebenfalls verifiziert werden.

Die Aufgaben der einzelnen Phasen und die Techniken zu ihrer Lösung sind akzeptiertes Übersetzerbauwissen und werden im folgenden detaillierter beschrieben.

Die **Analysephase** zerfällt in drei Teile. Zuerst werden in der lexikalischen Analyse die Symbole der Quellsprache aus dem Programmtext extrahiert. Anschließend werden in der Zerteilung die Phrasen über den Symbolen identifiziert. Programme, die unbekannte Symbole oder falsche Phrasen enthalten werden hierbei als inkorrekt zurückgewiesen.

Die Spezifikation der Symbole der Sprache erfolgt durch reguläre Ausdrücke. Die Syntax wird durch kontextfreie Grammatiken beschrieben. Die Abtrennung der Lexik von der Syntax ist eine Entwurfsentscheidung des Übersetzerbauers. Eigentlich könnte auch die Lexik in die kontextfreie Grammatik integriert werden. Jedoch können reguläre Sprachen mit endlichen Automaten wesentlich effizienter als kontextfreie Sprachen mit Kellerautomaten akzeptiert werden. Die Ausgabe der syntaktischen Analyse ist ein konkreter Strukturbaum in dem Prioritäten von Operatoren schon aufgelöst sind.

An die syntaktische schließt sich die semantische Analyse an. Dort werden kontextsensitive, statische semantische Eigenschaften eines Programms analysiert und berechnet. Es werden zusätzliche Attribute berechnet, um die der abstrakte Strukturbaum erweitert wird. Zusätzlich wird die Konsistenz dieser Attribute überprüft. Die Aufgaben der semantischen Analyse sind im Einzelnen:

- Die Bereichs- und Namensanalyse: sie setzt die Benutzung von Bezeichnern mit der jeweils gültigen Definitionsstelle in Beziehung.
- Die Typberechnung und Typüberprüfung sowie
- die Operatoridentifikation: sie bestimmen die Typen von Operanden und stellen sicher, daß die Typen für den angegebenen Operator erlaubt sind (hier zählt auch die Zuweisung zu den Operatoren).

Die Spezifikation der statischen Semantik ist eine attributierte Grammatik, aus der die semantische Analyse generiert werden kann. Das Ergebnis dieser Phase ist ein attributierter abstrakter Strukturbaum. Es werden nur Programme weiterübersetzt deren Attributierung (statische Semantik) konsistent ist.

In der **Transformationsphase** wird das Maschinenmodell der Quellsprache auf das Maschinenmodell der Zwischen- bzw. Zielsprache abgebildet. Mit diesen Übersetzungen beschäftigen wir uns in dieser Arbeit, da sie den Kern des Übersetzers darstellen. Wir untersuchen die konkreten Aufgaben dieser Phase noch genauer, nachdem wir die Eigenschaften von Quell- und Zielsprachen untersucht haben.

Eine andere Aufgabe, ohne die kein richtig effizienter Code erzeugt werden kann, ist die Durchführung von Optimierungen. Der Zwischensprachcode wird durch Anwendung algebraischer Identitäten vereinfacht. Nicht erreichbarer Code und redundante, überflüssige Berechnungen werden identifiziert und eliminiert. Um die Qualität der Optimierung sicherzustellen, benötigt man ein Kostenmaß, das von der Zielmaschine abhängig oder unabhängig sein kann.

Informationen, die für die Optimierung nötig sind, werden mit Datenflußanalysetechniken berechnet.

Die Transformationen werden durch (bedingte) Termersetzungsregeln oder Graphersetzungen spezifiziert. Das Ergebnis der Transformationsphase ist eine Graphstruktur über Zielmaschinenoperationen und mit den Datentypen der Zielmaschine.

In der **Codeerzeugungsphase** erfolgt die abschließende Codierung von Zwischensprachprogrammen in ausführbarem Code der Zielmaschine. Es werden folgende Arbeiten durchgeführt.

- Die Ressourcenzuteilung stellt fest, welche Ressourcen während der Ausführung von Instruktionen benötigt werden. Da es sich bei den Ressourcen vor allem um Register handelt, sprechen wir auch von Registerzuteilung.
- Die Speicherabbildung bestimmt die Anfangsadressen und die Größen von Objekten im Speicher und berechnet Adreßausdrücke für den Objektzugriff.
- Die Bestimmung von Ausführungsreihenfolgen legt die Sequenzen fest, in der die Teilberechnungen von Anweisungen ausgeführt werden.
- Die Codeselektion gibt die Kodierung von Operationen in Termen der Zielmaschine an.

In der Codeerzeugungsphase kann die Qualität des erzeugten Codes mit Kostenkriterien der Zielmaschine bewertet werden. Die Codeerzeugung ist damit ein Optimierungsproblem, das mit Heuristiken gelöst wird. Die angewendeten Techniken sind von der Architektur der Zielmaschine abhängig. Zum Beispiel eignet sich für RISC-Architekturen die kostengesteuerte Termersetzung zur Umsetzung von Ausdrucksbäumen in effizienten Maschinencode. Das Ergebnis der Übersetzung ist ein Programm als Folge von Bytes.

2.1.2 Struktur und Semantik von Programmiersprachen

Die Syntax einer Sprache definiert die Struktur von wohlgeformten Programmen und bestimmt welchen Programmtexten überhaupt eine Semantik zugeordnet werden kann. Die Semantik einer Sprache definiert eine Interpretation von Programmen in Form einer Abbildung von Eingaben in Ausgaben. Programme beschreiben also Zustandstransformationen; die Programmsemantik kann als Relation aufgefaßt werden, die Zuständen Zustände zuordnet.

Die in den vorigen Abschnitten beschriebene zusätzliche Unterteilung in Lexik und Syntax bzw. statische und dynamische Semantik hat ingenieurtechnische Gründe, wie z. B. effiziente Verarbeitung. Der Semantik von Programmiersprachen kommt bei der Konstruktion korrekter Übersetzer die größte Bedeutung zu. Die Korrektheit einer Übersetzung kann formal nur gegenüber einer formalen Semantik bewiesen werden. Ihre Struktur beeinflußt sowohl die Übersetzung als auch die Verifikation der Übersetzung.

Die Semantik von Programmiersprachen wird immer kompositionell definiert, weil nur so die Semantik angegeben und einem Menschen verständlich gemacht werden kann. Dabei setzt sich die Bedeutung eines Programms aus den Bedeutungen seiner Teilprogramme zusammen. Die Struktur der Sprachspezifikation ist durch die kontextfreie abstrakte Syntax bestimmt. Programme werden in einer konkreten Syntax notiert, die Eigenschaften berücksichtigt, die für die Kommunikation bzw. das Lesen von Programmen wichtig sind. Die statische Semantik und die dynamische Semantik sind üblicherweise informell angegeben und müssen erst formalisiert werden.

Konkrete und abstrakte Syntax

Die konkrete Syntax einer Sprache wird durch syntaktische Kriterien wie Lesbarkeit, Prioritäten von Operatoren, und die Möglichkeiten der kontextfreien Beschreibung bestimmt. Sie definiert Buchstabierregeln für die Grundsymbole und die Darstellung von Konstanten in Form regulärer Ausdrücke. Außerdem legt sie die Komposition von Sprachkonstrukten, wie z. B. Ausdrücken und Anweisungen fest, die aus Grundsymbolen und Konstanten aufgebaut sind. Die konkrete Syntax ist durch eine kontextfreie Grammatik beschrieben, die, wenn man daraus automatisch einen effizienten Zerteiler erzeugen will, auch noch die LALR(1)- oder LL(1)-Eigenschaft haben sollte. Reguläre Sprachen könnten zwar noch effizienter akzeptiert werden, aber Programmiersprachen gehören normalerweise nicht in diese Chomsky-Sprachklasse.

Die abstrakte Syntax beschreibt die bedeutungstragenden Einheiten der Sprache. Das Ziel der Analysephase ist es nur, die Bedeutung eines Programms herauszufinden. Aus diesem Grund werden syntaktische Besonderheiten, wie z. B. Operator-Präzedenzen oder bestimmte Schlüsselworte in der Darstellung der Semantik nicht benötigt. Mit der abstrakten Syntax abstrahieren wir von Details der äußeren Form, die für die Semantik und damit auch für die Übersetzung unwesentlich sind. Der AST ist die verdichtete Darstellung des konkreten Strukturbaums.

Es gibt zwei unterschiedliche Sichtweisen auf die Beziehung von konkreter und abstrakter Syntax. Der Übersetzerbauer gewinnt die abstrakte Syntax aus der konkreten, indem er syntaktische Komplikationen, wie Kettenproduktionen oder überflüssige Symbole eliminiert, die aus Gründen der Lesbarkeit oder Verständlichkeit eingeführt wurden. Für den Entwerfer der Sprache ist es genau umgekehrt. Er definiert die Semantik seiner Sprache über der abstrakten Syntax und gibt anschließend eine konkrete und leicht verständliche Syntax dazu an. In dieser konkreten Syntax sind aus Gründen der Bequemlichkeit und der Pragmatik auch Mehrdeutigkeiten erlaubt, die jedoch die Zerteilung erschweren. Beispiele hierfür sind unterschiedliche Darstellungen für Konstanten oder überflüssige Klammern, die zur Berechnung von Ausdrücken nicht benötigt werden.

Über der abstrakten Syntax wird die statische und die dynamische Semantik einer Sprache definiert.

Statische und dynamische Semantik

Attributierte Grammatiken beschreiben kontextfreie und kontextabhängige Eigenschaften einer Sprache und definieren die statische Semantik. Die Attribute assoziieren Bezeichner mit den entsprechenden Objekten oder geben die Typen von Operationen an. Zu der statischen Semantik gehören auch Steuer- und Datenflußinformationen. Die statische Semantik beschreibt zusätzlich Kontextbedingungen, die nicht in der kontextfreien Syntax beschrieben werden können.

Die Grenze zwischen statischer und dynamischer Semantik ist nicht fest definiert und hängt sowohl vom Sprachdesigner als auch vom Übersetzer ab. Zum Beispiel könnte ein Übersetzer durch Analysen Eigenschaften über das dynamische Verhalten eines konkreten Programms berechnen und hätte damit zur Übersetzungszeit mehr Informationen. Genau das passiert in optimierenden Übersetzern. Üblicherweise gehören zur statischen Semantik nur Informationen, die eine abstrakte Sicht auf Programme unabhängig von der Programmausführung beschreiben. Dazu zählt Typinformation, Steuer- und Datenflußabhängigkeiten.

Für diese Arbeit ist die Maxime, daß alle Informationen, die für die Transformation benötigt werden, zur statischen Semantik gehören und während der semantischen Analyse berechnet werden. Das bedeutet vor allem, daß wir Informationen schon dort berechnen, obwohl wir sie erst später benötigen. Die konkrete Implementierung der statischen Semantik und die effiziente Berechnung der gewünschten Information ist für uns nicht relevant. Wir setzen die Existenz einer Datenbank voraus, aus der die statischen Informationen erfragt werden können.

Die Semantik kann und muß auch aus Sicht des Sprachdesigners betrachtet werden. Er definiert mit der dynamischen Semantik die dynamischen Eigenschaften eines Programms. Für diese Definition sind bestimmte Information nötig, die berechnet sein müssen. Sind diese Informationen statisch berechenbar, dann gehören sie zur statischen Semantik. Aus diesem Blickwinkel bestimmt die Definition der dynamischen Semantik die Eigenschaften der Analysephase und schränkt die Möglichkeiten der Transformationsphase ein.

Die Semantik einer Programmiersprache wird induktiv über der Baumstruktur der abstrakten Syntax angegeben. Ist $G = (N, T, P, Z)$ die abstrakte Syntax mit Nichtterminalen N, Terminalen T, Produktionen P und Startsymbol Z, dann ordnet die Sprachdefinition jedem Nichtterminal $n \in N$ eine Abbildung $[\![n]\!]$ zu, die eine Interpretation im semantischen Modell festlegt. Die Gesamtsemantik eines Programms π ist durch $[\![Z]\!]$ definiert. Die dynamische Semantik baut sich ausgehend von den Semantiken der Einzelteile (Blätter) induktiv über dem Programmterm (Baum) auf. Allerdings kann diese induktive Struktur der Semantik nicht immer durchgehalten werden, weil man Kontextinformation benötigt. Für imperative Quellsprachen ist diese Kontextinformation normalerweise statisch und wird in der semantischen Analyse berechnet.

Die Strukturen der Ziel- bzw. Zwischensprache sind wesentlich einfacher. Zielprogramme sind durch Listen von Instruktionen beschrieben. Instruktionen können sequentiell oder über Sprünge verknüpft sein. Die Struktur von Zielprogrammen hängt natürlich damit zusammen, daß sie im sequentiellen Speicher der Maschine abgelegt werden. Die Einfachheit der Struktur hat den Nachteil, daß die Kontextinformation, die in Form von Sprungzielen benötigt wird, nicht mehr so leicht anhand der Struktur abgeleitet werden kann. Gehen wir davon aus, daß Sprünge in beliebige Programmteile erfolgen können, dann definieren Zielmaschinenprogramme spezielle Steuerflußgraphen. Anweisungen stellen Ecken dar, der Steuerfluß ist durch Kanten beschrieben. Die Sprünge innerhalb eines Programms können durch Kanten zwischen Instruktionen repräsentiert werden.

2.1.3 Transformationsphase eines Übersetzers

Die Transformationsphase hat die Aufgabe, die Konzepte der Quellsprache auf die weniger komplexen Konzepte der Maschinensprache abzubilden. Die Unterschiede zwischen den Konzepten der Quelle und des Ziels bestimmen die Aufgaben der Übersetzung. Die Transformation legt die Anforderungen an die semantische Analyse fest und bestimmt die abstrakte Syntax. Das Ergebnis dieser Phase ist die Eingabe der Codeerzeugungsphase.

Eigenschaften von Quell- und Zielsprachen

Wir können die Konzepte von imperativen Programmiersprachen in mehrere Klassen unterteilen:

Ausdrücke stellen die Grundlage von Berechnungen dar. Dazu gehören Werte und Operationen auf diesen Werten.

Typen abstrahieren bestimmte Eigenschaften von Objekten und definieren die anwendbaren Operationen auf Objekte eines Typs.

Speicher repräsentiert den Zustand eines Programms[2].

Ablaufsteuerung definiert die Ausführungsreihenfolgen von Berechnungen.

Die Basiswerte der Berechnung sind in Quell- und Zielsprachen meistens gleich. Allerdings unterscheidet sich die Codierung dieser Werte unter Umständen stark. Auf Quellsprachebene findet man z. B. natürliche Zahlen, während auf der Zielmaschine Bytefolgen verarbeitet werden.

Das Typsystem von Quellsprachen wird benutzt, um die Korrektheit von Operationsanwendungen zu überprüfen und um Programmierfehler statisch zu erkennen. Zwischensprach- und Zielsprachprogramme werden üblicherweise automatisch erzeugt, daher ist deren Typsystem wesentlich einfacher.

Die Unterschiede im Speichermodell hängen mit dem Abstraktionsniveau der Sprachen zusammen. Es ist einsichtig, daß sich der Programmierer keine Gedanken über die konkrete Repräsentation seiner Objekte und Werte auf der Zielmaschine machen will. Auf Quellsprachebene ist der Speicher eine Menge von Objekten, es können neue Objekte erzeugt oder existierende freigegeben werden, ihr Inhalt kann gelesen und geändert werden. Der Speicher ist strukturiert. Auf der Zielmaschine ist der Speicher eine Liste von Speicherstellen, die über Adressen zugegriffen werden.

Die Steuerflußkonzepte der Quellsprache sind so angelegt, daß ein Konzept ein bestimmtes Schema über Berechnungen definiert. Sieht man von Quellsprachen ab, die Sprünge an beliebige Programmpunkte erlauben, dann ist Steuerfluß lokal definiert. Auf der Zielmaschine gibt es nur die sequentielle Ausführung und Sprünge.

Quellsprachen definieren zusätzlich Konzepte zur Abstraktion, wie z. B. Modul oder Klasse. Auf der Zielmaschine gibt es diese Konzepte nicht. Die Unterschiede von Quell- und Zielsprachen sind in Tabelle 2.1 noch einmal zusammengefaßt.

Bemerkung: Viele Konzepte zur Abstraktion sind in Wirklichkeit keine zusätzlichen Konzepte, sondern schlagen sich nur als komplexere Typinformation in der statischen Semantik der Quellsprache nieder. In der Transformationsphase werden diese Konzepte mit Hilfe der Typinformation wie Standardkonzepte von Quellsprachen abgebildet. ♢

Aufgaben der Transformationsphase

Entsprechend der Analyse der Unterschiede von Quell- und Zwischensprachen und unter der Voraussetzung, daß die semantische Analyse die statische Semantik von Programmen berechnet, ergeben sich für die Transformationsphase eines Übersetzers konkret die folgenden Aufgaben:

1. Abbildung von Wertecodierungen zusammen mit den entsprechenden Operationen,
2. Speicherabbildung, wobei strukturierte Objekte in ihre Basisobjekte zerlegt werden und die entsprechenden Zugriffspfade in Relativadressen und Adreßrechnung umgewandelt und Variablendeklarationen durch Speicherallokationen ersetzt werden,

2) Der Speicher umfaßt auch Teile des Laufzeitsystems, wie den Befehlszähler oder den Kellerzeiger.

Konzept	Quellsprache	Zielsprache
Ausdrücke		
Werte	Konstanten entsprechend den Basistypen	Konstanten für Operationscodes, Adressen, Ganz- und Gleitkommazahlen
Operationen	Operation der Basistypen (mathematisch oder konkret), Funktionsaufruf	arithmetische Operation der Zielmaschine, Funktionsaufruf oder Sprung
Typen		
Basistypen	Integer, Float, Char, String	Integer, Float, Addr, Byte
Typkonstruktoren	Funktionstyp, strukturierter, generischer, polymorpher Typ	Liste
Speicher		
Objekte	Elementarobjekte, strukturierte Objekte	Speicherstellen
Erzeugung von Objekten	Variablendeklaration, explizite Speicherallokation	Speicherallokation
Zugriff auf Objekte (lesend, schreibend)	Zugriffspfad (Referenz-, Wertesemantik)	Adressierung
Ablaufsteuerung		
Sequenz	„;“	pc + 1
Iteration	While, Loop, Repeat	
Verzweigung	If, Case	un-/bedingter Sprung
Sonstige	Parallele Ausführung, Indeterministische Ausführung, Ausnahmen (benutzer- und operationendefiniert)	Ausnahmen operationendefiniert
Höhere Konzepte		
Abstraktion	Prozedur, Funktion, Modul, Klasse, Übersetzungseinheit	Funktion

Tabelle 2.1: Konzepte von Quell- und Zielsprachen

3. Abbildung der komplexen Befehle zur Ablaufsteuerung in entsprechend codierte Sprunganordnungen, Elimination von Indeterminismus und
4. Auflösung höherer Konzepte, die zu Abstraktionszwecken in der Quellsprache existieren oder Nachimplementierung dieser Konzepte mit den Konzepten der Zielsprache.

Die Quellsprache wird in der Transformationsphase in eine Repräsentation (Zwischensprache) abgebildet, die strukturell den Konzepten der Zielsprache entspricht, vgl. z. B. (Waite und Goos, 1984). Die Zwischensprache bildet die Schnittstelle zwischen einer Klasse von Hochsprachen und einer bestimmten Klasse von Zielmaschinen. Sie ist im Prinzip quell- und zielsprachunabhängig, kann aber mit Zielmaschineneigenschaften parametrisiert sein. Die Wahl der Zwischensprache ist essentiell für die Erzeugung effizienten Maschinencodes. Dafür gibt es mehrere Gründe:

- Paßt die Zwischensprache nicht zum Codeerzeugungsmechanismus, dann können im besten Fall die speziellen Vorteile der Codeerzeugungstechnik nicht ausgenutzt werden, im schlechtesten Fall muß ein zusätzlicher Übersetzungsschritt in eine andere Zwischensprache durchgeführt werden, bevor Code erzeugt werden kann. Zum Beispiel kann aus einer Zwischensprache, die eine Kellermaschine implementiert, kein effizienter Zielmaschinencode generiert werden, wenn der Erzeugungsmechanismus explizite und benannte Zwischenergebnisse zur optimierten Implementierung von Ausdrücken benötigt. Diese Teilausdrücke tauchen in der Kellermaschine nur implizit und durch die Kellerstruktur codiert auf.
- Für Transformationen/Optimierungen benötigt man bestimmte Hilfsinformationen. Können diese Informationen (z. B. Freiheitsgrade der Semantik), die im ursprünglichen Programm vorhanden waren, in der Zwischensprachen nicht repräsentiert werden, dann kann die entsprechende Optimierung anschließend nicht mehr durchgeführt werden. Ein Beispiel für diese Art von Information ist die beliebige Auswertungsreihenfolge von Argumenten eines Funktionsaufrufs.
- Bestimmte Optimierungen können nur auf einer bestimmten Repräsentation effizient durchgeführt werden (Trapp, 1999).
- Liegt die Zwischensprache zu nahe an Quellsprachen oder paßt sie nicht zur Zielsprache, weil die Operationen nicht die der Zielmaschine sind, dann verlagert sich die eigentliche Übersetzungsarbeit von der Transformationsphase in die Codeerzeugungsphase. Darauf sind die existierenden Techniken zur Codeerzeugung nicht ausgelegt.

In der Transformationsphase werden Übersetzungen bestimmter Sprachkonzepte zusammen durchgeführt. Das entspricht der Einführung neuer logischer, nicht notwendigerweise konkret repräsentierter, Zwischensprachen. Die Transformation wird durch Gruppen von Ersetzungsregeln beschrieben, die in handelsüblichen Übersetzern durch die Verwendung von Termersetzungstechniken und durch gesteuerte Graphersetzung effizient implementiert werden.

2.2 Formale Definition von Semantik

Formale Beschreibungsmethoden für Semantik teilen wir in zwei Klassen ein. Erstens Ansätze, die die Bedeutung eines Programms im Ganzen beschreiben. Hier betrachten wir stellvertretend die *denotationelle Semantik* (Strachey, 1966; Schmidt, 1986). Sie modelliert die Bedeutung eines Programms als Funktion oder Relation zwischen Ein- und Ausgaben. Einzig

der Effekt eines Programms ist von Interesse, nicht wie dieser Effekt zustande kommt. In diese Klasse fallen auch die Ansätze, bei denen die Bedeutung eines Programms durch Angabe von prädikatenlogischen Formeln oder Axiomen beschrieben ist. Die zweite Klasse sind operationelle Ansätze, die spezifizieren, wie ein Programm ausgeführt wird bzw. wie das Resultat berechnet wird. Eine Berechnungsfolge beschreibt dabei eine Folge von Zustandsänderungen. Prinzipiell definiert eine operationelle Semantik eine abstrakte Maschine, die Programme einer Sprache interpretieren kann.

Das wichtigste Kriterium für die Bewertung der Ansätze ist, daß wir eine einheitliche Beschreibungsmethode sowohl für Quell- als auch für Zwischen- und Zielsprachen haben wollen, da ein Wechsel des Formalismus zusätzliche Fehlermöglichkeiten an der Schnittstelle von einem Formalismus zum nächsten einführen würde. Außerdem müssen wir berücksichtigen, daß wir imperative Sprachen beschreiben wollen, die indeterministische und nichtterminierende Ausführungsfolgen und Fehlerbehandlung definieren können.

2.2.1 Denotationelle Semantik

In denotationeller Semantik wird die Bedeutung eines Programms ausgehend von mathematischen Objekten (semantischen Domains), wie z. B. Funktionen oder Zahlen konstruiert. Um Semantik zu beschreiben, werden Funktionen definiert, die über den semantischen Domains operieren. Rekursion und Iteration wird durch Fixpunktberechnungen ausgedrückt. Die Semantik ist kompositionell in dem Sinne, daß die Bedeutung einer syntaktischen Einheit aus der Bedeutung der syntaktischen Untereinheiten aufgebaut ist.

Funktionale Sprachen können mit denotationeller Semantik gut spezifiziert werden. Beliebige Änderungen im Steuerfluß können jedoch nur mit sog. Continuations (Strachey und Wadsworth, 1974) beschrieben werden, die auch in alle verwendeten semantischen Domains integriert werden müssen. Das zeigt einen Nachteil denotationeller Semantik, die Spezifikationen sind nicht modular. Man kann Spezifikationen nicht unabhängig spezifizieren und zusammenfügen, weil die Integration von Continuations auch die semantischen Domains anderer Teile einer Spezifikation verändert. Für realistische Programmiersprachen, die eine große Anzahl von Konstrukten enthalten, müssen viele semantische Domains angegeben werden, deren Komplexität mit der Gesamtzahl der Domains zunimmt. Das führt dazu, daß Spezifikationen, besonders für Nichtspezialisten, schwer verständlich sind und schlecht wiederverwendet werden können. Die Schwierigkeit der Spezifikation zeigt sich auch darin, daß es kaum veröffentlichte Spezifikationen von realistischen Programmiersprachen gibt. Für die existierenden Spezifikationen von Pascal (Tennent, 1978) oder Ada (Donzeau-Gouge et al., 1980) wurden stark vereinfachende Annahmen gemacht. Schwierig zu spezifizieren sind vor allem Sprachkonzepte, die eine Änderung des Steuerflusses definieren, die nicht direkt entlang der Struktur beschrieben werden kann. Dazu gehören Sprünge, *break* und *continue* (C, C++), *exit A* (Ada), aber auch die Behandlung von Ausnahmen. Hingegen können indeterministische Ausführungen relativ einfach durch mathematische Funktionen codiert werden, die über die Reihenfolge von Berechnungen keine Aussagen machen.

Da denotationelle Semantiken die Bedeutung eines Programms über die Bedeutung der Untereinheiten definieren, ist die bevorzugte Beweistechnik strukturelle Induktion. Zum Nachweis, daß eine Eigenschaft für alle Programme einer Sprache gilt, wird über die Teilkonstruktionen argumentiert. In Zusammenhang mit der Beschreibung von Schleifen und Rekursion wird Fixpunktinduktion benutzt.

Aktionssemantik (Mosses, 1992) stellt eine leichter zu lesende und modulare Variante denotationeller Semantik dar. Die Spezifikation enthält operationelle Anteile, die durch vordefinierte Operatoren angegeben sind und mit denen Variablendeklarationen, Steuerfluß und auch Seiteneffekte ausgedrückt werden können. Diese Standardfunktionen sind umgangssprachlich benannt (*execute*, *find*, *update* etc.) und daher für den Praktiker leicht verständlich. Allerdings können Operationen unterschiedliche Effekte haben und müssen daher für die Verwendung gut verstanden sein. Aktionssemantiken haben kein Konzept zur Definition neuer Standardberechnungen bzw. zur Erweiterung des Formalismus. Sie eignen sich nicht zur Spezifikation von Maschinensprachen, weil die Standardaktionen die konkreten Maschinenkonzepte nicht abdecken.

2.2.2 Natürliche Semantik

Natürliche Semantik (Kahn, 1987) stellt eine Mischung aus denotationeller und operationeller Semantik dar. Die Spezifikation beschreibt den Kontext, in dem eine Berechnung auftreten kann. Wie in denotationeller Semantik wird das Ergebnis einer Ausführung betrachtet, nicht der Weg, wie das Ergebnis berechnet wurde. Damit definieren natürliche Semantiken eine „big step"-Semantik, die nur die initialen Zustände mit den finalen Zuständen einer Ausführung in Beziehung setzt und dabei keine Zwischenzustände betrachtet.

Eine natürliche Semantik wird durch eine Menge von Inferenzregeln angegeben, eine vollständige Berechnung ist durch den entstehenden Beweisbaum definiert. Inferenzregeln haben die Form[3]:

$$\frac{\langle S_1, s_1\rangle \to s_1', \ldots, \langle S_n, s_n\rangle \to s_n'}{\langle S, s\rangle \to s'} \; if \ldots$$

Über der Linie stehen die Voraussetzungen unter denen die Konklusion, die unterhalb der Linie steht, gezogen werden kann. Die Bedingung spezifiziert, wann eine Regel gültig ist. Axiome sind Regeln ohne Prämissen.

Eigenschaften werden in natürlicher Semantik durch Beweise über den Aufbau von Ableitungsbäumen nachgewiesen. Das setzt jedoch voraus, daß ein Ableitungsbaum existiert. Für Aussagen über die statische Semantik ist das auf jeden Fall gegeben. Natürliche Semantik wird auch zur Definition der dynamischen Semantik einer Programmiersprache eingesetzt (Pettersson, 1995). Allerdings ergibt sich dann ein Problem mit nichtterminierenden Programmen. Nichtterminierende Berechnungsfolgen haben nämlich keine Bedeutung, weil in diesem Fall kein Ableitungsbaum berechnet werden. Eine Art von Fixpunktoperator zur Definition beliebig langer Ableitungsbäume ist nicht vorhanden. Daher wird z. B. die Semantik einer Schleife über schrittweises Ausrollen der Schleife definiert. Mit dieser selbstbezüglichen Definition geht für nichtterminierende Schleifen jedoch die Kompositionalitätseigenschaft verloren.

Beispiel 2.1 Die Semantik einer While-Schleife würde man in natürlicher Semantik folgendermaßen definieren:

$$(i) \quad \frac{\langle S, s\rangle \to s', \langle while\ b\ do\ S, s'\rangle \to s''}{\langle while\ b\ do\ S, s\rangle \to s''} \quad if\ eval(b) = true$$

$$(ii) \quad \langle while\ b\ do\ S, s\rangle \to s \quad if\ eval(b) = false$$

3) Wir verwenden hier die Syntax aus (Nielson und Nielson, 1992).

Diese Spezifikation definiert für die Schleife keine Semantik, wenn für ihre Prämisse, also im Falle von $eval(b) = true$ für die Schleife selbst, keine Ableitung gefunden werden kann. ♢

Ist die nichtterminierende Schleife Teil einer Anweisungsliste, dann hat auch die Anweisungsliste keine Semantik. Da die Semantik eines Programms üblicherweise über Anweisungslisten definiert wird, hat ein Programm mit einer nichtterminierenden Schleife überhaupt keine Semantik. Das Terminierungsproblem ist kritisch, weil wir auch die Übersetzung nichtterminierender Programme verifizieren wollen. Selbstbezügliche Definitionen haben den zusätzlichen Nachteil, daß strukturelle Induktion als Beweisprinzip ausgeschlossen ist.

Strukturiert operationelle Semantik (SOS) wurde von PLOTKIN (1981) eingeführt. SOS beschreibt Semantik, indem die einzelnen Schritte einer Berechnung spezifiziert werden. Im Prinzip beschreibt eine SOS-Spezifikation ein Termersetzungssystem und eine Menge von Inferenzregeln, die genau den Kontext definieren, in dem ein Berechnungsschritt ausgeführt werden kann. Es werden zwei Arten von Regeln unterschieden:

1. Die Berechnung terminiert, $\langle S, s\rangle \Rightarrow s'$, d. h., die Ausführung von S endet in einem Zustand s' oder
2. sie terminiert nicht, $\langle S, s\rangle \Rightarrow \langle S', s'\rangle$. Dann beschreibt $\langle S', s'\rangle$ die noch verbleibende Ausführung[4].

Die Berechnung der Semantik eines Programms entspricht der Anwendung von Termersetzungsregeln. Jeder Interpretationsschritt definiert einen neuen Term. SOS-Semantiken sind kompositionell, Probleme mit selbstbezüglichen Regeln, wie bei natürlicher Semantik, können nicht auftreten. Der Mechanismus beschreibt eine „small step“ Semantik, weil jede Berechnung einen kleinen Schritt zum Ziel darstellt. Die Beschreibung nichtterminierender Programme stellt bei diesem Berechungsmodell kein Problem dar, weil jeder einzelne Berechnungsschritt beschrieben wird. Die anwendbare Beweistechnik ist Induktion über die Anzahl der Berechnungsschritte. Allerdings kann man auch strukturell argumentieren, da die Definition der Semantik sehr stark die Struktur der Programmiersprache nutzt. Voraussetzung ist ein Vorliegen von Programmen als Term.

2.2.3 Abstrakte Zustandsmaschinen

Abstrakte Zustandsmaschinen (**A**bstract **S**tate **M**achine, ASM) wurden von GUREVICH (1995, 1997) vorgeschlagen und beschreiben Semantiken operationell durch Definition eines abstrakten Interpreters. Die zugrundeliegenden Konzepte sind aus der Mathematik übernommen:

- Zustände sind statische Algebren über einer Signatur Σ,
- Zustandsübergangsrelationen werden durch Regeln definiert, welche die Interpretation der Funktionen aus Σ ändern und
- eine Menge ausgezeichneter Zustände beschreibt die Initialzustände der Ausführung.

ASMs definieren ebenfalls eine „small step“-Semantik. Die Semantik eines Programms wird durch eine Folge von Zuständen beschrieben, die bei der abstrakten Ausführung des Programms auftreten. Machbarkeitsstudien zeigen, daß ASMs zur Spezifikation der kompletten Semantik imperativer Programmiersprachen wie C und C++ (GUREVICH und HUGGINS, 1993; WALLACE, 1994), OBERON (KUTTER und PIERANTONIO, 1997a) oder JAVA (BÖRGER

4) Diese Beschreibung einer Ausführung nennt man auch *Continuation Passing Style* (*CPS*)

und SCHULTE, 1998c) geeignet sind. Dabei wurden die kompletten Sprachen ohne vereinfachende Annahmen beschrieben, was nicht selbstverständlich ist, wenn man z. B. die Arbeiten mit denotationeller Semantik betrachtet.

Die existierenden Sprachspezifikationen definieren Interpreter mit einem abstrakten Befehlszähler *ct*, der bei der Ausführung eines Programms entsprechend dem Steuerfluß weitergeschaltet wird. Bei der Ausführung eines Kommandos wird die Interpretation bestimmter Funktionen der ASM verändert. Diese Zustandsänderung ist durch eine Regel beschrieben, die damit die operationelle Semantik eines Kommando festlegt. Zum Beispiel würde man eine Schleife durch folgende Regel für das While-Kommando definieren, das darüber entscheidet, ob die Schleife beendet ist oder ob der Schleifenrumpf noch einmal ausgeführt wird.

```
if ct = While then
  if value(ct, condition) = true then
     ct := truetask(ct)
  else
     ct := falsetask(ct)
  endif
endif
```

Diese Regel besagt, daß in Abhängigkeit der aktuellen Interpretation von *value(ct, condition)* der abstrakte Befehlszähler verändert wird. Die Steuerflußinformation, die definiert, daß *truetask(ct)* dem ersten Kommando der Schleife entspricht und die Interpretation von *falsetask(ct)* das nachfolgende Kommando der Schleife ist, gehört zur statischen Semantik. Mit dieser Definition gibt es keine Probleme mit nichtterminierenden Berechnungen, da bei der Interpretation nur jeweils ganz lokal entschieden wird, welche Zustandsänderungen durchgeführt werden.

Eigenschaften über ASMs werden durch Induktionsbeweise über der Anzahl der Zustandsübergänge bewiesen. Die Korrektheit von Übersetzungen wird durch Simulationsbeweise, wie in der Berechenbarkeits- und Komplexitätstheorie, gezeigt. Übersetzungen können auch durch Verfeinerung korrekt erzeugt werden. Allerdings definiert der ASM-Formalismus keinen festen Verfeinerungsbegriff.

In Zusammenhang mit der Spezifikation von Programmiersprachen wurden *Montages* (KUTTER und PIERANTONIO, 1997b) als eine graphische Notation für spezielle abstrakte Zustandsmaschinen entwickelt. Mit Montages können komplette Programmiersprachen mit konkreter Syntax, statischer und dynamischer Semantik spezifiziert werden. Eine Montages-Spezifikation definiert für einen konkreten Strukturbaum zwei abstrakte Zustandsmaschinen. Die erste Maschine beschreibt im Prinzip die semantische Analyse. Sie weist Programme zurück, die nicht wohlgeformt im Sinne der statischen Semantik sind, und berechnet den Initialzustand der zweiten abstrakten Zustandsmaschine. Diese definiert die dynamische Semantik der Sprache durch einen Interpretierer für konkrete Strukturbäume. Montages ermöglichen die strukturelle Definition der operationellen Semantik von Programmiersprachen. Diese Spezifikation ist modular anhand der kontextfreien Syntax der Sprache aufgebaut. Ein Problem von Montages liegt in der Definition der statischen Semantik. Die abstrakte Zustandsmaschine, welche die statische Semantik festlegt, ist mit einer Links-Rechts-Tiefensuche über dem konkreten Strukturbaum definiert. Allerdings ist die statische Semantik imperativer Sprachen normalerweise nicht in einem einzigen Durchlauf berechenbar, weil z. B. die Benutzung von Variablen vor ihrer Definition erlaubt ist. Dann kann es passieren, daß die Namensanalyse im Moment der Verwendung keine Informationen über eine Variable hat. Montages lösen dieses

Problem durch zusätzliche Anforderungen an Programme oder verlagern die Auflösung in die dynamische Semantik.

2.2.4 Bewertung der Formalismen zur Spezifikation von Semantik

Denotationelle Semantiken sind gut geeignet, um Aussagen über Eigenschaften von imperativen Quellprogrammen nachzuweisen. Die strukturelle Definition der Semantik ist geeignet für Quellsprachen, aber Kontextinformation, die sich nicht einfach über der Struktur definieren läßt ist problematisch. Daher sind denotationelle Semantiken nicht zur Definition von Zwischen- und Zielsprachen geeignet. Außerdem sind die Spezifikationen nicht modular und nur schwer wiederverwendbar. Für unseren Zweck ist denotationelle Semantik daher nicht die richtige Technik zur Spezifikation von Programmiersprachen.

Natürliche Semantik eignet sich sehr gut zur Spezifikation von Typsystemen und statischer Semantik. Es können semantische Analysierer und Typüberprüfer generiert werden (Pettersson, 1995). Eine Erweiterung zur Spezifikation der statischen Semantik objektorientierter Programmiersprachen stellen mehrsortige natürliche Semantiken dar (Glesner, 1999). Für die Spezifikation von Zwischensprachen ist eine „big step“ Semantik aus den gleichen Gründen wie denotationelle Semantiken nicht geeignet. Die verzahnte Ausführung paralleler Berechnungen[5] läßt sich aufgrund der „big step“ Eigenschaft nur schwer beschreiben, da die Semantik eines Konstrukts aus atomaren Teilen abgeleitet wird. Das ist problematisch, wenn z. B. die Semantik der Ausdrucksberechnung besagt, daß Teilausdrücke nicht nur komplett und sequentiell, sondern auch verzahnt berechnet werden dürfen. Ebenso kann eine indeterministische Ausführungsfolge nur durch Aufzählung aller Möglichkeiten modelliert werden Die Bedeutung nichtterminierender Programme kann mit der beschriebenen Variante nicht spezifiziert werden.

Mit strukturiert operationeller Semantik läßt sich die dynamische Semantik von Quellsprachen gut beschreiben. Bei der Beschreibung von Zwischensprachen macht die starke Abhängigkeit von der Struktur Probleme. Ein weiteres Problem in Zusammenhang mit der semantikbasierten Konstruktion von Übersetzern ist, daß Semantikbeschreibung und Programm in dieser Art von Semantiken vermischt sind. Wir sind jedoch an einer klaren Trennung von Interpretation und Programm interessiert, da der Übersetzer selbst nur auf dem Programm arbeitet. Die Modellierung indeterministischer Auswertungsfolgen kann durch Aufzählung aller Möglichkeiten modelliert werden. Allerdings kann damit nicht die beliebige Verzahnung von Berechnungen spezifiziert werden.

ASMs basieren auf einfachen mathematischen Konzepten und die operationelle Sicht kommt dem intuitiven Verständnis des Informatikers entgegen. Durch die freie Wahl der Algebren, die den Zustand beschreiben, erlaubt die Methode die Definition beliebiger Abstraktionsebenen. Unendliche Ausführungen stellen genausowenig ein Problem dar, wie die Modellierung von Indeterminismus. Ein Vorwurf, der ASMs immer wieder gemacht wird ist, daß die Spezifikationen sehr maschinennah sind. Alle Semantikspezifikationen führen einen Befehlszähler ein und jeder Zustandsübergang ändert diesen Befehlszähler und den Speicher. Aus genau diesem Grund sind ASMs für die Definition von Zwischen- und Zielsprachen sehr geeignet, einen Eindruck geben z. B. (Gaul et al., 1995; Gaul und Zimmermann, 1995). Ein Nachteil von ASMs und *Montages* ist, daß der Formalismus zu viele Freiheiten erlaubt und die Syntax kaum eingeschränkt ist. Das hat zur Folge, daß zwei unabhängig von einander erstellte Spezifikatio-

5) Man nennt dies auch *interleaving*.

nen für dieselbe Sprache normalerweise syntaktisch nicht gleich sind. Semantische Gleichheit drückt sich nicht durch strukturelle Gleichheit aus. Für die Definition von Transformationsregeln ist es jedoch ein Entwurfskriterium, daß gleiche semantische Dinge auch auf das gleiche syntaktische Konzept abgebildet werden. Dieses Kriterium findet auch schon in der Analysephase Anwendung, wo z. B. unterschiedliche syntaktische Repräsentationen für Integers auf den gleichen Knotentyp abgebildet werden.

Nichtsdestotrotz erfüllen abstrakte Zustandsmaschinen zusammen mit Montages einen Großteil unserer Anforderungen und sind daher die formale Basis für unseren Ansatz zur Definition von Programmiersprachsemantiken. Die Probleme von Montages bei der Definition statischer Semantik können wir durch eine alternative Semantik auf der Basis von Attributgrammatiken beheben. Wir geben eine formale Definition in Zusammenhang mit der formalen Beschreibung von ASMs in Kapitel 4 an. Die Freiheiten von ASMs, die oft erwünscht sind, in unserem Kontext aber hindern, schränken wir durch eine speziell auf unser Anwendungsgebiet ausgelegte Semantikbeschreibungssprache ein. Wir definieren diese Einschränkung in Form der Sprache *AL* in Kapitel 5.

2.3 Verifizierte Konstruktion eines Übersetzers

Für die Konstruktion eines korrekten Übersetzers verwenden wir die von Chirica und Martin (1986) vorgeschlagene Aufteilung der Aufgabe. Zum formalen Nachweis der Korrektheit einer Übersetzung[6] geht man wie folgt vor.

1. Formale Definition der Semantik der Quell- und Zielsprache und Validierung der Spezifikation.
2. Spezifikation der Transformation.
3. Verifikation der Transformation gegenüber den Semantiken der beteiligten Sprachen durch Nachweis der Verhaltensgleichheit von Quell- und resultierenden Zielsprachprogrammen.
4. Implementierung der Transformation.
5. Verifikation der Implementierung gegenüber der Transformationsspezifikation und der Semantik der Implementierungssprache.

Diese Struktur hat sich unter anderem auch im *Verifix*-Projekt zur Konstruktion korrekter Übersetzer (DFG-Nummer: GO323/3-3) erfolgreich bewährt (Goerigk et al., 1996, 1998b).

Wir setzen für die Konstruktion voraus, daß die Hardware die formale Semantik der Zielmaschine korrekt implementiert und daß Betriebssystemaufrufe, z. B. zur Ein-/Ausgabe von Daten, korrekt implementiert sind. Die Korrektheit eines Übersetzers sagt nichts über die Korrektheit von Quellprogrammen aus. Der Nachweis, daß ein Quellprogramm die intendierte Semantik bzw. eine gegebene Spezifikation korrekt implementiert, ist die Aufgabe des Programmierers.

Um eine Übersetzung der Quell- in die Zielsprache korrekt zu definieren, gibt es zwei unterschiedliche Ansätze.

6) Mit „formal“ meinen wir formal im mathematischen Sinne. Will man Werkzeuge zur Verifikation einsetzen, dann bedeutet formal jedoch maschinell verarbeitbar.

- Verfeinerung der Semantikspezifikation der Quellsprache bis hin zur Semantik der Zielsprache und Nachweis, daß jeder Verfeinerungsschritt korrekt ist.
- Spezifikation einer Übersetzung $\mathcal{C}$ der Quell- in die Zielsprache und dann der Nachweis, daß die erzeugten Zielprogramme die Semantik der Quellprogramme beschreiben.

Der Verfeinerungsansatz ist aus Verifikationssicht geschickt, da man existierende Verfeinerungstheorien nutzen kann. Allerdings werden diese Verfeinerungen unter Verifikationsgesichtspunkten definiert und haben nichts oder nur wenig mit den Transformationsschritten zu tun, die in einem traditionell konstruierten Übersetzer ausgeführt werden. Die Struktur der entstehenden Übersetzer ist monolithisch. Das schließt Optimierungen aus, die Codeumordnungen ausnutzen. Da mit diesem Ansatz kein effizienter Maschinencode erzeugt werden kann, gehen wir auch nicht weiter darauf ein.

Der zweite Alternative orientiert sich am Bau unverifizierter Übersetzer und ist die Methode der Wahl. Sie erlaubt die Einführung von Zwischensprachen und es entstehen Übersetzer, die von der Struktur identisch mit unverifizierten Übersetzern sind, die Optimierungen erlauben und daher die Erzeugung effizienten Maschinencodes nicht ausschließen.

Die Implementierung der Übersetzungsspezifikation in einer höheren Implementierungssprache und die Verifikation der Implementierung gegenüber der Spezifikation mit Techniken der Programmverifikation schließen den Prozess ab. Diese Aufgabe ist zwar gut verstanden, aber nichtsdestotrotz aufwendig und fehleranfällig, vor allem, wenn man bedenkt, daß es sich bei Übersetzern um sehr große Programme handelt.

Für die Verifikation der Übersetzung benötigt man einen exakten Korrektheitsbegriff, den wir im nächsten Abschnitt einführen. Anschließend stellen wir die prinzipiellen Techniken zur Verifikation der Transformationsspezifikation und zur korrekten Implementierung dieser Spezifikation vor.

2.3.1 Ein praktikabler Korrektheitsbegriff

Der Korrektheitsbegriff für Übersetzungen ist für die Definition realistischer Übersetzer essentiell. Er muß die Definition von Optimierungen erlauben, für indeterministische Programme geeignet sein und auch Ressourcenbeschränkungen berücksichtigen.

Die Ausführung eines Programms kann durch die Ausführung eines anderen Programms ersetzt werden, wenn beide die gleichen Resultate erzielen. Wir sagen dann, daß beide Programme das gleiche Verhalten zeigen. Realzeiteigenschaften werden von uns dabei nicht berücksichtigt. Die Ausführung eines Programms kann durch eine Folge von Zuständen $q_0, q_1, q_2, \ldots$ beschrieben werden, die in einem Initialzustand q_0 beginnt. Der Zustandsraum setzt sich für ein Programm π einer Quellsprache QS aus den Variablen, den Konstanten, den Objekten usw. zusammen, die das Programm erzeugt und auf die es zugreift. Für ein Zielprogramm π' einer Maschinensprache ZS besteht der Zustandsraum aus den Registern des Prozessors und dem Speicher der während der Programmausführung zugegriffen wird. Der Zustandsraum hängt in beiden Fällen von der Eingabe und einem möglicherweise indeterministischen Verhalten ab.

In ausgezeichneten Zuständen der Ausführungsfolge findet eine Kommunikation mit der Umgebung eines Programms statt. Dabei werden Daten ein- oder ausgegeben. Ein außenstehender Betrachter kann zwei unterschiedliche Ausführungen von Programmen nicht unterscheiden, solange er immer die gleiche Interaktion mit der Umwelt beobachtet. Das bedeutet aber, daß die Details der Berechnungen für einen Betrachter nicht von Interesse sind und sich beliebig

unterscheiden dürfen. Wir können daher die Programmausführung durch die entsprechende Folge beobachtbarer Zustände abstrahieren, die das beobachtbare Verhalten beschreiben. Für ein korrekt übersetztes Programm π' verlangen wir nur, daß es exakt das beobachtbare Verhalten des ursprünglichen Programms π nachbildet (simuliert). Speziell können damit auch Übersetzungen als korrekt nachgewiesen werden, die Codeumordnungen machen, wie sie in optimierenden Übersetzern vorkommen. Ein Zielprogramm darf auch nicht mehr beobachtbares Verhalten zeigen als das Quellprogramm, da sonst Trojanische Pferde eingeführt werden könnten, die Dritte über die Ausführungen eines Programms informieren oder sonstiges unerwünschtes Verhalten verursachen (HOFFMANN, 1998).

Ein Quell- und ein Zielprogramm mit gleichem beobachtbaren Verhalten durchlaufen eine Reihe von beobachtbaren Zuständen. Zwischen entsprechenden Zuständen q und q' muß es eine eineindeutige Relation ρ geben, die die beobachtbaren Teile des Zustandsraums der Quelle und des Ziels bzw. ihre Werte in Beziehung setzt, siehe Abb. 2.1. Indem wir diese Relation

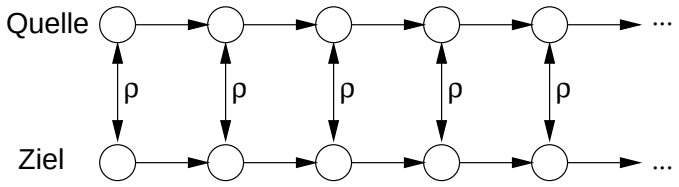


Abbildung 2.1: In Beziehung stehende beobachtbare Zustände

ρ angeben und damit einen Simulationsbeweis führen, wie er aus der Komplexitäts- und Berechenbarkeitstheorie bekannt ist (VAN LEEUWEN, 1990; SCHÖNING, 1995), weisen wir nach, daß ein Zielprogramm π' das beobachtbare Verhalten eines Quellprogramms π simuliert. Der Nachweis ist leicht zu führen, falls ein Programm π einen Algorithmus implementiert, der nach einer bestimmten Anzahl von Berechnungsschritten terminiert und dann die Resultate für eine bestimmte Eingabe ausgibt. In diesem Fall sind nur der Initialzustand und der Finalzustand beobachtbar und die Relation ρ ist leicht zu definieren. Allerdings können sequentielle reaktive Programme unendlich viele beobachtbare Zustände durchlaufen, die betrachtet werden müssen. Dann ist die Definition der Relation ρ komplizierter.

Da wir imperative Programme betrachten, müssen wir zusätzlich Indeterminismus in Betracht ziehen. Bei der Ausführung eines Quellprogramms π kann indeterministisch ein *erlaubter* Berechnungspfad $q_0, q_1, \ldots$ ausgewählt werden. In diesem Fall kann sich der Übersetzer beliebig entscheiden, welche Berechnungsfolge er implementiert. Der Indeterminismus braucht im Zielprogramm nicht mehr enthalten sein. Für das Zielprogramm π' verlangen wir daher nur, daß seine Ausführungsfolge $q'_0, q'_1, \ldots$ das gleiche beobachtbare Verhalten zeigt, wie *eine* erlaubte Berechnungsfolge des Quellprogramms. Zeigt andererseits das Zielprogramm π' ein indeterministisches Verhalten, dann muß dieser Indeterminismus auch im Quellprogramm vorkommen. Wir betrachten nur indeterministisches Verhalten auf der Ebene beobachtbarer Zustände.

Da wir realistische Übersetzer bauen wollen, muß unser Korrektheitsbegriff auch die Ressourcenbeschränkungen von Zielmaschinen berücksichtigen. Wir erlauben, daß ein Zielsprachprogramm mit einem Fehler wegen Ressourcenbeschränkungen terminieren darf, auch wenn das Quellprogramm fehlerfrei läuft. Ein Ressourcenfehler liegt zum Beispiel vor, wenn der Speicher während der Ausführung überläuft. Ist es nicht möglich die Genauigkeit der Arithmetik der Quellsprache auf der Zielmaschine nachzubilden, dann ist dies auch ein Ressourcenfehler. Allerdings muß der Programmierer oder der Benutzer sicherstellen, daß Eingaben oder

Berechnungen bestimmte Eigenschaften erfüllen, um unerwünschtes Verhalten aufgrund von Rundungsfehlern oder sonstigen Bereichsverletzungen zu vermeiden. Der Übersetzer kann für solche Fehler nicht verantwortlich gemacht werden.

Definition 2.1 (Korrekte Übersetzung)
Ein Zielprogramm π' ist genau dann eine korrekte Übersetzung eines Quellprogramms π, wenn für alle erlaubten Eingaben einer der folgenden Fälle gilt:

1. *Zu jeder Folge beobachtbarer Zustände $q'_0, q'_1, q'_2, \ldots, q'_k$ von π', die regulär terminiert, gibt es eine Berechnungsfolge $q_0, q_1, q_2, \ldots, q_k$ von π mit $q_i \ \rho \ q'_i$ für $0 \leq i \leq k$.*
2. *Zu jeder nicht terminierenden Folge beobachtbarer Zustände $q'_0, q'_1, q'_2, \ldots$ von π' gibt es eine Berechnungsfolge $q_0, q_1, q_2, \ldots$ von π mit $q_i \ \rho \ q'_i$ für alle i.*
3. *Zu jeder Folge beobachtbarer Zustände $q'_0, q'_1, q'_2, \ldots, q'_{k+1}$ von π', die mit einem Ressourcenfehler terminiert, gibt es eine Berechnungsfolge $q_0, q_1, q_2, \ldots, q_k, \ldots$ von π mit $q_i \ \rho \ q'_i$ für $0 \leq i \leq k$.*

Bemerkung: Die Definition korrekter Übersetzung ist absichtlich partiell, weil wir die Forderung nach totaler Korrektheit eines Übersetzers für nicht erfüllbar halten. C-Programme mit 10^{20} Zeilen können zwar theoretisch korrekt übersetzt werden, aber in der Praxis scheitert die Übersetzung, weil die vorhandenen Betriebsmittel nicht ausreichen. Mit obiger Definition ist ein Übersetzer, der alle Programme mit der Meldung „wegen beschränkter Betriebsmittel nicht übersetzbar “ zurückweist, im mathematischen Sinne korrekt. Natürlich ist dieser Übersetzer völlig unbrauchbar und das Ziel ist es, einen Übersetzer zu konstruieren, der möglichst viele Programme korrekt übersetzt. ♢

Entsprechend der Aufteilung in Spezifikation und Implementierung der Transformation aus dem vorigen Abschnitt zerfällt die Verifikation der Übersetzung in zwei Teile: die Verifikation der Übersetzungsspezifikation und die Verifikation der Implementierung dieser Spezifikation. Abbildung 2.2 veranschaulicht den Begriff der Korrektheit der Spezifikation und der Korrektheit der Implementierung. Die Übersetzungsspezifikation ist korrekt, wenn das linke Diagramm kommutiert.

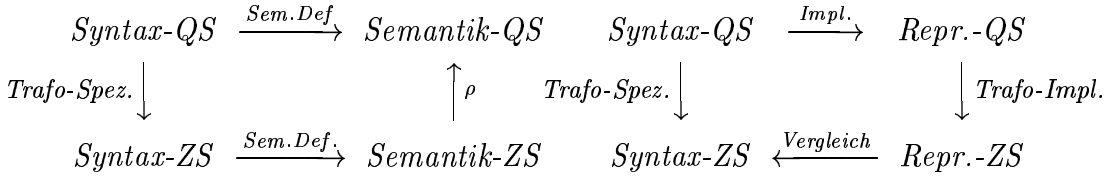


Abbildung 2.2: Korrektheit von Übersetzungsspezifikation und Übersetzungsimplementierung

Der hier beschriebene Korrektheitsbegriff wird so auch im Verifix-Projekt verwendet. Eine ausführliche Abhandlung zu praktisch relevanter Übersetzungskorrektheit findet man in (Goos und Zimmermann, 1999) und (Goerigk et al., 1996). Die für diese Arbeit gültige formale Definition von Übersetzungskorrektheit auf der Basis von Ausführungsfolgen abstrakter Zustandsmaschinen geben wir in Kapitel 6 an.

2.3.2 Verifikation der Übersetzungsspezifikation

Die Korrektheitsbeweise für Übersetzungen können sehr komplex werden. Modularisierung verringert die Komplexität der Einzelbeweise und erlaubt den Einsatz von Werkzeugen zum

Beweisen bzw. zur Beweisüberprüfung. Für die Zerlegung des Korrektheitsbeweises haben wir zwei Möglichkeiten:

1. Wir führen zusätzliche Zwischensprachen ein und können dann gezielt einzelne Aspekte übersetzen und die Transformation verifizieren. Wir nennen diese Technik **vertikale Komposition**.

2. Für die zweite Möglichkeit betrachten wir die Ausführungsebene von Programmen. Seien π_1 und π_2 Teilprogramme der Quellsprache, die korrekt in π_1' bzw. π_2' übersetzt werden. Dann ist auch $\pi_1'; \pi_2'$ eine korrekte Übersetzung der sequentiellen Ausführung von $\pi_1; \pi_2$. Diese Art der Komposition bezeichnen wir als **horizontale Komposition**.

Abbildung 2.3 veranschaulicht horizontale und vertikale Komposition auf der Ebene von Zustandstransformationen. Dabei sind q_i Zustände, die bei der Ausführung von Programmen auftreten. Die ρ_i setzen Zustände von Quelle und Ziel in Relation und induzieren damit die Simulation von Verhalten. Vertikale Komposition definiert aus Verifikationsgründen zusätzliche

$$\begin{array}{ccccc}
q_1 \in Q_{QS} & \xrightarrow{[\![\pi_1]\!]_{QS}} & q_1' \in Q_{QS} & \xrightarrow{[\![\pi_2]\!]_{QS}} & q_1'' \in Q_{QS} \\
\rho_1 \uparrow & & \rho_1 \uparrow & & \rho_1 \uparrow \\
q_2 \in Q_{ZR} & \xrightarrow{[\![\pi_1']\!]_{QS}} & q_2' \in Q_{ZR} & \xrightarrow{[\![\pi_2']\!]_{ZR}} & q_2'' \in Q_{ZR} \\
\rho_2 \uparrow & & \rho_2 \uparrow & & \\
q_3 \in Q_{ZS} & \xrightarrow{[\![\pi_1'']\!]_{ZS}} & q_3' \in Q_{ZS} & &
\end{array}$$

Abbildung 2.3: Vertikale und horizontale Komposition von Übersetzungen

Übersetzungsschritte nur zu dem Zweck, den Beweis einfacher zu gestalten bzw. bestimmte Aspekte einzeln betrachten zu können. In der konkreten Implementierung eines Übersetzers können diese Übersetzungen zusammen mit anderen Übersetzungen ausgeführt werden. Allerdings muß dann nachgewiesen werden, daß sich die separate und die integrierte Ausführung gleich verhalten. Es ist zu zeigen, daß die betreffenden Übersetzungen orthogonal sind. Horizontale Dekomposition bedeutet die Zerlegung einer Folge von Zustandsübergängen in Teilfolgen, die die Semantik von einzelnen Sprachkonzepten der Sprache repräsentieren.

Die Korrektheit der Übersetzung einzelner Sprachkonzepte hängt von der Definition des jeweiligen Sprachkonzepts und der entsprechenden Transformation ab. Reihenfolgebedingungen und Kontexte ergeben sich, wenn es Abhängigkeiten zwischen Übersetzungen bzw. zwischen den übersetzten Konzepten gibt. Die Übersetzung kann sich an der Struktur der Semantik orientieren, das ist jedoch nicht zwingend. Es gibt zusätzliche Möglichkeiten:

1. Die Transformation führt Optimierungen durch. Dabei werden unter Umständen Teilbäume eliminiert oder umgruppiert und dadurch die Struktur verändert.

2. Die Transformation macht implizite Fallunterscheidungen der Semantik explizit und erweitert dabei die ursprüngliche Struktur um zusätzliche Alternativen.

3. Die Transformation zerstört die Struktur, indem Teilbäume durch Kombinationen von spezialisierten Mustern ersetzt werden.

Die Definition der Quellsprachsemantik und die Definition von Übersetzungen bestimmen die Techniken für Korrektheitsbeweise. Ist die Semantik durch Komposition über den Semantiken

von Teiltermen definiert, dann bietet es sich an, Korrektheit durch strukturelle Induktion über den Aufbau der Terme nachzuweisen. Ist die Semantik einer Quellsprache operationell durch eine Interpretation über Termen in beliebigen Kontexten definiert, dann eignet sich ein Korrektheitsbeweis durch Induktion über die Anzahl der Zustandsübergänge der abstrakten Maschine.

Die Tatsache, daß unser Ansatz an die klassische Übersetzerarchitektur angelehnt ist (WAITE und GOOS, 1984; AHO ET AL., 1986) erlaubt die Anwendung existierender Datentypen und Übersetzungstechniken, wie z. B. Definitionstabelle und Termersetzungen, sowie die Einführung zusätzlicher Zwischensprachen $ZR_1, ZR_2, \ldots$. Die Konstruktion kann auf jeden Teilübersetzungsschritt $ZR_i \rightarrow ZR_{i+1}$ angewendet werden. Daß diese vertikale Komposition von Übersetzungen nicht die globale Korrektheitsaussage zerstört, ergibt sich aus der Transitivitätseigenschaft von Verhaltensgleichheit.

Satz 2.2
Gegeben seien die Sprachen L_1, L_2 und L_3. Ist $\mathcal{C}_1$ eine korrekte Übersetzung von L_1 nach L_2 und übersetzt $\mathcal{C}_2$ L_2 korrekt in L_3 dann ergibt die Komposition $\mathcal{C}_2 \circ \mathcal{C}_1$ einen korrekten Übersetzer von L_1 nach L_3.

Beweis $\mathcal{C}_1$ ist eine korrekte Übersetzung der Sprache L_1 in die Sprache L_2, wenn es entsprechend Definition 2.1 zu jedem Programm $\pi \in L_1$ eine Relation ρ_1 gibt, die beobachtbare Zustandsübergangsfolgen von $\mathcal{C}_1(\pi) \in L_2$ mit den beobachtbaren Zustandsübergangsfolgen von π in Beziehung setzt. Für die Übersetzung $\mathcal{C}_2$ gibt es eine entsprechende Relation ρ_2. Die Simulation beobachtbaren Verhaltens ist transitiv. Daher definiert $\rho_2 \circ \rho_1$ eine Simulationsbeziehung zwischen Programmen $\pi' = \mathcal{C}_2(\mathcal{C}_1(\pi)) \in L_3$ und den entsprechenden Quellprogrammen $\pi \in L_1$. ♢

Die Einführung beliebiger Zwischensprachen ist sowohl unter Beweisgesichtspunkten, als auch unter Effizienzgesichtspunkten von Bedeutung. Einerseits können wir Übersetzungsschritte beliebig klein wählen und dadurch die Komplexität der Einzelbeweise reduzieren, andererseits können wir Zwischenrepräsentationen einführen, die für die Erzeugung effizienten Maschinencodes oder für Optimierungen benötigt werden. Eine weitere Modularisierung der Beweise ergibt sich, wenn wir die Teile der Ausführung von Programmen betrachten und die Übersetzung einzelner Anweisungen verifizieren.

2.3.3 Korrekte Implementierung

Für die korrekte Implementierung der Übersetzungsspezifikation $\mathcal{C}$ in einer höheren Implementierungssprache $\mathcal{L}$ gibt es mehrere Möglichkeiten.

- Eine **Implementierung von Hand** muß man gegenüber der Spezifikation mit Programmverifikationstechniken, vgl. HOARE (1969); DIJKSTRA (1976), als korrekt beweisen. Zur Verifikation benötigt man einen Semantikkalkül für die Implementierungssprache $\mathcal{L}$.
- **Programmprüfung** ist ein alternativer Ansatz, der zu einem korrekten Ergebnis der Übersetzung führt. Die grundlegende Idee dazu stammt von BLUM und KANNAN (1989). Anstatt die Korrektheit der Transformationsimplementierung zu verifizieren, überprüft man Eigenschaften des Ergebnisses, die hinreichend für die Korrektheit der Transformation sind. Kann der Überprüfer, der verifiziert sein muß, diese Eigenschaft nicht nachweisen, dann wird das Zielprogramm zurückgewiesen.

Mit Hilfe der Programmprüfung kann nur die partielle Korrektheit eines Übersetzers gezeigt werden. Allerdings ist das durchaus legal, da jeder existierende Übersetzer nur eine Teilmenge der theoretisch beschreibbaren Programme übersetzen kann, siehe auch die Bemerkung auf S. 35. Das Verfahren kann immer dann sinnvoll eingesetzt werden, wenn die Verifikation des Algorithmus schwieriger bzw. wesentlich aufwendiger ist als die Verifikation der Überprüfung von Korrektheitseigenschaften. Speziell im Übersetzerbau, wo hocheffiziente und komplexe Algorithmen eingesetzt und Optimierungen durchgeführt werden, bringt dieser Ansatz Vorteile. Außerdem, existiert ein verifizierter Überprüfer, dann können sogar Algorithmen und ihre Implementierungen einfach ausgetauscht werden, ohne daß die Korrektheit des Übersetzers neu gezeigt werden muß.

Pnueli et al. (1998) betrachten reaktive Systeme und stellen durch Überprüfung die korrekte Übersetzung von endlichen Automaten sicher. Allerdings bestehen die betrachteten Programme, bedingt durch die Anwendung, aus einer einzelnen Schleife, die eine Funktion von Ein- in Ausgaben implementiert. Das System berechnet automatisch Constraints an den Schleifenrumpf und prüft, ob diese eingehalten werden. In (Heberle et al., 1999) benutzen wir den Programmprüfungsansatz, um die Korrektheit der Analysephase eines Übersetzers zu garantieren. Diese Anwendung ist zusätzlich interessant, weil wir durch die Prüfung des Resultats erst in der Lage sind, die Korrektheit der Analysephase zu definieren. Goerigk et al. (1998a) beschreiben die Überprüfung des Ergebnisses der Codeerzeugungsphase und legen damit den Grundstein einer Generatortechnik für korrekte Übersetzer-Backends. Necula und Lee (1998) verbessern die Qualität ihres Übersetzers, indem sie automatisch notwendige Bedingungen für die Korrektheit von Assemblerprogrammen, z. B. Typsicherheit und Speichersicherheit, überprüfen. Allerdings liegt diesen Korrektheitsaussagen keine formale Semantik, sondern nur die Erfahrung des Übersetzerbauers zugrunde. Der wichtigste Unterschied zum Programmprüfungsansatz ist jedoch, daß dort hinreichende Bedingungen für die Korrektheit des Ergebnisses garantiert werden.

- **Generatoren zur Implementierung** von Transformationsspezifikationen sind der am weitesten automatisierte Ansatz. Ist der Generator als korrekt nachgewiesen und die angegebene Spezifikation der Transformation ebenfalls, dann ist auch die erzeugte Implementierung der Transformation korrekt. Kritisch bei diesem Ansatz ist jedoch die Korrektheit des Generators, der selbst wieder ein komplexes Softwaresystem darstellt und nicht so einfach zu verifizieren ist. Eine vielversprechende Idee ist die Kombination von Generatoren und Programmprüfern, vgl. (Heberle et al., 1999) und (Gaul et al., 1999).

Die Einführung einer allgemeinen Schnittstelle zwischen Quell- und Zielsprachen reduziert den Verifikationsaufwand für die Implementierung zusätzlich. Anstatt $n * m$ Übersetzer für n Quell- und m Zwischensprachen zu erzeugen, muß man nur n quellsprachabhängige und m zielsprachabhängige Übersetzer implementieren, die über die gemeinsame Schnittstelle kombiniert werden.

Die traditionelle Architektur von Übersetzern, von der wir wissen, wie sie effizient implementiert werden kann, gibt weitere Anforderungen an die Implementierung vor. Zum Beispiel wird üblicherweise Namensinformation in einer Definitionstabelle abgelegt, die effizienten Zugriff erlaubt. Desweiteren „weiß“ der versierte Übersetzerbauer, welche Transformationen geschickterweise zusammen und in welcher Reihenfolge Übersetzungen ausgeführt werden. Allerdings steht die Effizienz der Übersetzung für diese Arbeit erst an zweiter Stelle; Modularisierungs-

und Korrektheitsfragen haben die höhere Priorität. Die gemischte Ausführung von Übersetzungen in der Implementierung sollte natürlich durch die Konstruktion und durch Berücksichtigung von Korrektheitsfragen nicht ausgeschlossen sein.

Jede Implementierung eines Übersetzers in einer höheren Programmiersprache $\mathcal{L}$ setzt voraus, daß auch ein korrekter Übersetzer für $\mathcal{L}$ existiert. Das bedeutet aber, daß man am Anfang einen initialen korrekten Übersetzer benötigt, der bis hinunter zum Maschinencode verifiziert ist. Diesen Übersetzer verifiziert in einer höheren Programmiersprache zu implementieren führt, ebenso wie die verifizierte Implementierung in Assembler, zu einem Zirkelschluß. Die Verifikation einer Maschinencodeimplementierung scheitert am Aufwand und an der Fehleranfälligkeit. Als einzig gangbarer Weg verbleibt die Verwendung eines unverifizierten Übersetzers mit zusätzlicher Überprüfung des Ergebnisses von Hand.

Dieses Problem wird im Rahmen des Verifix-Projekts behandelt (Dold et al., 1996b). Aus einer verifizierten Übersetzungsspezifikation wird durch Verfeinerung eine korrekte Implementierung in einer Hochsprache abgeleitet, die dann unverifiziert in Maschinensprache übersetzt wird. Mit einer speziellen Bootstrapping-Technik wird anschließend ein korrekt in Maschinensprache implementierter Übersetzer erzeugt (Hoffmann, 1998). Dabei werden, beginnend bei der Zielsprache und unter Benutzung mehrerer spezieller Zwischensprachen, kleine Übersetzungsschritte durch Inspektion des erzeugten Codes als mathematisch korrekt eingesehen. Dieses Vorgehen stellt eine Verschärfung der Codeinspektion dar, wie sie z. B. Balzert (1998) zur Qualitätssicherung verwendet. An der Universtität Kiel wurde auf diese Art ein Übersetzer konstruiert, der eine Untermenge von CommonLisp korrekt in den Binärcode des Transputers übersetzt. Aufbauend auf diesen initialen korrekten Übersetzer können durch Bootstrapping weitere Übersetzer erzeugt werden.

2.4 Anforderungen an einen Lösungsansatz

Die Analyse in den vorigen Abschnitten versetzt uns in die Lage, einen Anforderungskatalog aufzustellen, den ein Ansatz erfüllen muß, um korrekte Übersetzer zu konstruieren, die auch effizienten Maschinencode erzeugen.

Anforderung 2.1 (Formale Semantik)
Die Spezifikationen der beteiligten Sprachen müssen formal sein.

Ohne formale Semantik kann nichts formal über die Korrektheit einer Übersetzung ausgesagt werden. Daher ist diese Anforderung essentiell.

Anforderung 2.2 (Adäquate Formalismen)
Einfache Formalisierung von Eigenschaften der beteiligten Sprachen.

Der Formalismus zur Semantikspezifikation muß dafür geeignet sein, sowohl indeterministische Ausführungsfolgen, wie sie in imperativen Programmiersprachen vorkommen, z. B. bei der Berechnung von Ausdrücken in ANSI-C, als auch die Beschreibung nichtterminierender Programme, z. B. bei Steuerungssoftware, zu spezifizieren. Die Anwendung komplexer mathematischer Theorien, wie z. B. Kategorientheorie in denotationeller Semantik, ist nicht adäquat, da vom Übersetzerbauer, die notwendigen formalen Kenntnisse nicht erwartet werden können.

Anforderung 2.3 (Durchgängigkeit der Formalismen)
Für die Spezifikation der Semantik sollte ein Formalismus durchgängig verwendet werden.

Wir wollen sowohl Quell-, als auch Zwischen- und Zielsprachen mit dem gleichen Formalismus beschreiben. Das bedeutet, daß mit dem Formalismus auch die unterschiedlichen Konzepte der Sprachen formuliert werden können. Jeder Wechsel von Formalismen bedeutet eine Schnittstelle im Prozeß, durch die zusätzliche Fehlerquellen eingeführt werden. Sind Quell- und Zielsprache mit dem gleichen Formalismus spezifiziert, dann können Beweise immer im gleichen formalen Rahmen geführt werden.

Anforderung 2.4 (Modularisierung der Übersetzung und der Verifikation)
Die Übersetzung und ihre Verifikation müssen zerlegbar sein und die Komplexität von Beweisschritten sollte so wählbar sein, daß auch mechanische Beweisunterstützung möglich ist.

Die Modularisierung der Verifikation ist aus zwei Gründen wichtig. Erstens ist die Verifikation der Übersetzung von realen Programmiersprachen aus Komplexitätsgründen nicht in einem Stück möglich, und zweitens müssen sehr viele Eigenschaften nachgewiesen werden, die nicht schwer, aber aufwendig zu verifizieren sind. Die Voraussetzung für eine Modularisierung der Verifikation ist die Modularisierung der Übersetzung. Die Modularität der Übersetzung ist für die Erzeugung effizienten Maschinencodes essentiell, da kein effizienter Maschinencode erzeugt werden kann, wenn Optimierungen mit Codeumordnungen ausgeschlossen sind. Der Verfeinerungsansatz zur Herleitung einer korrekten Transformationsspezifikation ist aus diesem Grund ungeeignet, da eine monolithische Übersetzung Optimierungen ausschließt.

Anforderung 2.5 (Wahl der Zwischensprache)
Die Zwischensprache darf nicht von der Methode vorgegeben werden, sondern muß frei wählbar sein.

Die Qualität von Optimierungen hängt stark von der Zwischensprache ab, die zur Zielmaschine und der Codeerzeugungstechnik passen muß.

Die folgenden Anforderungen sind für die Konstruktion eine korrekten Übersetzers nicht zwingend, machen aber den Konstruktionsprozeß erst praktikabel.

Anforderung 2.6 (Modularität und Erweiterbarkeit von Semantikspezifikationen)
Die Spezifikation der formalen Semantik von Sprachen muß modular aufgebaut und erweiterbar sein.

Modularität bedeutet hier, daß man sich, z. B. bei der Spezifikation der Semantik einer Schleife, keine Gedanken über die Beschreibung der Umgebung machen muß. Das ist auch die Voraussetzung für eine Modularisierung der Verifikation. Unter Erweiterbarkeit einer Spezifikation verstehen wir, daß neue Sprachkonzepte zu einer schon existierenden Spezifikation hinzugefügt werden können, ohne alle existierenden Spezifikationen verändern zu müssen.

Anforderung 2.7 (Wiederverwendung)
Sprachspezifikationen, verifizierte Übersetzungen und deren korrekte Implementierung sollten wiederverwendet werden können.

Betrachtet man die Komplexität von Korrektheitsbeweisen für Übersetzungen oder stellt sich den Aufwand für die Implementierungsverifikation vor, dann wird klar, daß diese Anforderung essentiell für die praktikable Anwendung einer Methode ist. Die Erweiterung von Sprachspezifikationen geht mit dieser Anforderung Hand in Hand.

Den Anforderungskatalog nutzen wir im folgenden zur Bewertung existierender Arbeiten zur semantikbasierten Erzeugung von Übersetzern.

2.5 Systeme für semantikbasierte Übersetzung

In diesem Abschnitt untersuchen wir existierende Arbeiten zur semantikbasierten Übersetzung. Unter *semantikbasierter Übersetzung* verstehen wir, daß die Transformation aus der formalen Semantikspezifikation der Quell- und Zielsprache abgeleitet wird. Wir lassen offen, ob die Transformation verifiziert ist und ob sie automatisch erzeugt wird. Gilt letzteres, dann sprechen wir auch von *semantikbasierter Generierung* von Übersetzern.

2.5.1 Ansätze mit fester Quell- und Zielsprache

In (Börger und Durdanović, 1996) wird die korrekte Übersetzung von Occam in Transputer-Code dargelegt. Es wird eine ASM-Definition der Semantik von Occam und des Transputer-Instruktionssatzes definiert. Die Abbildung wird durch Verfeinerung von ASMs als korrekt nachgewiesen. Teile der Übersetzung wurden mit Hilfe des Werkzeugs KIV maschinell überprüft. In (Börger und Rosenzweig, 1994) wird die korrekte Übersetzung von Prolog nach WAM beschrieben. Auch in dieser Arbeit wird Verfeinerung als grundlegende Technik zur Definition korrekter Übersetzungen eingesetzt. Eine weitere Arbeit behandelt die Übersetzung von Java nach Java Virtual Machine Code (Börger und Schulte, 1998b). Zu dieser Spezifikation existiert auch eine funktionale Implementierung, die allerdings nicht verifiziert ist. Diese Arbeiten betrachten jeweils nur ein festes Quell-/Zielsprachpaar. Es werden nur Korrektheitseigenschaften an eine Übersetzung definiert, eine konkrete Spezifikation der Transformation und eine Implementierung der Übersetzung gibt es abgesehen von der Java-Übersetzung nicht. Die für uns zentrale Aussage dieser Arbeiten ist, daß abstrakte Zustandsmaschinen prinzipiell zur Spezifikation von Programmiersprachsemantiken und zur verifizierten Übersetzerkonstruktion geeignet sind. Die Sprachen wurden ohne vereinfachende Annahmen spezifiziert und übersetzt. Die Wiederverwendung von Semantik- und Übersetzungsspezifikationen wird nicht untersucht.

Desweiteren gibt es einige Übersetzer, die auf der Basis denotationeller Semantik für feste Quell-/Zielsprachkombinationen als korrekt nachgewiesen wurden, siehe z. B. (Polak, 1981). Die Betrachtung dieser Arbeiten macht deutlich, daß schon die Konstruktion verifizierter Übersetzer für ein festes Quellsprach-/Zwischensprachpaar eine schwierige und komplexe Aufgabe ist. Die Übersetzung wird durch die Verfeinerung der Quellsprachsemantik verifiziert. Die angewendeten Verfeinerungen sind aus Sicht der Verifikation motiviert. Daher unterscheiden sich die auftretenden Zwischensprachen von den Zwischensprachen, die in einem traditionellen Übersetzer zur Codeerzeugung oder als Basis für Optimierungen benutzt werden. Die entstandenen Übersetzer sind ineffizient und erzeugen ineffizienten Code.

In (Pnueli et al., 1998) wird die Korrektheit eines Übersetzers für Statecharts-Spezifikationen nach C-Code durch die Validierung des Ergebnisses sichergestellt. Bei diesem Ansatz wird automatisch nachgewiesen, daß bei der Übersetzung Eigenschaften der Ablaufsteuerung erhalten bleiben. Dazu werden aus einem Quellprogramm π und aus dem entsprechenden Zielprogramm π' Aussagen E und E' abgeleitet und dann wird durch direkten Vergleich, mit Modellprüfung oder mit Methoden der temporalen Logik nachgewiesen, daß π' die Eigenschaften von π erfüllt. Diese Technik ist gut zum Nachweis bestimmter High-

Level-Eigenschaften über Ablaufstrukturen geeignet und kann mechanisch angewendet werden, wobei die Validierung automatisch durchgeführt wird. Allerdings wird der Code nicht im Gesamten betrachtet. Das hat zur Folge, daß man zwar die Terminierung einer Schleife nachweisen kann, aber nicht, daß die Anzahl der Schleifendurchläufe im Zielprogramm und im Quellprogramm gleich ist. Ein weiteres Problem ist, daß die Komplexität der behandelbaren Programme durch die Zustandsexplosion bei der Modellprüfung beschränkt ist. Außerdem können nach den bisher veröffentlichten Arbeiten nur Ablaufstrukturen behandelt werden, die durch endliche Automaten beschreibbar sind, also z. B. nicht allgemeine rekursive Prozeduren.

2.5.2 Semantikbasierte Generierung

Bei der semantikbasierten Generierung von Übersetzern sind vor allem zwei neuere Arbeiten hervorzuheben, die zumindest in Teilen unsere Kriterien erfüllen.

Diehl (1996) stellt einen Ansatz vor, bei dem aus SOS-Spezifikationen der Quellsprache Übersetzer und abstrakte Maschinen generiert werden. Die Zwischensprache wird durch den Generierungsprozeß bzw. durch die Definition der Quellsprachsemantik festgelegt. Das bedeutet, man codiert in der Spezifikation der Quellsprachsemantik Eigenschaften, die für die Zwischensprache erfüllt sein sollen. Damit nimmt aber der Entwerfer der Sprache Aufgaben des Übersetzerbauers vorweg oder der Übersetzerbauer muß die Originalsprachspezifikation entsprechend transformieren, was schon einer Übersetzung ist, die verifiziert werden müßte. Messungen zeigen, daß der letztendlich erzeugte Maschinencode um 2-3 Größenordnungen langsamer als der Code kommerzieller C-Compiler ist. Die Transformationen, die bei der Generierung benutzt werden, sind verifiziert. Die Implementierung und Übersetzung des Generators ist unverifiziert. Die Arbeit von Diehl stellt eine Erweiterung des Ansatzes von Hannan (1994) dar, der ebenfalls aus operationellen Semantiken Übersetzer und abstrakte Maschinen erzeugt.

Pettersson (1995) beschreibt die Übersetzung von natürlicher Semantik in C-Code. Er hat eine Teilmenge von ML als Spezifikationssprache für natürliche Semantiken definiert, aus der effiziente C-Implementierungen abgeleitet werden können. Die Übersetzung von Quellsprachen in Maschinensprachen kann durch eine entsprechende Definition der Sprachsemantiken erfolgen. Dabei werden mehrere Zwischensprachen definiert, deren Aneinanderreihung einen Übersetzer ergeben. Bei diesem Vorgehen definiert der Sprachdesigner die Übersetzung und zwar genau für ein Quell-Zwischensprach- bzw. Quell-/Maschinensprachpaar. Verifikationsaspekte werden in dieser Arbeit nicht berücksichtigt, dafür ist der erzeugte Code effizient. Der existierende Generator wird vor allem zur Erzeugung von Typcheckern, Interpretern und semantischen Analysatoren benutzt. Diese Tatsache untermauert die Aussage in Abschnitt 2.2.2 über natürliche Semantiken.

Übersetzergeneratoren für denotationelle Semantikspezifikationen funktionieren alle nach dem gleichen Prinzip. Die semantischen Gleichungen der Spezifikation definieren eine Abbildung von abstrakten Strukturbäumen in λ-Ausdrücke. Bei der Generierung werden diese λ-Ausdrücke erzeugt und durch partielle Auswertung soweit wie möglich ausgewertet. Der reduzierte λ-Ausdruck beschreibt das Zielprogramm, das mit einer Eingabe dann weiter reduziert werden kann. Der erzeugte Code von Generatoren wie SIS (Semantics Implementation System) (Mosses, 1976, 1979) oder SPS (Semantics Prototyping System) (Wand, 1984) ist wesentlich zu ineffizient. Palsberg (1992) beschreibt einen verifizierten Übersetzer-Generator, der Aktionssemantiken verarbeitet und Übersetzer mit einer abstrakten RISC-Maschine als Zielcode erzeugt. Der Generator wurde erfolgreich auf eine Teilmenge von ADA angewendet. Die Zwischen- bzw. die Zielsprache ist fest. Der erzeugte Code ist im Vergleich zu unverifizierten

Übersetzern immer noch um 2 Größenordnungen langsamer (DIEHL, 1996). BROWN ET AL. (1992) stellten den Übersetzer-Generator *Actress* vor, um Aktionssemantiken in C-Code zu übersetzen. Auch hier ist der erzeugte Code nicht effizient. Die Korrektheit des Generators ist nicht nachgewiesen. Das Problem bei diesem Ansatz ist auch wieder die vordefinierte Zielsprache. Ist der erzeugte C-Code maschinennah, dann sind darin Eigenschaften einer bestimmten Zielmaschine codiert, ist der erzeugte Code abstrakt, dann hat man wieder ein Übersetzungsproblem.

POETZSCH-HEFFTER (1997) beschreibt ein werkzeuggestütztes Framework, das MAX-System, zur Generierung von sprachspezifischer Software aus der formalen Spezifikation von Programmiersprachen. Die statische Semantik einer Sprache wird durch eine Attributierung mit Hilfe von *Occurrence Algebren* definiert, dynamische Semantik wird durch Regeln einer ASM angegeben. Die Spezifikation einer Sprache funktioniert damit ähnlich wie mit unserem Ansatz. Hauptziel der Arbeit von Poetzsch-Heffter ist die Erzeugung von Prototypen zum Test einer Programmiersprache sowie die Erzeugung effizienter Komponenten wie Browsern, Debuggern oder Simulatoren. Die Korrektheit von Übersetzungen und Komponenten wird nicht betrachtet. Allerdings ist vorstellbar, daß das Framework in dieser Richtung erweitert werden kann.

2.5.3 Bewertung

Wir haben einen Auszug existierender Ansätze vorgestellt. Ein guter Überblick zur semantikbasierten Generierung von Übersetzern inklusive einer Bewertung findet sich in (DIEHL, 1996). Unter Berücksichtigung unserer Anforderungen gelten für die semantikbasierten Arbeiten mindestens einer der folgenden Kritikpunkte:

- Die Spezifikationsmethode ist nicht oder nur bedingt zur Spezifikation kompletter imperativer Quell- und Zwischensprachen geeignet (siehe Bewertung in Abschnitt 2.2.4).
- Die Übersetzung ist für genau ein Paar Quell-/Zwischensprache verifiziert durchgeführt.
- Der erzeugte Zielcode ist ineffizient, weil abstrakte Zwischen- bzw. Zielsprachen verwendet werden oder die Zwischensprache fest ist.
- Die Transformationen sind nicht verifiziert.

Keine der Arbeiten berücksichtigt die unterschiedliche Codierung und die unterschiedliche Bedeutung der Basisdatentypen in der Quell- und der Zielsprache. Es wird immer die mathematische Bedeutung für Integer und Float angenommen.

Die Bewertung existierender Ansätze ist in Tabelle 2.2 noch einmal zusammengefaßt. Die Tabelle gibt auch Auskunft über den Formalismus zur Definition der Semantik. Es sind folgende Bewertungskriterien aufgeführt.

1. Korrektheit der Übersetzungsspezifikation/-implementierung: $+ \mathrel{\hat{=}}$ nachgewiesen, $- \mathrel{\hat{=}}$ nicht nachgewiesen[7]
2. Sprachen: Semantik/Quelle/Ziel
3. Generalisierung: $+ \mathrel{\hat{=}}$ möglich, $- \mathrel{\hat{=}}$ nicht vorgesehen

7) Einen Sonderfall stellen NECULA und LEE (1998) dar. Die Korrektheit ihres Überprüfers ist nachgewiesen, jedoch ohne eine formale Semantikspezifikation der beteiligten Sprachen zu berücksichtigen.

4. Effizienz: $+ \mathrel{\hat{=}}$ Effizienz des erzeugten Codes in der Größenordnung von realen Übersetzern, $\circ \mathrel{\hat{=}}$ 1-2 Größenordnungen schlechter, $- \mathrel{\hat{=}} > 2$ Größenordnungen schlechter

2.6 Zusammenfassung

In diesem Kapitel haben wir die Eigenschaften untersucht, die ein Ansatz zur Konstruktion korrekter *und* effizienter Übersetzer erfüllen muß.

- Die Semantikbeschreibungen der am Übersetzungsprozeß beteiligten Sprachen müssen operationell sein, da sowohl die Quell-, als auch die Zielsprache in demselben Formalismus beschrieben werden sollen. Indeterminismus und Nichtterminierung sollten einfach beschrieben werden können, weil die Quellsprachen, die wir verarbeiten wollen, diese Eigenschaften definieren.
- Ein korrekter Übersetzer muß die traditionelle Struktur unverifizierter Übersetzer haben, um effizient zu sein und effizienten Maschinencode zu erzeugen. Verfeinerungsansätze führen nicht zu der gewünschten Effizienz des Maschinencodes.
- Es müssen unterschiedliche Zwischensprachen erzeugt werden können, da nur mit einer Zwischenrepräsentation, die zur Zielmaschine und dem Codeerzeugungsschema paßt, effizienter Code erzeugt werden kann.
- Die Verifikation muß modular aufgebaut sein, um die Komplexität der Beweise handhabbar zu machen.
- Die Implementierung muß generiert oder durch Wiederverwendung von Transformationsimplementierungen erstellt werden.

Existierende Systeme erfüllen maximal einzelne dieser Anforderungen. Die Semantiken sind denotationell spezifiziert und/oder die Zwischensprachen sind abstrakt oder es gibt nur eine feste Zwischensprache. Der Maschinencode, den die konstruierten Übersetzer erzeugen, ist verglichen mit unverifizierten Übersetzern nicht effizient. Im nächsten Kapitel stellen wir einen Ansatz vor, der alle geforderten Eigenschaften hat.

Arbeit	Korrektheit	Sprachen	Generalisierung Effizienz
Brown et al. (1992)	Spez: ?[a] Impl: –	Sem: Aktions Semantik QS: Nano-ML, Mini-Δ[b] ZS: C	Gen: + Eff: ∘
Palsberg (1992)	Spez: + Impl: –	Sem: Aktions Semantik QS: ADA$^-$[c] ZS: abstr. RISC-Maschine	Gen: + Eff: ∘
Hannan (1994)	Spez: – Impl: –	Sem: Natürliche Semantik QS: While[d], funkt. Sprache ZS: abstr. Maschine	Gen: + Eff: ?[a]
Pettersson (1995)	Spez: – Impl: –	Sem: Natürliche Semantik QS: Pascal, Java ZS: C	Gen: ∘[e] Eff: +
Diehl (1996)	Spez: + Impl: –	Sem: Natürliche Semantik QS: Mini-ML,Aktions Semantik ZS: Interpretierer	Gen: + Eff: ∘
Poetzsch-Heffter (1997)	Spez: ?[a] Impl: ?[a]	Sem: ASM QS: Mini-ML, C$^-$[f] ZS: Interpretierer	Gen: + Eff: ?[a]
Pnueli et al. (1998)	Spez: + Impl: –	Sem: Statecharts QS: Statecharts ZS: C	Gen: + Eff: ?[a]
Necula und Lee (1998)	Spez: ∘ Impl: –	Sem: – QS: C ZS: DEC-Alpha	Gen: – Eff: +
Pnueli et al. (1998)	Spez: + Impl: –	Sem: Statecharts[g] QS: Statecharts ZS: C	Gen: ∘ Eff: ∘

a) Es war keine Information zugänglich bzw. das wurde von dieser Arbeit nicht behandelt.
b) Mini-Δ ist eine Sprache mit Zuweisung, bedingter Anweisung, Schleife und Prozeduren mit Parametern, Datentypen sind Integer und Bool.
c) ADA$^-$ ist eine Teilmenge von ADA.
d) While ist die Beispielsprache aus (Nielson und Nielson, 1992).
e) Der Generator von Pettersson ist vor allem zur Erzeugung von Typcheckern und Analysatoren geeignet, zur Erzeugung von Übersetzern eignet sich der Ansatz nur bedingt.
f) C$^-$ ist eine Teilmenge von C.
g) Das System verarbeitet Statecharts-Spezifikationen und erzeugt daraus C-Implementierungen.

Tabelle 2.2: Bewertung existierender (automatisierter) Ansätze

3 Korrekte Transformationsphase

Im letzten Kapitel haben wir die Transformationsphase als den zentralen Teil des Übersetzers identifiziert, der die eigentliche Übersetzungsarbeit leistet und daher die Anforderungen an das Frontend vorgibt und die Eingabe für die Codeerzeugungsphase festlegt. Eine wichtige Feststellung war, daß die Analyse alle Informationen zu liefern hat, die für die Transformation benötigt werden. Dieses Kapitel beschreibt unseren Ansatz im Überblick. Im nächsten Abschnitt stellen wir unseren Prozeß zur Konstruktion einer korrekten Transformationsphase vor. Die folgenden Abschnitte stellen jeweils die Lösung eines Teilproblems der Konstruktion vor, die dann in einem der späteren Kapitel genau ausgearbeitet wird.

Zuerst stellen wir eine universelle Sprache zur formalen Spezifikation der dynamischen Semantik imperativer Programmiersprachen vor, die alle unsere Anforderung erfüllt. Dann beschreiben wir, wie der kompositioneller Aufbau der Semantik zur Modularisierung der Übersetzungsverifikation genutzt werden kann. Dabei spalten wir die Sprache in Teilsprachen auf und weisen nach, daß diese korrekt übersetzt und korrekt komponiert werden. Aus Beweissicht bedeutet dieses Vorgehen, daß wir die komplette Zustandsübergangsfolge eines Quellprogramms in Teilzustandsfolgen zerlegen, dann lokal die Simulation über diesen Teilzustandsfolgen beweisen und zusätzlich nachweisen, daß richtig komponiert wurde. Anschließend beschreiben wir die Verwendung unserer Spezifikationssprache für die Semantikspezifikation und für die Konstruktion korrekter Übersetzungen. Wir zeigen, wie eine Bibliothek wiederverwendbarer Spezifikationen von Teilsprachen und Übersetzungen definiert werden kann und präsentieren ausgewählte Techniken zur Wiederverwendung von Semantikspezifikation und Korrektheitsaussagen über Transformationsmuster. Ein einfaches Schema zur objektorientierten Implementierung der Bibliothek vervollständigt unseren Rahmen zur Konstruktion korrekter Übersetzer, die effizienten Code erzeugen können.

3.1 Die Konstruktion der Transformation

Die Konstruktion eines korrekten Übersetzers wurde schon in Abschnitt 2.3 beschrieben. Dieses Vorgehen wenden wir nun direkt auf die Konstruktion einer korrekten Transformationsphase an. Wir nehmen dazu an, daß das Frontend, welches der Transformationsphase vorgeschaltet ist, korrekt ist.

Der Ausgangspunkt für die korrekte Konstruktion der Transformationsphase ist die informelle Spezifikation der Semantik der Quellsprache. Aus dieser informellen Spezifikation wird eine formale Spezifikation abgeleitet und gegenüber der Originaldefinition validiert. Das ist nicht unsere Aufgabe und wir setzen voraus, daß eine korrekte formale Spezifikation der Semantik in Form einer abstrakten Zustandsmaschine (ASM) gegeben ist. In Kapitel 7 werden wir sehen, daß unser Rahmen auch die Definition dieser ASM unterstützt. Die ASM definiert einen Interpretierer von Quellprogrammen, die durch abstrakte Strukturbäume repräsentiert sind. Die Funktionen der ASM beschreiben den Zustandsraum. Ein Teil der Funktionen modelliert die statische Semantik eines Programms, d. h., diese Funktionen beschreiben den abstrakten

Strukturbaum (AST) und eine Menge von Attributen, die kontextsensitive Informationen repräsentieren. Dazu gehören sowohl der Verweis von der Verwendung eines Bezeichners auf seine jeweils gültige Definition und die üblichen Typinformationen über Operanden und Operatoren als auch Datenabhängigkeiten und Steuerflußinformationen, die der Interpretierer benötigt. Die komplette formale Spezifikation einer Sprache inklusive abstrakter Syntax, statischer und dynamischer Semantik wird durch eine Montages-Spezifikation angegeben. Von der Transformationsphase werden nur Programme verarbeitet, die eine definierte statische Semantik haben, fehlerhafte Programme werden von der semantischen Analyse zurückgewiesen. Zusätzliche Attribute, die für die Transformation benötigt werden und die nicht in den statischen Funktionen der ASM modelliert sind, werden in der Analysephase berechnet.

Die formale Semantik der Zwischen- bzw. Zielsprache stellt das andere Ende des Übersetzungsprozesses dar. Sie ist nach dem gleichen Schema, wie die Semantik der Quellsprache definiert und wird üblicherweise vom Übersetzerbauer angegeben, der solche Sprachen auch entwirft. Die abstrakte Struktur über der die formale Semantik definiert wird, ist ein Graph und leitet sich aus der Struktur der Zielsprache ab. Mit Angabe der Zwischensprache bestimmt der Übersetzerbauer auch, welche Transformationen durchgeführt werden müssen.

Zur Definition von Semantik benutzen wir in dieser Arbeit nicht den Formalismus der abstrakten Zustandsmaschinen in seiner ganzen Allgemeinheit, sondern wir beschränken die Sprache, mit der die Semantik angegeben werden darf. Unsere Sprache schränkt nicht die Freiheitsgrade bei der Spezifikation dynamischer Semantik ein, sie ist jedoch speziell auf die Spezifikation der Semantik imperativer Programmiersprachen zugeschnitten und erlaubt es, mächtigere, allgemeine Aussagen zu beweisen.

Entsprechend der Aufteilung in Kapitel 2.3 definieren wir eine Übersetzungsspezifikation über den Strukturen der Quelle (dem attributierten abstrakten Strukturbaum (AAST)) und dem Ziel (Graph) in Form von bedingten Term- bzw. Graphersetzungsregeln. Für die Korrektheit der Übersetzungsspezifikation zeigen wir durch Simulationsbeweise, daß die erzeugten Zielprogramme das beobachtbare Verhalten (siehe Abschnitt 2.3.1) der Quellprogramme erhalten. Die Modularisierung dieses Beweises ist dabei ein entscheidender Punkt, auf den wir in Abschnitt 3.3 genauer eingehen.

Die korrekte Implementierung der Übersetzungsspezifikation ist Voraussetzung, um einen vollständig korrekten Übersetzer zu erhalten. In dieser Arbeit geben wir ein Schema für die objektorientierte Implementierung der Übersetzungsspezifikation an, aus dem auch Vor- und Nachbedingungen für die Verifikation abgeleitet werden können (siehe Abschnitt 3.7). Für einzelne Teile der Implementierung, z. B. bei der Speicherabbildung, kann die Korrektheit auch durch Laufzeit-Programmprüfung sichergestellt werden. Die Implementierungsverifikation mittels Hoare-Kalkül ist zwar aufwendig und mühsam, aber gut untersucht und wird daher von uns nicht behandelt. Die Techniken zur Laufzeitprogramm-Prüfung sind ein interessantes und vielversprechendes Gebiet, siehe z. B. (Goerigk et al., 1998a; Heberle et al., 1999), gehen aber auch über den Rahmen dieser Arbeit hinaus. Unser Vorgehen setzt voraus, daß ein korrekter Übersetzer für die Implementierungssprache existiert, sieh dazu auch Abschnitt 2.3.3.

Die beschriebenen Schritte führen zu der Architektur zur Konstruktion eines korrekten Übersetzers in Abb. 3.1. Dabei entsprechen Rechtecke Spezifikationen, Ellipsen stellen Semantiken dar, Kästen mit abgerundeten Ecken sind Repräsentationen von Programmen und Parallelogramme sind Implementierungen. Diese Architektur ist sehr eng an die klassische Überset-

zerarchitektur (WAITE und GOOS, 1984; AHO ET AL., 1986) angelehnt. Daher können die traditionellen Übersetzerbautechniken und Datentypen, wie z. B. Definitionstabelle und Termersetzungen angewendet sowie zusätzliche Zwischensprachen $ZR_1, ZR_2, \ldots$ eingeführt werden.

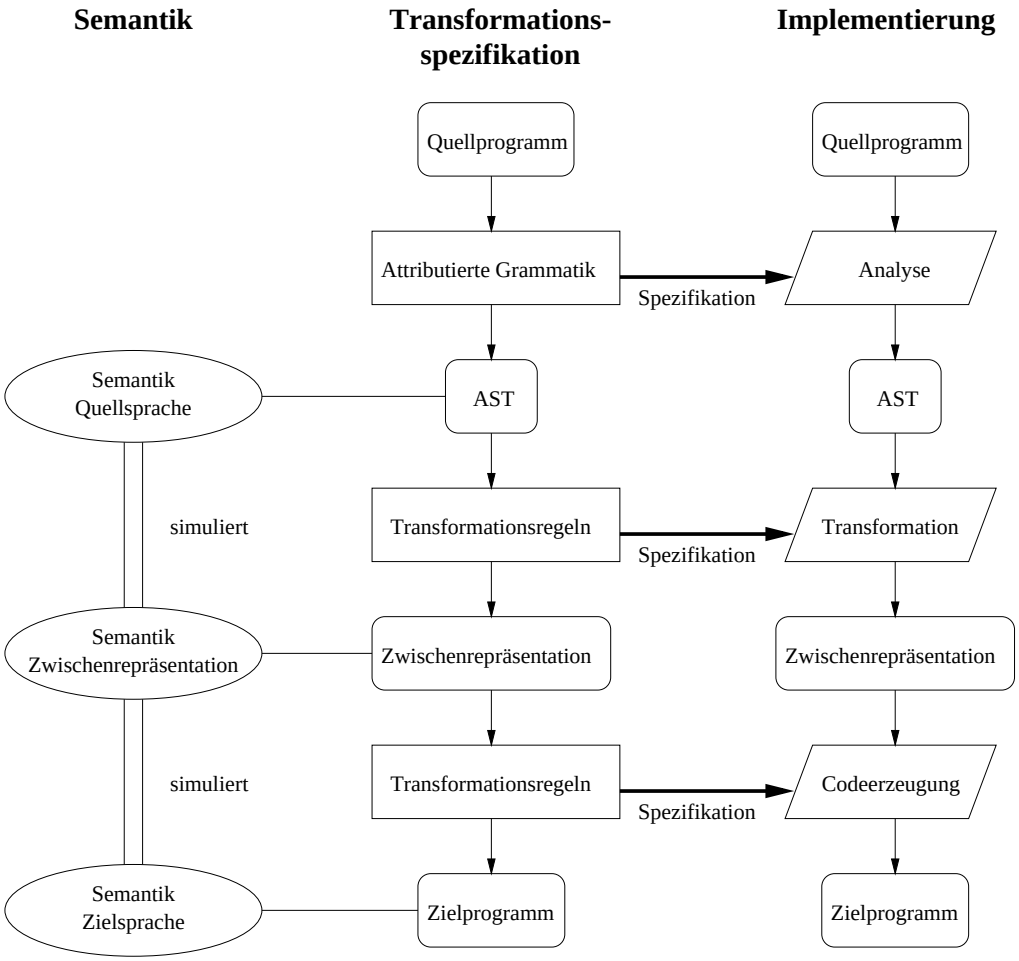


Abbildung 3.1: Architektur zur korrekten Übersetzerkonstruktion

Die Konstruktion kann auf jeden Teilübersetzungsschritt $ZR_i \rightarrow ZR_{i+1}$ angewendet werden. Daß diese vertikale Komposition von Übersetzungen nicht die globale Korrektheitsaussage zerstört, ergibt sich aus der Transitivitätseigenschaft von Verhaltensgleichheit, vgl. Satz 2.2. Mit der Möglichkeit zur Einführung beliebiger Zwischensprachen beheben wir sowohl unter Verifikationsgesichtspunkten (Modularisierung) als auch unter Codeerzeugungsgesichtspunkten (Effizienz des Codes) ein Defizit existierender Ansätze (siehe Abschnitt 2.3.2).

Neben der Übersetzungsspezifikation, die wir bisher betrachtet haben, gibt es auch noch eine **Übersetzerspezifikation**, die die Zerlegung des Übersetzers in seine einzelnen Teile angibt. Verwenden wir in der Konstruktion jeweils exakt die Ausgaben des vorangehenden Teils als Eingaben des nachfolgenden Teils und können damit die Korrektheit der Übersetzung nachweisen, dann liefert die Gesamtkonstruktion einen korrekten Übersetzer. Die Zerlegung des Übersetzers entsprechend der Übersetzerspezifikation ist durch die verwendeten Techniken im traditionellen Übersetzerbau motiviert. Zum Beispiel werden in der semantischen Analyse strukturelle Eigenschaften, also eigentlich syntaktische Eigenschaften, überprüft, die nicht kontextfrei beschrieben und daher nicht effizient zerteilt werden können. Genauso könnten

Teile der dynamischen Semantik durch Analysen statisch vorberechnet werden und würden damit zur statischen Semantik gehören. Die Übersetzerspezifikation legt fest, welche dieser Eigenschaften effizient während der semantischen Analyse berechnet werden können. Daraus folgt jedoch, daß die Spezifikation der dynamischen Semantik nicht unabhängig von der Übersetzerspezifikation ist, obwohl sie das eigentlich sein sollte. Legt der Übersetzerbauer die Sprachelemente der Quellsprache fest, d. h., definiert er die abstrakte Syntax und bestimmt, was in der semantischen Analyse berechnet wird, dann kann das Ergebnis ein attributierter AST sein, der unterschiedlich zu dem AST der formalen Semantik ist. In diesem Fall wäre das ein zusätzlicher Übersetzungsschritt, der verifiziert werden muß. Im folgenden setzen wir jedoch voraus, daß die Übersetzerspezifikation und die formale Semantik dieselbe Zerlegung implizieren.

3.2 Eine universelle Sprache zur Spezifikation dynamischer Semantik

In der Analyse formaler Semantiken in Kapitel 2 hat sich der ASM-Formalismus als die für unsere Zwecke erste Wahl herauskristallisiert. Mit ASMs können modulare und erweiterbare Spezifikation der dynamischen Semantik von imperativen Programmiersprachen adäquat angegeben werden. Die existierenden Beschreibungen von Java (Börger und Schulte, 1998c), der Java Virtual Machine (Börger und Schulte, 1998a) und die Spezifikationen für C und C++ (Gurevich und Huggins, 1993; Wallace, 1994) zeigen, daß der Formalismus zur operationellen Beschreibung der dynamischen Semantik imperativer Programmiersprachen gut geeignet ist. Die Untersuchung der existierenden ASM-Sprachspezifikationen hat zusätzlich ergeben, daß allen dasselbe abstrakte Maschinenmodell zur Interpretation von Programmen zugrundeliegt. Das nutzt der Montages-Formalismus aus, um Sprachen komplett mit konkreter Syntax, statischer und dynamischer Semantik auf der Basis von ASMs zu spezifizieren. Nichtsdestotrotz sind ASMs eigentlich zur einfachen Beschreibung von Algorithmen entwickelt worden. Der Formalismus und die zugrundeliegenden mathematischen Konzepte sind sehr mächtig und die Syntax ist wenig eingeschränkt; eine freie Wahl von Abstraktionen bei der Spezifikation ist möglich. Das ist aber für unser Ziel, den Nachweis möglichst mächtiger allgemeingültiger Aussagen zur Vereinfachung der Korrektheitsbeweise für Transformationen, kontraproduktiv. Aufgrund der Allgemeinheit von ASMs können nur wenige allgemeingültige Theoreme über Transformationen formuliert werden. Wir beheben dieses Problem, indem wir die spezielle Sprache *AL* einführen, die genau auf die Spezifikation der dynamischen Semantik imperativer Programmiersprachen zugeschnitten ist und mit der wir für uns wichtige allgemeingültige Aussagen zu Übersetzungen nachweisen können.

An die Sprache *AL* haben wir unterschiedliche Anforderungen:

1. *AL* sollte alle wichtigen semantischen Konzepte imperativer Programmiersprachen unterstützen, damit die Spezifikation einfach möglich ist.
2. Wir wollen die bewährte Struktur existierender ASM- bzw. Montages-Spezifikationen beibehalten, die eine modulare und erweiterbare Definition von Programmiersprachsemantiken ermöglicht.

Für die erste Anforderung sind übersetzerbautechnische Eigenschaften entscheidend. Programmiersprachen werden auf von Neumann-Architekturen abgebildet. Daher basieren Zielsprachen

alle auf den gleichen semantischen Konzepten, die von Zielmaschine zu Zielmaschine noch unterschiedlich ausgeprägt und komponiert werden können, die jedoch im Kern identisch sind. Grundlage der Berechnungen einer Zielmaschine sind Operationen auf Werten die durch Basisdatentypen wie *Int, Float* und *Bool* beschrieben sind. Die Resultate von Berechnungen können in einem Speicher, der eine Funktion *Objekt* $\rightarrow$ *Wert* definiert, für eine spätere Verwendung gesichert werden. Es existieren Anweisungen, mit denen Objekte erzeugt und ihre Werte gelesen und verändert werden können. Teilweise kann statisch vorhergesagt werden, welche Objekte während der Programmausführung benötigt werden, teilweise ist das laufzeitabhängig. Das führt zu einer Partitionierung des Speichers in einen Keller für statisch erzeugte und eine Halde für dynamisch erzeugte Objekte. Desweiteren gibt es Anweisungen, die eine sequentielle Reihenfolge über Berechnungen oder die bedingte bzw. unbedingte Verzweigung in der Berechnungsfolge erlauben. Auch die Abstraktion von Berechnungen in Form von Funktionen und Funktionsaufrufen erlauben alle Zielmaschinen. Da unsere Sprache *AL* alle diese Konzepte unterstützt, können wir vom praktischen Standpunkt alle imperativen Programmiersprachen beschreiben. Vom theoretischen Standpunkt ist *AL* turingmächtig. Die exakte Definition des abstrakten Maschinenmodells von Zielsprachen geben wir in Abschnitt 5.2.

Aus der zweiten Anforderung folgt, daß *AL* das Maschinenmodell existierender Sprachspezifikation verwendet. Dieses Maschinenmodell ist eng an das Maschinenmodell der Zielmaschine angelehnt. Es gibt einen abstrakten Befehlszähler der entlang statisch vorberechneter Steuerflußinformation über das zu interpretierende Programm läuft. In jedem Berechnungsschritt der abstrakten Zustandsmaschine wird eine Anweisung interpretiert, die wir in Spezifikationen und im folgenden auch mit **Kommando** bezeichnen. Den Kommandos sind Universen (Sorten) der ASM zugeordnet und für jedes Kommando-Universum gibt es eine Regel, die den Zustandsübergang bei der Interpretation beschreibt. Jedes Kommando hat einen Wert und zwischen Kommandos gibt es Datenabhängigkeiten. Die Steuerfluß- und Datenabhängigkeiten sind auch die Grundlage des Kompositionsmechanismus von Montages. Der Speicher ist durch eine Funktion modelliert. Dieses Modell für Spezifikationen legt eine spezielle Klasse von ASMs fest, die wir **S**tandard **ASM**s (SASM) nennen und in Abschnitt 5.1 exakt definieren. Alle Spezifikationen in unserem Übersetzungsrahmen sind aus dieser Klasse.

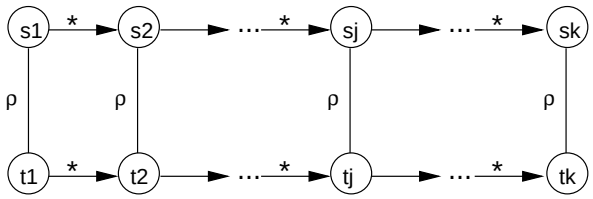
AL gibt nun Kommando-Universen für genau die Operationen und Anweisungen von Zielmaschinen vor und benutzt gleichzeitig das abstrakte Laufzeitmodell aus SASM. Die Semantik einer Sprache wird definiert, indem eine Abbildung der abstrakten Syntax auf die Syntax von *AL* angegeben wird[1]. *AL*-Terme werden durch Graphen syntaktisch repräsentiert. Die Knoten beschreiben Operationen der Sprache und die Kanten stellen Beziehungen zwischen Operationen dar. Eine Semantik bekommt *AL* durch die ASM-Spezifikation, die wir in Anhang B angegeben haben. Daher können wir über Ausführungen von *AL*-Programmen genau wie über Ausführungen beliebiger abstrakter Zustandsmaschinen argumentieren und die gleichen Techniken zum Nachweis von Eigenschaften anwenden. Semantische Äquivalenzen über *AL*-Termen werden im semantischen Kalkül von ASMs nachgewiesen. *AL* zusammen mit der ASM-Semantik definiert einen Semantikkalkül. In unserem Ansatz nutzen wir auch aus, daß *AL* selbst eine spezielle Zwischen- bzw. Zielsprache darstellt.

1) Eine Spezifikation in einem beliebigen Formalismus definiert eine solche Abbildung in das semantische Modell, das dem Formalismus zugrundeliegt. Für ASMs ist das semantische Modell die Algebra.

3.3 Modulare Übersetzungsverifikation

In diesem Abschnitt erklären wir, wie der Korrektheitsbeweis für die Übersetzung einer Sprache über die vertikale Dekomposition hinaus auch horizontal zerlegt werden kann. Wir nutzen dazu die kompositionellen Eigenschaften der Semantik. Im Prinzip betrachten wir die einzelnen Teile aus denen eine Programmiersprache aufgebaut ist als Minisprachen, verifizieren die Übersetzung dieser Minisprachen und weisen nach, daß die Komposition der übersetzten Teile korrekt ist. Die Korrektheit einer Übersetzung wurde in Abschnitt 2.3.2 mit Simulation des beobachtbaren Verhaltens des Quellprogramms durch das Zielprogramm definiert. Zum Nachweis der Korrektheit müssen wir die Zustandsfolgen bei der Ausführung eines Programms für eine bestimmte Eingabefolge (Abb. 3.2 (a)) betrachten. Mit unserer Zerlegung haben wir lokale Simulationsbeziehungen für Teilsprachen nachzuweisen. In Abb. 3.2 (b) beschreiben S, S', T, T' Teilsprachen, die sich über ρ' und ρ'' simulieren. Teilsprachen können selbst auch wieder aus Teilsprachen aufgebaut sein. Die globale Korrektheit ergibt sich durch strukturelle Induktion über den Aufbau der Sprache bzw. den Aufbau von Sprachkonzepten. Die sequentielle Komposition von Teilfolgen ist ein Spezialfall dieser strukturellen Induktion. Die

(a) Zustandsfolgen



(b) Zerlegung in Teilsprachen

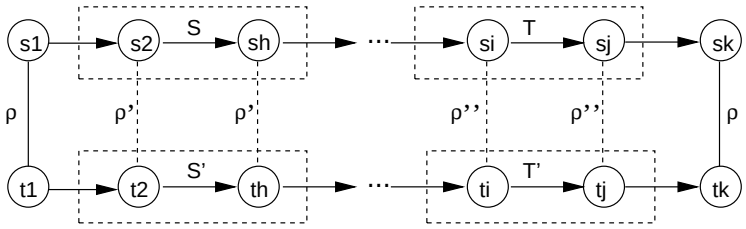


Abbildung 3.2: Unterschiedliche Ebenen für Simulationsbeweise

Zerlegung impliziert eine Verfeinerung unseres Korrektheitsbegriffes. Durch die Zerlegung in Teilzustandsfolgen werden die Anfangs- und Endzustände der einzelnen Teile beobachtbar, da über sie die Komposition definiert ist. Diese Verfeinerung invalidiert nicht die ursprüngliche Korrektheitsaussage, weil bisher beobachtbare Zustände auch weiterhin beobachtbar bleiben. Allerdings kann diese Verfeinerung die Menge der ursprünglich korrekten Transformationen einschränken, da nun mehr Zustände beobachtbar sind, die in Beziehung stehen müssen.

Die Semantik einer Programmiersprache ist induktiv über der abstrakten Syntax aufgebaut. Eine Montages-Spezifikation der Semantik definiert für jede Produktion der abstrakten Syntax eine eigene Montage. Eine Montage oder eine Produktion definiert eine Minisprache in Abhängigkeit der Minisprachen, die durch Nichtterminale auf der rechten Seite der Produktion beschrieben sind. Komponiert werden diese Minisprachen über Steuerfluß und Datenabhängigkeiten. Wir nennen diese Minisprachen **Sprachkonzepte**, denn das ist es, was sie eigentlich beschreiben. Eine Schleife, eine Zuweisung, aber auch ein Bezeichner sind Sprachkonzepte einer

Programmiersprache, denen eine abstrakte Syntax, eine statische und eine dynamische Semantik gegeben wird. Die Montage in Abb. 4.3 definiert das Sprachkonzept Zuweisung (Assign), das unter Verwendung der Sprachkonzepte Ausdruck (Expr) und Designator (Des) spezifiziert ist. Sie werden sequentiell berechnet und ihr Resultat wird von dem Zuweisungs-Kommando (assign) benutzt, um den Wert eines Objekts zu verändern. Die statische Semantik von Assign sichert zu, daß es ein solches Objekt gibt und daß der Wert, der bei der Ausführung von Expr berechnet wird, auch an dieses Objekt zugewiesen werden kann. Da wir nur Programme mit korrekter statischer Semantik betrachten und Informationen aus dem Kontext laut Voraussetzung vorliegen, können wir sicheren Gebrauch von statischen Informationen machen. Zum Beispiel können wir auch sicher sein, daß zu einem Bezeichner ein Verweis auf eine entsprechende Definition existiert.

Eigentlich wollen wir für Korrektheitsbeweise, wie bei der konkreten Syntax, ganz allgemein über den Aufbau des AST argumentieren. Die Montages-Spezifikation induziert zwar eine Zerlegung in Minisprachen, aber für die korrekte Abbildung des Sprachkonzepts Program muß immer noch über die komplette Sprache argumentiert werden, weil diese Minisprache in Abhängigkeit der Minisprachen ihrer Teilbäume definiert ist. Wir haben also nur eine Zerlegung gewonnen, nicht aber die Komplexität der Beweise reduziert.

Die Untersuchung von Sprachkonzepten zeigt jedoch, daß die Semantik eines Konzepts sehr wohl definiert werden kann, ohne die Semantik der Teile, aus denen es aufgebaut ist, explizit zu kennen. Wir nutzen das aus und definieren die dynamische Semantik eines Sprachkonzepts generisch, ohne explizite Kenntnis der Semantik der Teilkonzepte. Häufig werden wir allerdings Minimalannahmen für Teilkonzepte machen, die wir dann auch durch ASMs beschreiben. Unter Verwendung dieser Minimalannahmen hat ein Sprachkonzept eine lokale Semantik über die wir argumentieren können, ohne die konkrete Semantik der Teilkonzepte zu kennen. Die Instantiierung eines so definierten generischen Sprachkonzepts ist genau dann korrekt, wenn die konkret eingesetzten Teilkonzepte die Minimalanforderungen erfüllen, d.h. wenn es eine Simulationsbeziehung zwischen den konkreten ASM-Semantiken der Teile und den ASMs gibt, die die Minimalanforderungen definieren.

Beispiel 3.1 Eine Schleife besteht aus einer Schleifenbedingung Expr und einer Liste von Anweisungen, dem Schleifenrumpf Stats. Die Ausführung der Schleife berechnet zuerst den Wert der Schleifenbedingung. Ist dieser Wert gleich *true*, dann wird der Schleifenrumpf abgearbeitet. Anschließend wird die Schleife wieder ausgeführt. Liefert der Wert der Schleifenbedingung *false*, dann ist die Abarbeitung der Schleife beendet. Die komplette operationelle Semantik der Schleife hängt natürlich von dem konkreten Ausdruck (Expr) und dem konkreten Schleifenrumpf (Stats) ab. Zum Beispiel könnte die Berechnung des Ausdrucks einen Seiteneffekt haben oder der Rumpf enthält eine Break-Anweisung. Seiteneffekte der Ausdrucksberechnung haben auf die lokale Semantik der Schleife keine Auswirkungen. Allerdings kann die Iteration durch eine Break-Anweisung abgebrochen werden und damit hat die Abarbeitung des Schleifenrumpfs sehr wohl Auswirkungen auf die Schleife. Für die Semantik der Schleife müssen also bestimmte Minimaleigenschaften von Ausdrücken und Anweisungslisten berücksichtigt werden. Im einfachsten Fall würde man verlangen, daß die Schleifenbedingung eine zweiwertige Entscheidung berechnet und daß der Schleifenrumpf ausgeführt werden kann. Will man im Schleifenrumpf Break- und Continue-Anweisungen erlauben, dann berücksichtigt die Minimalanforderung für die Anweisungsliste, daß die Abarbeitung auf unterschiedliche Art und Weise (nach der letzten Anweisung oder über Break und Continue) beendet werden kann. Solange Expr bzw. Stats diese Minimalanforderungen erfüllen, können beliebige konkrete Ausdrücke bzw. Anweisungslisten eingesetzt werden.

Für die komplette Programmiersprache ergeben sich aus der Summe der Verwendungen von Ausdrücken Anforderungen an die Definition der Ausdrücke. Nur wenn für alle Sprachkonzepte diese Anforderungen widerspruchsfrei sind und von der tatsächlichen Definition erfüllt werden, ist die Semantik der Sprache widerspruchsfrei definiert. ♢

Die globale Korrektheit einer Übersetzung ist durch die korrekte Abbildung des Sprachmusters Program gegeben und wird durch strukturelle Induktion über den Aufbau von Programmen nachgewiesen. Das impliziert, daß für alle durch Montages definierten Sprachkonzepte die korrekte Übersetzung nachgewiesen werden muß.

3.4 Benutzung der Sprache *AL* zur Verifikation von Übersetzern

Die Spezifikationssprache *AL* hat mehrere Aufgaben in unserem Übersetzerrahmen.

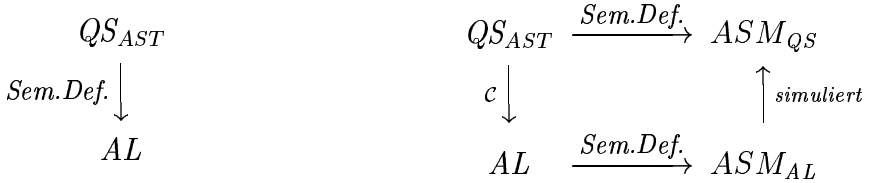
1. Wir definieren mit *AL* die Semantik von Sprachkonzepten in Form von abstrakten Zustandsmaschinen.
2. Wir verwenden *AL* als abstrakte Zwischensprache bei der Übersetzung, die den Übersetzungsprozeß aufspaltet und Übersetzerteile wiederverwendbar macht.

Aus diesen beiden Aufgaben leiten sich widersprüchliche Anforderungen an *AL* ab. Eine Spezifikationssprache sollte möglichst komfortabel und einfach verwendbar sein. Also sollte *AL* auch die üblichen Konzepte imperativer Sprachen unterstützen. Das bedeutet, daß *AL* imperativen Quellsprachen sehr ähnlich wäre. Damit ist *AL* jedoch keine Zwischensprache, die einfach in existierende Zielsprachen übersetzt werden kann. In diesem Fall müßten die Konzepte von *AL* nämlich eng mit den üblichen Zielsprachkonzepten verwandt sein.

Die Lösung dieses Dilemmas erreichen wir, indem wir *AL* mit den Konzepten von Zwischensprachen definieren und für die Spezifikation der Semantik eine Bibliothek vordefinierter imperativer Konzepte zur Verfügung stellen, für die wir korrekte Übersetzungen nach *AL* angeben. *AL* stellt die Basiskonzepte dieser Bibliothek von Semantikspezifikationen und korrekten Übersetzungen zur Verfügung, mit deren Hilfe wir neue Konzepte und korrekte Transformationen definieren können. Aus theoretischer Sicht können wir *AL* auch als die Grundlage eines erweiterbaren Kalküls für korrekte Übersetzungen ansehen. *AL* definiert die Syntax des Kalküls. Die Semantik ergibt sich durch die ASM-Semantik. Gerichtete Regeln des Kalküls entsprechen Übersetzungen. Der Kalkül ist genau dann korrekt, wenn für alle Regeln $r \equiv t \rightarrow t'$ gilt, daß die Semantik $[\![t]\!]$ des Terms t durch die Semantik $[\![t']\!]$ simuliert wird.

3.4.1 Spezifikation von Semantik mit *AL*

Die Semantik einer Quellsprache *QS* wird durch eine Abbildung $QS \rightarrow AL$ definiert, die wir auch mit $[\![ ]\!]$ bezeichnen. In diesem Fall gibt es zwei Szenarien, die in Abbildung 3.3 dargestellt sind. Entweder hat diese Abbildung definitorischen Charakter, weil sie die formale Spezifikation der Quellsprache darstellt, die gegenüber der informellen Spezifikation nur validiert werden kann, oder diese Abbildung stellt schon den ersten Übersetzungsschritt dar, der gegenüber einer formalen Semantik verifiziert werden muß. Für die Definition einer Quellsprache haben wir normalerweise Fall (a) vorliegen, während wir bei der Spezifikation einzelner Sprachkonzepte der Bibliothek Fall (b) antreffen.



(a) definitorische Semantik (b) zu verifizierende Definition der Semantik

Abbildung 3.3: Szenarien der Semantikspezifikation

Wir verwenden normalerweise zur Spezifikation der Semantik einer Quellsprache QS nicht AL selbst, sondern eine Sprache AL^+, die speziell auf QS zugeschnitten ist, indem sie die Konzepte von QS unterstützt. Die Sprache AL^+ wird unter Zuhilfenahme der Bibliothek von Sprachkonzepten zusammengebaut. Jede Sprache AL^+ beschreibt eine Teilmenge der Konzepte der Bibliothek und enthält speziell auch die semantischen Konzepte aus AL.

Die Sprachdefinition einer Quellsprache QS definiert eine Abbildung des AST_{QS} auf eine Sprache AL^+. Die Semantik einer konkreten Sprache AL^+ ist durch die Abbildung nach AL erklärt. So wird einerseits die Semantik von QS formal spezifiziert, andererseits stellt diese Abbildung auch eine Übersetzung dar. Abbildung 3.4 zeigt die Situation bei der Spezifikation. Geben wir eine Abbildung für ein Quellsprachkonzept an, das in dieser Form in der Bibliothek

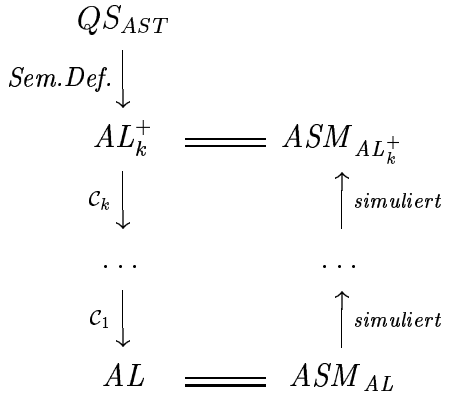


Abbildung 3.4: Spezifikation der Semantik mit AL^+

bisher nicht existiert, dann definiert das eine neue Übersetzungsregel. Zusätzlich können wir Übersetzungsregeln definieren, die AL^+-Konzepte in Terme über den Konzepten der Bibliothek transformieren. Diese Regeln müssen ebenfalls verifiziert werden. Die Korrektheitsbeweise sind Simulationsbeweise gegenüber der AL-Semantik der Quelle und des Ziels. Jede Definition einer korrekten neuen Regel erweitert den Übersetzungskalkül.

3.4.2 *AL* im Übersetzungsprozeß

Übersetzen wir eine Quellsprache in eine konkrete Zwischensprache, dann definieren wir Abbildungen zwischen zwei Sprachen AL_1^+ und AL_2^+. AL_1^+ beschreibt die Quellsprache, AL_2^+ beschreibt die Zwischensprache. Die Abbildung ist korrekt, wenn sie mit Regeln des Kalküls konstruiert werden kann. Werden zusätzliche Übersetzungsregeln benötigt, dann müssen diese ebenfalls verifiziert werden.

Bemerkung: Verwenden wir komplette Sprachen AL_i^+ als Zwischensprachen bei der Übersetzung beliebiger Quellsprachen wieder, dann definiert das eine Aufspaltung des Übersetzungs-

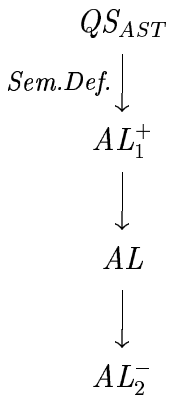


Abbildung 3.5: AL und die Bibliothek bei der Übersetzung

prozesses, der mit der Aufteilung eines Übersetzers in Front- und Backend vergleichbar ist. In diesem Fall müssen wir sicherstellen, daß für alle Sprachen aus AL^+ eine Übersetzung nach AL_i^+ existiert. Eine hinreichende Bedingung dafür ist, daß es eine Abbildung von AL nach AL_i^+ gibt. ♢

Für realistische Übersetzungen $QS \to AL \to ZS$ müssen wir annehmen, daß eine Ziel- bzw. Zwischensprache ZS Konzepte der Quellsprache hat, die nicht in AL vorkommen. Dann könnten wir zwar eine korrekte Übersetzung $QS \to ZS$ über den Weg $QS \to AL \to ZS$ konstruieren, aber wir könnten nicht die Konzepte von Quelle und Ziel für die Erzeugung effizienten Codes ausnutzen. Für konkrete Paare von Quell- und Zielsprache lösen wir dieses Problem, indem wir zusätzliche Übersetzungsregeln definieren und verifizieren, die genau die gemeinsamen Konzepte der Quelle und des Ziels ausnutzen. Diese Technik ist von der Codeerzeugung adaptiert. Bei der kostengesteuerten Termersetzung, wie sie z. B. zur Codierung von Ausdrucksbäumen eingesetzt wird, können zusätzliche Termersetzungsregeln angegeben werden, die Zielcode mit geringeren Kosten erzeugen. Abbildung 3.6 illustriert das Vorgehen. Anstatt über C_1 und C zu übersetzen, bilden wir die Sprache AL_1^+ direkt mit C' in die Zwischensprache ab.

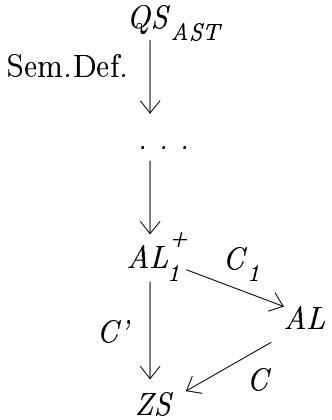


Abbildung 3.6: Optimierte Übersetzung einer Quellsprache mit AL

Betrachten wir den Kalkül aus Beweissicht, dann beschreiben die Regeln des Kalküls Schlußregeln auf der Semantik. Transformationen eines Programms und semantische Schlüsse entsprechen sich. Für eine Transformation $t \to t'$, die als korrekt bewiesen wurde, gilt auch der semantische Schluß $[\![t']\!]$ simuliert $[\![t]\!]$. Der Kalkül speichert korrekte semantische Schlüsse,

die wiederum zum Nachweis neuer Schlüsse ausgenutzt werden können. Will man eine neue Transformation verifizieren, dann können dazu die existierenden Kalkülregeln benutzt werden. Läßt sich die Transformation mit existierenden Transformationen beschreiben, dann müssen keine neuen Beweisschritte angegeben werden. Läßt sich eine Transformation nicht direkt aus Transformationen der Bibliothek zusammenstecken, dann können zumindest Teilschritte übernommen werden, die vom Anwender (dem Übersetzerbauer oder auch dem Sprachdesigner) von Hand ergänzt werden müssen und der Anwender muß die Korrektheit der neuen Schritte nachweisen.

Das Ziel des Kalküls ist **nicht** die Herleitung einer automatischen Beweisstrategie zur Verifikation von Übersetzungen. Der Anwender muß immer noch die Beweisschritte bzw. die Transformationen auswählen, wird aber durch den Kalkül von dem Nachweis der Korrektheit dieser Beweisschritte entbunden.

3.5 Ausnutzung von Transformationsmustern

In Abbildung 3.4 haben wir den Prozeß der Spezifikation der Semantik und der Transformation illustriert. Jede Abbildung einer Sprache AL^+ nach AL muß verifiziert werden. Eine solche Abbildung ist aus Teilabbildungen von Sprachkonzepten komponiert. In Abschnitt 3.3 haben wir dargelegt, wie Sprachkonzepte bzw. korrekte Transformationen korrekt komponiert werden können. Bisher haben wir dem Anwender jedoch nur einen geeigneten Korrektheitsbegriff für Transformationen und eine allgemeine Simulationsbeweistechnik an die Hand gegeben. Die Verifikation einer konkreten neuen Transformation ist Aufgabe des Benutzers. Zu seiner Unterstützung haben wir bestimmte **Transformationsschemata** identifiziert und allgemein als korrekt bewiesen. Dabei nutzen wir keine konkreten Eigenschaften eines Sprachkonzepts aus, sondern argumentieren über Eigenschaften des ASM-Formalismus selbst. Die Korrektheit eines Transformationsschemas muß genau einmal nachgewiesen werden. Entspricht eine konkrete Transformation einem solchen Transformationsmuster, dann reicht es zur Verifikation der Transformation aus, zu zeigen, daß die Transformation tatsächlich eine Instanz des Transformationsschemas ist.

Der Nutzen von Transformationsmustern resultiert aus der Tatsache, daß alle Übersetzer bestimmte Klassen gleichartiger Transformationen durchführen. Das wird einsichtig, wenn man die Aufgaben der Übersetzung betrachtet. Bei der Abbildung einer Quellsprache werden komplexe Operationen durch einfachere Operationen der Zielmaschine implementiert, Berechnungssequenzen für den Zugriff in den Speicher werden in Spezialbefehlen der Zielmaschine zusammengefaßt, die Überladung von Operationen wird aufgelöst und Indeterminismus, z. B. bei der Ausdrucksauswertung, wird eliminiert.

Konkret haben wir die folgenden Transformationsschemata identifiziert und verifiziert:

- **Dekomposition** zur Zerlegung komplexer Konstrukte,
- **Komposition** zur Zusammenfassung von Operationssequenzen,
- **Elimination von Indeterminismus** zur Reduktion von Freiheiten und
- **Spezialisierung** zur Auflösung von Überladung.

In (Heberle und Löwe, 1998) ist die prinzipielle Idee der Definition von generischen Transformationen über ASMs beschrieben, die in (Merke, 1998) weiter ausgearbeitet wird. In Kapitel 8 spezifizieren und verifizieren wir diese Transformationsschemata.

3.6 Wiederverwendung von Sprachkonzepten und Transformationen

Die formale Spezifikation der dynamischen Semantik einer Programmiersprache und die verifizierte Transformation in eine Zwischensprache ist sehr aufwendig. Daher hat Wiederverwendung einen hohen Stellenwert. Verwenden wir AL als konkrete Zwischensprache in einem Übersetzer, dann können wir sowohl die Abbildung $QS \rightarrow AL$ für die Übersetzung einer Quellsprache QS in unterschiedliche Zwischensprachen ZS als auch einen Übersetzer $AL \rightarrow ZS$ zur Übersetzung unterschiedlicher Quellsprachen nach ZS wiederverwenden. Von größerem Nutzen ist die Verwendung einer Bibliothek von Semantikspezifikationen und korrekten Transformationen. In diesem Abschnitt geben wir die Voraussetzungen für Wiederverwendung an, stellen unsere Bibliothek vor und beschreiben unterschiedliche Möglichkeiten ihrer Verwendung zur Definition neuer Sprachkonzepte und Übersetzungen.

Der Ausgangspunkt für Wiederverwendung ist die Spezifikation der Semantik. Können wir semantische Konzepte zur Spezifikation wiederverwenden, dann ergibt sich die Wiederverwendung der Transformation dieser Konzepte und damit der korrekten Implementierung dieser Transformationen automatisch, da die semantischen Konzepte einer Sprache durch die abstrakte Syntax repräsentiert sind und Transformationen auf der abstrakten Syntax durchgeführt werden. Sehen wir die abstrakte Syntax als Repräsentation für Semantik an, dann definiert unsere Bibliothek eine sprachunabhängige abstrakte Syntax für imperative Sprachen.

Unser Vorgehen ist in Abb. 3.7 illustriert. Wir geben die Semantik einer Quellsprache als Abbildung auf eine Sprache AL^+ an, die sich aus Konzepten der Bibliothek zusammensetzt. Diese Abbildung ist kompositionell über den einzelnen semantischen Konzepten der Quellsprache definiert, die durch die Produktionen der abstrakten Syntax bestimmt sind. Innerhalb unserer Bibliothek ist sichergestellt, daß es für jedes Konzept aus AL^+ eine korrekte Abbildung in Terme von AL gibt. Unter der Voraussetzung, daß die Komposition der Konzepte aus AL^+ bestimmte Eigenschaften erfüllt, haben wir dann auch eine korrekte Übersetzung der Quellsprache nach AL. Da der Übersetzerbauer die abstrakte Syntax angibt, die im Über-

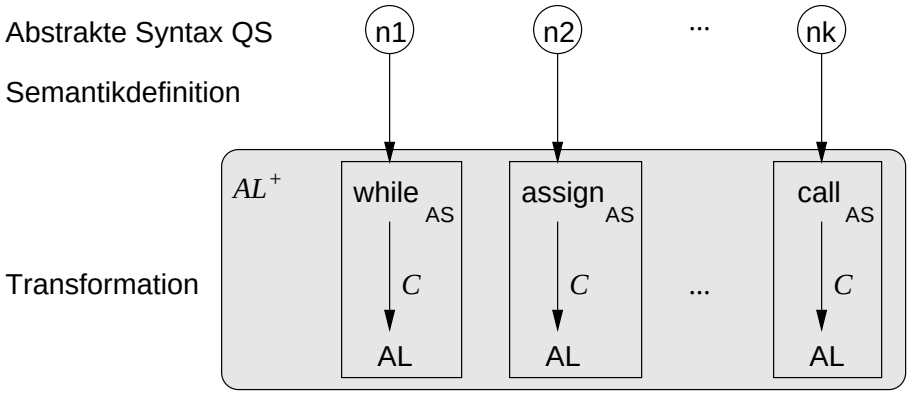


Abbildung 3.7: Bibliotheksunterstützte Spezifikation und Übersetzung einer Quellsprache

setzer verwendet wird, hat er auch die Freiheit eine sprachunabhängige abstrakte Syntax zu wählen. Unsere Konstruktion sorgt dafür, daß das aus Übersetzungssicht keine Nachteile hat, wir aber so einen korrekten Übersetzer konstruieren können. Definiert man eine Sprache aus vordefinierten Konzepten bzw. über einer sprachunabhängigen abstrakten Syntax, dann ist die entscheidende Frage, wie diese Konzepte komponiert werden bzw. wie die Konsistenz dieser

Komposition sichergestellt werden kann. Wir haben zwei Möglichkeiten die Korrektheit der Komposition abzusichern:

1. Die statische Semantik der Programmiersprache: sie muß auch in unserem Rahmen wohlgeformt sein. Das bedeutet, bei der Komposition darf es nicht zu widersprüchlichen Bedingungen kommen, die an Programme bzw. die Attributierung der Programme einer Sprache gestellt werden. Außerdem müssen die statischen Informationen, die jedes einzelne Sprachkonzept benötigt, eindeutig definiert sein.

2. Die Minimalanforderungen für die Teilkonzepte eines Sprachkonzepts: sie sichern zu, daß die Teile, über denen ein Sprachkonzept aufgebaut ist, auch dort eingesetzt werden dürfen. Der Sprachentwerfer bzw. der Übersetzerbauer, der die Semantikdefinition (die Abbildung nach AL^+) angibt, muß beweisen, daß die Anforderungen für jede Verwendung erfüllt sind. Wird ein Sprachkonzept innerhalb mehrerer Sprachkonzepte verwendet, wie das z. B. bei Ausdrücken der Fall ist, dann ergibt die Menge der Verwendungen eine Menge von Minimalanforderungen an dieses Sprachkonzept. Eine falsche Verwendung drückt sich in Widersprüchen aus.

Für das Zusammenstecken ist die Wahl der Abstraktionsebene, auf der wir Semantik betrachten wichtig. Wir betrachten Semantik auf der Ebene von Produktionen der abstrakten Syntax. Das bedeutet, wir reden auf dieser Ebene z. B. nicht von einem Konzept Polymorphie, sondern betrachten die Sprachkonzepte polymorphen Aufruf und die Definition einer Klasse als Untertyp einer anderen Klasse. Auf dieser Ebene können wir auch Konflikte erkennen, wenn Sprachkonzepte nicht orthogonal sind.

Das komfortable Zusammenstecken einer Sprache hängt stark von den Konzepten ab, die durch die Bibliothek unterstützt werden. Da unser erstes Ziel die Übersetzung imperativer Sprachen ist, haben wir folgende imperative Sprachkonzepte definiert und ihre Übersetzung verifiziert:

- Basisdatentypen werden auf spezielle parametrisierte Datentypen der Bibliothek abgebildet und können symbolisch übersetzt werden (Heberle und Heuzeroth, 1997).
- Strukturierte Datentypen mit Werte- und Referenzsemantik werden auf Basisdatentypen mit entsprechenden Zugriffs- und Kopierfunktionen abgebildet.
- Dynamische Datenstrukturen erfordern ein Konzept Halde (engl.: *Heap*), auf dem Objekte mit unbekannter Lebenszeit abgelegt und verwaltet werden.
- Unterschiedliche Parameterübergabemechanismen für Prozeduren und Funktionen, wie Werte-, Referenz-, In- und In/Out-Parameter, werden mit entsprechenden Zugriffsfunktionen realisiert.
- Komplexe Steuerflußkonstrukte, wie Schleifen, bedingte und Case-Anweisung werden in Strukturen mit Sprüngen übersetzt.
- Überladene Anweisungen wie z. B. die Zuweisung werden durch speziellere Zwischensprachkonzepte implementiert.
- Indeterminismus wird mit Hilfe eines speziellen AL-Konzepts in Zwischensprachprogrammen explizit gemacht. In unserer Fallstudie betrifft das vor allem die Auswertungsreihenfolge bei Ausdrücken.

Die Arbeit von Heuzeroth (1998) beschreibt imperative Konzepte, die bei der Übersetzung einer realistischen imperativen Programmiersprache als Standard identifiziert wurden.

Für die Flexibilität des Ansatzes ist die Art und Weise wichtig, wie wir Sprachkonzepte wiederverwenden. Im einfachsten Fall finden wir exakt das Sprachkonzept der Quellsprache in der Bibliothek und sind fertig. Dieser Fall dürfte, zumindest solange die Bibliothek noch im Aufbau ist, selten eintreten, da Programmiersprachen meist unterschiedliche Nuancen von Sprachkonzepten definieren. Häufiger ist der Fall, daß sich ein Konzept aus Konzepten zusammenfügen läßt, die schon in der Bibliothek vorhanden sind. Dieses Prinzip funktioniert immer, weil die Sprache *AL* den Kern der Bibliothek darstellt und so gewählt wurde, daß sich alle imperativen Sprachkonzepte beschreiben lassen. Allerdings ist dieser Weg nicht sonderlich komfortabel.

Flexible Wiederverwendung erreichen wir durch die Generizität von Sprachkonzepten, die wir auch schon für die Verifikation ausgenutzt haben. Sind generische Sprachkonzepte nämlich unter Wiederverwendungsgesichtspunkten entworfen, dann abstrahieren sie von den Nuancen unterschiedlicher Programmiersprachen. Ein konkretes Programmiersprachkonzept wird dann durch eine entsprechende Instantiierung angegeben. Ein Beispiel dafür wäre eine Schleife, die sowohl für boolesche Ausdrücke aus C, als auch PASCAL verwendbar ist, weil sie in der Schleifenbedingung und dem Schleifenrumpf generisch definiert ist. Allgemein können wir generische Sprachkonzepte beliebig instantiieren, solange die statische Semantik widerspruchsfrei bleibt und die Minimalanforderungen aus Sicht der dynamischen Semantik erfüllt sind. Einen Spezialfall generischer Sprachkonzepte haben wir vorliegen, wenn die Parameter keine Teilkonzepte darstellen, sondern Schrauben zur Feindefinition eines bestimmten Sprachkonzepts sind. Ein Beispiel dafür ist der Integer-Datentyp, den wir mit den Parametern *MinInt, MaxInt* und *Arithmetik* in unsere Bibliothek aufgenommen haben. Die Sprachspezifikation legt dann nur noch die Parameter fest, die Wertebereiche der Parameter sind vorgegeben und können sogar statisch geprüft werden. Die Transformation ist in Abhängigkeit der Parameter definiert, und die Verifikation kann symbolisch durchgeführt werden.

3.7 Implementierung

Als letzte Aufgabe des Konstruktionsprozesses bleibt die korrekte Implementierung der verifizierten Transformationsspezifikation in einer höheren Implementierungssprache. Wir geben ein Implementierungsschema für die Bibliothek von Semantik- und Transformationsspezifikationen an und zeigen, wie sich damit Vor- und Nachbedingungen für die Implementierung ergeben. Die eigentliche Implementierungsverifikation stellt dann vor allem eine Fleißaufgabe dar, auf die wir in dieser Arbeit nicht weiter eingehen.

Wir haben uns für die objektorientierte Implementierungssprache SATHER-K (GOOS, 1997) entschieden, da sie Generizität mit Typschranken als Sprachkonzept anbietet. Damit lassen sich unsere generisch spezifizierten Sprachkonstrukte direkt in eine Implementierung umsetzen. Außerdem gibt es schon Arbeiten über das korrekte Zusammenstecken objektorientierter Komponenten (MEYER, 1992; FRICK ET AL., 1997), die wir für die Verifikation ausnutzen können[2].

Die Implementierung der Sprachdefinitionen und des Übersetzungskalküls nutzt konsequent die Konzepte der objektorientierten Implementierungsprache:

- Generische Klassen implementieren generische Sprachkonzepte und Typschranken definieren minimale Eigenschaften und Beweisverpflichtungen für korrekte Instantiierungen.

2) Prinzipiell hätten wir auch eine Implementierungssprache wählen können, die keine beschränkte Generizität kennt. Dann wäre die Umsetzung in eine Implementierung jedoch aufwendiger.

- Die syntaktischen Einheiten des Kalküls werden in Klassen umgesetzt. Befehls- bzw. Instruktionstypen definieren Klassen, Untertypsbeziehungen spiegeln Simulationsbeziehungen wieder.
- Daten- und Steuerflußinformation wird in Attributen gespeichert.
- Transformationsregeln entsprechen Methoden, die eine Übersetzung implementieren.
- Statische Informationen, die wir für die Übersetzung benötigen, implementieren wir durch eine Symbol- und eine Definitionstabelle.

Das Schema zur Umsetzung wird in (Heberle et al., 1998a) beschrieben und in (Heberle et al., 1998b) ausgearbeitet.

Zur Ableitung von Korrektheitsanforderungen für die Implementierung interpretieren wir die Ersetzungsregeln der Transformationsspezifikation als Vor- und Nachbedingungen. Die linke Seite einer Ersetzungsregel definiert die Vorbedingung, die rechte Seite legt die Nachbedingung fest.

Anstatt die Implementierung der Transformation selbst zu verifizieren, können wir verifizierte Überprüfer der Ergebnisse einer Transformation einsetzen (siehe Abschnitt 2.3.3). Das Beispiel Speicherabbildung zeigt die Vorteile. Das Ergebnis der Speicherabbildung ist einfach zu überprüfen, Adressen müssen eindeutig sein, dürfen sich nicht überlappen und müssen Typgrößen einhalten. Die Implementierung eines Überprüfers ist daher wesentlich einfacher zu verifizieren als ein optimierter Algorithmus, der eine möglichst kompakte Speicherbelegung berechnet. In der Transformationsphase ist die Speicherabbildung bisher die einzige Anwendung für Programmprüfung.

Die Wiederverwendung von Implementierungen ergibt sich durch die Wiederverwendung von Semantik- und Transformationsspezifikationen. Zu jeder Transformationsspezifikation gehört eine korrekte Implementierung in einer höheren Programmiersprache. Die Spezifikation beschreibt die kombinierte Anwendung von Transformationen und damit auch, wie die entsprechenden Implementierungen zusammengesteckt werden.

3.8 Zusammenfassung

Wir haben Verifikationsanforderungen in den traditionellen Prozeß zur Konstruktion von Übersetzern integriert und sind damit in der Lage, eine korrekte Transformationsphase für realistische Übersetzer zu konstruieren.

In diesem Kapitel haben wir unseren konkreten Ansatz im Überblick beschrieben. Wir haben die Verifikationsaufgabe strukturiert und damit die Komplexität von Beweisen verringert. Die Wiederverwendung existierender, verifizierter Spezifikationen für die Semantik und Transformationen von Sprachkonzepten, zusammen mit den Implementierungen der Transformationen, senkt die Kosten und reduziert die Möglichkeiten für Fehler bei der Konstruktion von Übersetzern für unterschiedliche Quell- und Zielsprachen. In den folgenden Kapiteln werden die hier skizzierten Lösungsschritte detailliert ausgearbeitet. Die Abbildung 3.8 zeigt die Zuordnung der einzelnen Kapitel in unsere Architektur zur Konstruktion eines verifizierten Übersetzers.

Unser Ansatz erfüllt alle im vorigen Kapitel aufgestellten Anforderungen. Der Übersetzer hat die gleiche Struktur, wie unverifizierte Übersetzer und die Verifikation ist in den Konstruktionsprozeß integriert. Die Spezifikationen der Semantik sind operationell, sowohl Quell- als auch

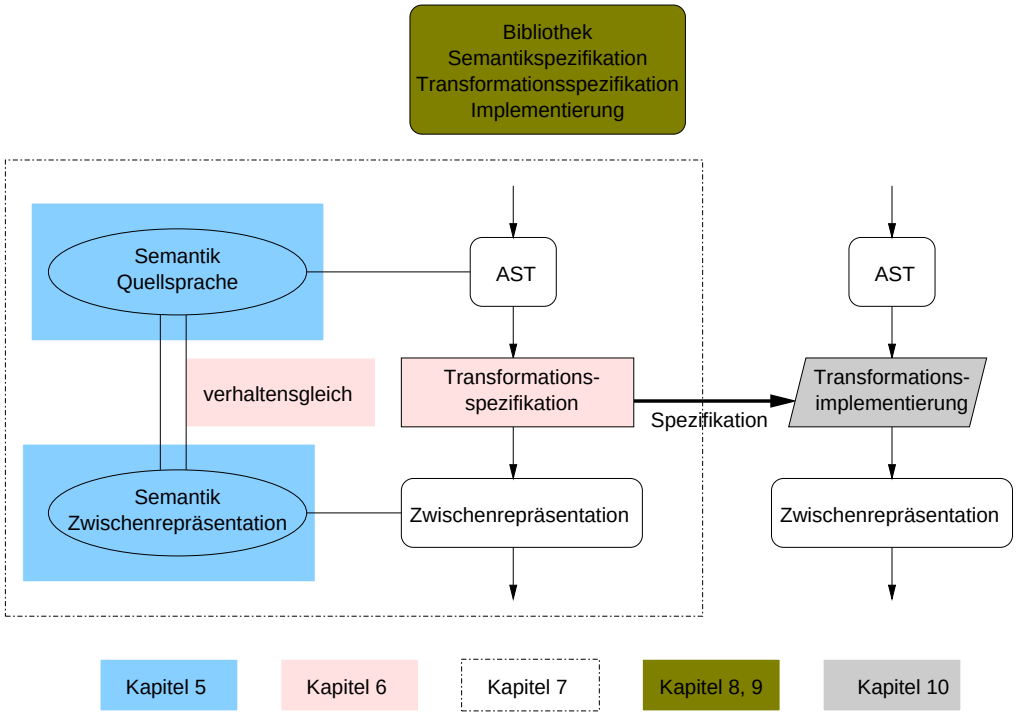


Abbildung 3.8: Zuordnung von Kapiteln und Konstruktionsprozeß

Zwischensprachen können im gleichen Formalismus beschrieben und alle imperativen Spracheigenschaften können einfach spezifiziert werden. Die Komplexität der Übersetzungsverifikation kann durch die Einführung zusätzlicher Zwischensprachen reduziert werden. Übersetzungen können für einzelne Sprachkonzepte verifiziert werden, weil unsere Semantikspezifikation mit generischen ASMs und Minimalanforderungen für Teilkonzepte eine abgeschlossene Semantik eines Sprachkonzepts definiert. Außerdem haben wir auf der Basis der Spezifikationssprache *AL* ein Konzept für die Wiederverwendung von Sprachspezifikationen und korrekt implementierten Transformationen angegeben. Sogar die Verifikation von Optimierungen, speziell Codeumordnungen ist möglich, weil die Korrektheit einer Übersetzung über beobachtbarem Verhalten definiert ist. Da unser Rahmen nicht auf eine feste Zwischensprache festgelegt ist, können Zwischensprachen entsprechend dem gewünschten Codeerzeugungsverfahren gewählt werden. Damit haben wir alle Anforderungen zur Erzeugung effizienten Maschinencodes erfüllt.

4 Abstrakte Zustandsmaschinen und Montages

Wir haben in Kapitel 2 den Formalismus der abstrakten Zustandsmaschinen informell vorgestellt. Für diese Arbeit benötigen wir jedoch eine detaillierte Definition, weil wir formal über die Semantik von Programmiersprachen und über die Korrektheit von Transformationen argumentieren wollen. Wir definieren einen Dialekt von Montages auf der Basis attributierter Grammatiken. Zusätzlich beschreiben wir die graphische Notation von Montages, die wir entsprechend der Originaldefinition verwenden, deren Semantik wir aber für unsere Zwecke anpassen.

Wenn wir im folgenden von Montages sprechen, dann ist unser Dialekt gemeint, es sei denn wir erwähnen explizit die Originaldefinition. Wenn wir von ASMs reden, dann müssen wir zwei Ebenen unterscheiden. Einerseits ist ASM der Formalismus selbst, also die Sprache zur formalen Beschreibung von Zustand und Zustandsübergängen. Andererseits reden wir auch von einer ASM, wenn wir eine konkrete Spezifikation meinen. Welche Ebene jeweils gemeint ist, wird aus dem Kontext jedoch sofort klar.

4.1 Die Definition von ASMs

Die folgenden Definition abstrakter Zustandsmaschinen ist aus (Gurevich, 1997) abgeleitet. Eine abstrakte Zustandsmaschine ist ein Tupel $\mathcal{A} = (\Sigma, Alg(\Sigma), \rightarrow, I)$. Σ ist eine einsortige Signatur mit der Sorte $\mathcal{U}$, $Alg(\Sigma)$ die Menge der Σ-Algebren, den Zuständen der ASM, $\rightarrow \subseteq Alg(\Sigma) \times Alg(\Sigma)$ die Übergangsrelation und $I \subseteq Alg(\Sigma)$ die Menge der Initialzustände von $\mathcal{A}$.

Ein Zustand q der abstrakten Maschine $\mathcal{A}$ ist eine Algebra mit Trägermenge $\mathcal{U}$ zusammen mit Interpretationen der Funktionsnamen aus Σ. Ein r-stelliger Funktionsname f wird als eine Funktion von $\mathcal{U}^r \rightarrow \mathcal{U}$ interpretiert. Wir bezeichnen die Interpretation $[\![f]\!]_q$ von f im Zustand q als **Basisfunktion** bzw. **Basisrelation**. Ein nullstelliger Name aus Σ wird als Element von $\mathcal{U}$ interpretiert. Zusätzlich gibt es noch einige spezielle Interpretationen von Namen:

- *true, false, undef* entsprechen ihrer intuitiven Interpretation.
- Die Interpretation eines Prädikats $P : \mathcal{U}^r \rightarrow \{true, false\}$ ist eine Menge von r-Tupeln $\overline{a}$, so daß $P(\overline{a}) = true$ gilt. Ein einstelliges Prädikat definiert ein Universum.
- *Bool* ist das Universum $\{true, false\}$. Das Gleichheitszeichen wird als Identitätsrelation auf $\mathcal{U}$ interpretiert.

Zusätzlich existiert ein spezielles Universum *reserve*, das als Quelle für neue Elemente fungiert. In einem Zustand q enthält *reserve* alle Elemente so, daß

- jedes Basisprädikat mit Ausnahme der Gleichheit *false* liefert, wenn zumindest eines ihrer Elemente zu *reserve* gehört.
- jede Basisfunktion *undef* liefert, wenn zumindest eines ihrer Argumente zu *reserve* gehört.
- keine Basisfunktion als Ergebnis ein Element von *reserve* liefert.

Die Algebren, die Zustände einer ASM beschreiben, sind in (GUREVICH, 1997) einsortig über dem Superuniversum $\mathcal{U}$ definiert. Einstellige Prädikate beschreiben Universen. Für die Elemente x eines Universums P gilt $P(x) = true$. Setzen wir ein Universum P mit der Menge der Elemente gleich, für die $P(x) = true$ gilt und sehen die Teilmengenbeziehung als partielle Ordnung über den Universen einer ASM an, dann gibt es keinen prinzipiellen Unterschied zwischen Universen und Sorten mehr. Wir können Σ daher auch als mehrsortige Signatur interpretieren, indem wir die Informationen über Universen miteinbeziehen. Eine ASM ist dann ein Tupel $(\Sigma, Alg(\Sigma), \rightarrow, I)$, wobei Σ nun eine geordnete mehrsortige Signatur ist. In unseren Spezifikationen benutzen wir Universen wie Sorten und benennen die Sorten wie die entsprechenden Prädikate, die Universen definieren. Für eine Funktion $f : \mathcal{U} \rightarrow \mathcal{U}$ schreiben wir $f : A \rightarrow B$, wenn gilt: $A(x) = false \Rightarrow f(x) = undef$ und $A(x) = true \Rightarrow B(f(x)) = true$.

Bemerkung: $I \subseteq Alg(\Sigma)$ kann durch eine Menge von Gleichungen *Init* definiert werden, für die gilt $I = \{i \in Alg(\Sigma) \mid i \models Init\}$. Sehen wir $\langle \Sigma, Init \rangle$ als algebraische Spezifikation an, dann gilt $I = Mod(\langle \Sigma, Init \rangle)$. Eine ASM ist dann durch $\mathcal{A} = (\langle \Sigma, Init \rangle, \longrightarrow)$ beschrieben. In dieser Arbeit nutzen wir diese Sicht für die Definition von Transformationen und in Korrektheitsbeweisen aus, wo wir über Gleichungen argumentieren. ♢

Das Grundkonzept von ASMs sind **Aktualisierungen** der Interpretation von Funktionen (engl.: *Updates*). Änderungen der Interpretation entsprechen Zustandsübergängen und werden durch **Aktualisierungsregeln** beschrieben. Die einfachste Aktualisierungsregel hat die Form:

$$f(\overline{t}) := t,$$

wobei f ein Funktionsname aus Σ mit Stelligkeit n und $t, \overline{t} = t_1, \ldots, t_n$ ein Term bzw. ein Tupel von Termen sind. Eine solche Regel nennen wir auch **atomar**. Informell ändert diese Aktualisierung die Interpretation der Funktion f an der $\overline{t}$ entsprechenden Stelle in die Interpretation von t. Die Semantik der Aktualisierungregel ist formal über Aktualisierungsmengen definiert:

Definition 4.1 (Aktualisierungsmenge)
Die zu einer Regel $r \equiv f(\overline{t}) := t$ gehörende Aktualisierungsmenge $Up(r, q)$ im Zustand q ist folgendermaßen definiert:

$$Up(r, q) \stackrel{\text{def}}{=} \{(L, y)\}$$

wobei L eine Stelle(engl.: location*) $L = (f, [\![\overline{t}]\!]_q)$ bezeichnet und $y = [\![t]\!]_q$ die Interpretation von t im Zustand q ist.*

Die Semantik einer Menge von Aktualisierungsregeln $R = \{r_1, \ldots, r_n\}$ in einem Zustand q ist durch

$$Up(R, q) \stackrel{\text{def}}{=} \bigcup_{i=1}^{n} Up(r_i, q)$$

definiert.

Die Ausführung einer Aktualisierungsregel R im Zustand q bewirkt die durch $Up(R, q)$ beschriebene Änderung der Interpretation. Auf Basis dieser einfachen Aktualisierungsregel werden komplexere Regeln definiert.

Aktualisierungsregeln können in Blöcken zusammengefaßt werden.

do in-parallel
  R$_1$
  R$_2$
enddo

Die Aktualisierungen eines Blocks werden simultan ausgeführt. Die entsprechende Aktualisierungsmenge ist

$$Up(\textbf{do in-parallel} R_1\ R_2\ \textbf{enddo}, q) \hat{=}\ Up(R_1, q) \cup Up(R_2, q)$$

Eine solche Aktualisierungsmenge enthält üblicherweise mehr als ein Element. Dadurch kann es zu widersprüchlichen Aktualisierungsmengen kommen. Zum Beispiel ist für

$f := \mathit{false}$
$f := \mathit{true}$

nicht klar, welchen Wert f nach der Aktualisierung hat.

Definition 4.2 (Konsistenz einer Aktualisierungsmenge)
*Eine Aktualisierungsmenge $Up(R, q)$ heißt genau dann **konsistent**, wenn es für alle $L = (f, a)$ höchstens ein Element $(L, y) \in Up(R, q)$ gibt.*

Eine Zustandsänderung wird nur ausgeführt, wenn die Aktualisierungsmenge **konsistent** ist.

Aktualisierungen können durch Bedingungen gesteuert bzw. durch das Erfülltsein einer Bedingung ausgelöst werden. Zustandsänderungen, die von der gleichen Bedingung abhängen, werden in einer Regel zusammengefaßt.

if *Cond* **then** *Updates* **endif**,

wobei *Cond* $\in$ *Bool* und *Updates* ein Block von Aktualisierungsregeln ist. Eine Regel ist in einem Zustand q genau dann anwendbar, wenn $[\![\mathit{Cond}]\!]_q = \mathit{true}$. Sind mehrere Regeln in einem Zustand anwendbar, dann wird indeterministisch eine Regel ausgewählt.

Die Zustandsübergangsrelation $\rightarrow$ wird durch eine endliche Menge von Aktualisierungsregeln R definiert. $\stackrel{n}{\rightarrow}$ bezeichnet die Komposition von n Zustandsübergängen, wobei die Komposition von Relationen wie üblich durch $\rho_1 \circ \rho_2 = \{(u, w) \mid \exists v : (u, v) \in \rho_1 \wedge (v, w) \in \rho_2\}$ definiert ist. $\stackrel{*}{\rightarrow}$ bezeichnet die reflexive, transitive Hülle von $\rightarrow$. Ein Zustand $q \in Alg(\Sigma)$ ist genau dann erreichbar, wenn es einen Initialzustand $i \in I$ gibt, so daß $i \stackrel{*}{\rightarrow} q$. Die Menge $F = \{f \in Alg(\Sigma) : \forall q' \in Alg(\Sigma) : f \not\rightarrow q'\}$ beschreibt die Menge der **Finalzustände**. Eine ASM ist genau dann **deterministisch**, wenn zu jedem Zustand $q \in Alg(\Sigma)$ höchstens ein Zustand $q' \in Alg(\Sigma)$ existiert, so daß $q \rightarrow q'$.

In ASMs gibt es zwei Klassen von Funktionen:

- Die Interpretation einer **dynamischen** Funktion wird durch Zustandsübergangsregeln verändert. f ist eine dynamische Funktion, wenn eine Aktualisierung der Form $f(t_1, \ldots, t_n) := t_{n+1}$ irgendwo in einer Regel auftaucht.
- Die Interpretation einer **statischen** Funktion ändert sich niemals.

Zusätzlich Außerdem gibt es **externe** Funktionen, die die Interaktion mit der Umwelt erlauben. Sie müssen nicht exakt spezifiziert sein, aber es können bestimmte Eigenschaften angegeben werden, die in jedem Zustand erfüllt sind. Externe Funktionen können in unterschiedlichen Zuständen unterschiedliche Werte liefern. In einem Zustand liefert eine externe Funktion jedoch bei jeder Verwendung den gleichen Wert.

Der ASM-Formalismus definiert einige nützliche Erweiterungen des Grundmodells, die in Abbildung 4.1 zusammen mit ihrer informellen Bedeutung beschrieben sind. Die Abbildung

Erweiterung	Informelle Bedeutung
choose v: v **in** U R **endchoose**	Führe die Regel R für ein beliebiges Element $v \in U$ aus. Diese Regel implementiert die nichtdeterministische Auswahl.
do forall $v : P(v)$ R **enddo**	Führe die Regel R für alle Elemente aus dem Bildbereich von P parallel aus.
extend U **with** u R **endextend**	Nimm ein Element u aus dem Reserveuniversum, füge das Element zum Universum U hinzu und führe die Regel R aus. Vor der Ausführung von **extend** gilt $reserve(u) = true$ und $U(u) = false$. Anschließend gilt $reserve(u) = false$ und $U(u) = true$.

Abbildung 4.1: Erweiterungen des ASM-Modells

4.2 zeigt einige Abkürzungen, die wir verwenden, um die Lesbarkeit von Spezifikationen zu erhöhen. Außerdem können in ASM-Regeln Makros verwendet werden. Sie werden einfach

Regel	Abkürzung
if c_1 **then** R_1 **endif** **if** $\neg c_1$ **then** R_2 **endif**	**if** c_1 **then** R_1 **else** R_2 **endif**

if $c_1 \wedge c_2$ **then** R_1 **endif**
if $c_1 \wedge \neg c_2$ **then** R_2 **endif**
	if c_1 **then**
	if c_2 **then** R_1
	else R_2 **endif**
	endif

Abbildung 4.2: Abkürzungen von ASM-Regeln

textuell durch ihre Definition ersetzt. Ein Makro kann parametrisiert sein.

$$macro(p_1, \ldots, p_k) \hat{=} \; t(p_1, \ldots, p_k)$$

Dabei ist t ein Term mit Variablen $p_1, \ldots, p_k$.

Wir verwenden in unseren Spezifikationen die Datentypen *SET, LIST* und die natürlichen Zahlen $\mathbb{N}$ mit den üblichen Operationen. Dabei nehmen wir an, daß diese Datentypen durch entsprechende Termalgebren definiert sind, deren Trägermenge die entsprechende Sorte repräsentiert.

4.2 Die Vereinigung von ASMs

In dieser Arbeit verwenden wir einen speziellen Vereinigungsoperator über abstrakten Zustandsmaschinen, um die formale Semantik einer Sprache induktiv über den abstrakten Zustandsmaschinen aufzubauen, die Teilsemantiken definieren.

Zwei ASMs $\mathcal{A}_1$ und $\mathcal{A}_2$ können vereinigt werden, $\mathcal{A}_1 \uplus \mathcal{A}_2$, indem ihre Initialzustände kombiniert und ihre Aktualisierungsregeln vereinigt werden.

Definition 4.3 (Vereinigung von ASMs)
Die Vereinigung $\mathcal{A}_1 \uplus \mathcal{A}_2$ von ASMs $\mathcal{A}_1 = (\Sigma_1, Alg(\Sigma_1), R_1, I_1)$ und $\mathcal{A}_2 = (\Sigma_2, Alg(\Sigma_2), R_2, I_2)$ ergibt die ASM $\mathcal{A} = (\Sigma_1 \cup \Sigma_2, Alg(\Sigma_1 \cup \Sigma 1), R_1 \cup R_2, I_{1,2})$, wobei $I_{1,2} \mid_{\Sigma_1} = I_1$ und $I_{1,2} \mid_{\Sigma_2} = I_2$.

Definition 4.4 (Gültigkeit der Vereinigung von ASMs)
*Wir nennen die Vereinigung zweier ASMs $\mathcal{A}_1$ und $\mathcal{A}_2$ **gültig**, wenn es einen Initialzustand der neuen ASM $\mathcal{A}_1 \uplus \mathcal{A}_2$ gibt, in dem alle Funktionssymbole $f \in \Sigma_1 \cup \Sigma_2$ eine konsistente Interpretation haben.*

In unserem Anwendungsfall sind wir jedoch nicht ganz so restriktiv. Wir erlauben explizit, daß die Interpretation einer Funktion im Initialzustand der einen ASM undefiniert ist, während sie im Initialzustand der anderen ASM einen definierten Wert hat. Die konsistente Interpretation einer Funktion $f : s_1 \times \ldots \times s_k \to s$, die sowohl in Σ_1, als auch in Σ_2 definiert ist, bedeutet dann, daß für zwei Initialzustände i_1 aus $\mathcal{A}_1$ und i_2 aus $\mathcal{A}_2$ gilt:

$$\forall \overline{x} \in s_1 \times \ldots \times s_k : [\![f]\!]_{i_1}(\overline{x}) \neq undef \wedge [\![f]\!]_{i_2}(\overline{x}) \neq undef \Rightarrow [\![f]\!]_{i_1}(\overline{x}) = [\![f]\!]_{i_2}(\overline{x}).$$

Die Definition von Vereinigung und Gültigkeit sagt nichts über die Konsistenz der Regeln zur Definition der Zustandsübergangsrelation aus. Das ist schon durch den ASM-Formalismus abgedeckt, der besagt, daß inkonsistente Aktualisierungsmengen, verursacht durch widersprüchliche Aktualisierungen von Funktionen, überhaupt keine Aktualisierung bewirken. In diesem Fall bleibt die abstrakte Zustandsmaschine also einfach stehen.

4.3 Ein Dialekt von Montages

Die ursprüngliche Definition von Montages setzt auf der konkreten Syntax einer Programmiersprache auf und definiert die statische Semantik mit Hilfe von Prädikaten über konkreten Strukturbäumen. Natürlicherweise legt man beim Entwurf einer Sprache jedoch erst eine abstrakte Syntax fest und definiert darüber die statische und die dynamische Semantik. Anschließend definiert man die konkrete Syntax durch eine Abbildung der abstrakten Syntax auf Symbolfolgen. Daher verwenden wir eine modifizierte Version von Montages, die sowohl die statische, als auch die dynamische Semantik über abstrakten Strukturbäumen (AST) definiert und zusätzlich mehr Möglichkeiten zur Spezifikation statischer Semantik bietet.

Die statische Semantik wird durch eine attributierte Grammatik $AG = (G, A, R, C)$ mit der abstrakten Syntax $G = (N, T, P, Z)$ und einer Familie von Mengen $A = (A_X)_{X \in N \cup T}$, den Attributen definiert. $R = (R_p)_{p \in P}$ ist eine Menge von Attributierungsregeln, und $C = (C_p)_{p \in P}$ beschreibt die Menge der Bedingungen, die für die Attributierung statisch semantisch korrekter Programme erfüllt sein müssen. Durch die Verwendung von attributierten Grammatiken ist eine beliebige Auswertungsreihenfolge von Attributierungsregeln möglich. Das ist ein weiterer Unterschied zur Originaldefinition von Montages, wo die statische Semantik durch eine einmalige Links-Rechts-Tiefensuche über dem Strukturbaum berechnet wird. Diese Einschränkung verhindert unter Umständen die effiziente Berechnung der statischen Semantik. Erlaubt eine Programmiersprache z. B. die Benutzung eines Namens vor seiner Definition, dann kann nicht in einem einzigen Tiefensuchdurchlauf überprüft werden, ob die statische Semantik eines Programms korrekt ist. In der Originalversion von Montages werden Prädikate, die nicht gleich

berechnet werden können, aufgesammelt und erst nach dem kompletten Durchlauf aufgelöst. Alternativ wird die Überprüfung semantischer Bedingungen in die dynamische Semantik verlagert. Für die erste Möglichkeit gibt es keine effiziente Berechnungsstrategie, und die zweite Möglichkeit ist nicht erwünscht, weil statische, semantische Eigenschaften in der dynamischen Semantik nichts zu suchen haben.

Die grundlegende Idee zur Beschreibung dynamischer Semantik besteht darin, daß einem Knoten des AST entweder Kommandos (Tasks) zugeordnet werden, die über Steuer- und Datenflußinformation in Beziehung stehen, oder daß ein Knoten einen Teilbaum (eine Phrase) repräsentiert, dessen Semantik wiederum durch einen Teilgraphen über Kommandos repräsentiert wird. Unsere Montages-Spezifikation definiert eine Abbildung von kompletten abstrakten Strukturbäumen auf Kommando-Graphen. Aus der Attributierung des abstrakten Strukturbaums und den Zustandsübergangsregeln konstruieren wir eine ASM, die Programmgraphen interpretiert und damit die dynamische Semantik einer Sprache operationell spezifiziert. Die dynamische Semantik ist nur über Programmen definiert, die wohlgeformt sind im Sinne der statischen Semantik der Sprache. Programme mit statischen semantischen Fehlern haben keine dynamische Semantik.

Eine Montages-Spezifikation $M = (AG, \mathcal{A})$ definiert nun eine attributierte Grammatik $AG = (G, A, R, C)$ und einen Interpretierer $\mathcal{A} = (\Sigma, Alg(\Sigma), \rightarrow, I)$. Die Spezifikation enthält für jede Produktion aus G eine Montage, die aus vier Komponenten besteht:

1. der Produktion der abstrakten Syntax,
2. einer Menge von Attributierungsregeln (teilweise graphisch dargestellt), die die statische Semantik und eine Abbildung des abstrakten Strukturbaums auf einen Instruktionsgraphen beschreiben,
3. einer Menge von Bedingungen an die Attributierung und
4. einer Menge von ASM-Regeln, die die Interpretation von Instruktionen beschreiben.

Im folgenden beschreiben wir, wie aus einer Montages-Spezifikation zu einem konkreten Programm die Menge der Σ-Algebren abgeleitet werden kann, die den Initialzustand einer ASM $\mathcal{A}$ beschreibt. Diese Abbildung zusammen mit den ASM-Regeln der vierten Komponente definiert dann die Semantik der Sprache.

Der Ausgangspunkt für die Spezifikation ist der attributierte Strukturbaum eines Programms. Der Aufbau des Strukturbaums ist durch die abstrakte Syntax angegeben, die definiert, wie Programmterme aussehen. Die Attributierung eines Programmterms kann abhängig vom Kontext unterschiedlich sein. Daher müssen wir Terme abhängig von ihrem Auftreten im Strukturbaum unterscheiden können. Wir erreichen das, indem wir jedem Knoten des Baums noch eine eindeutige Kennzeichnung seines Auftretens zuordnen[1].

Die Nichtterminal- und Terminalsymbole $X \in N \cup T$ definieren Sorten. Die Struktur von G impliziert eine partielle Ordnung auf diesen Sorten und definiert Konstruktoren zum Aufbau der Terme einer Sorte.

- Eine Regel $X ::= Y \mid Z \in P$ führt Sorten Y und Z mit $\mathsf{Y} \leq \mathsf{X}$ und $\mathsf{Z} \leq \mathsf{X}$ ein. Da Sorten Mengen sind, wird die partielle Ordnung $\leq$ durch die Teilmengenbeziehung induziert.
- Eine Regel $X ::= Y_1 \ldots Y_k \in P$ führt einen Konstruktor $c_X : \mathrm{N}^* \times Y_1 \times \ldots \times Y_k \rightarrow X$ ein.

1) Dieses Vorgehen ähnelt dem Vorgehen von Poetzsch-Heffter (1997).

Die Sorte Node bildet die Vereinigung aller von $N \cup T$ induzierten Sorten und beschreibt die Knoten eines Strukturbaums.

Der Einfachheit halber codieren wir das Auftreten eines Terms mit einer Liste natürlicher Zahlen. Ein Term $\mathsf{x} = c_X(o_x, \mathsf{y}_1, \ldots \mathsf{y}_k)$ ist wohldefiniert, wenn

$$tail(o_{y_1}) = \ldots = tail(o_{y_1}) = o_x$$

Per Konstruktion können wir Wohldefiniertheit sicherstellen. Für die Terme y_i, die als Teilterme eines Terms $\mathsf{x} = c_X(o_x, \mathsf{y}_1, \ldots \mathsf{y}_k)$ auftreten gilt:

$$o_{y_1} = [1|o_x] \quad \ldots \quad o_{y_k} = [k|o_x]$$

wobei $o_x = []$, falls der Term $\mathsf{x} = c_Z(o_x, \mathsf{y}_1, \ldots, \mathsf{y}_k)$ ein komplettes Programm beschreibt.

Die Attribute definieren Funktionen aus Σ. Für jedes Attribut $a \in A_X$ mit Typ T existiert eine Funktion $a \in \Sigma, a : X \to T$. Eine Attributierungsregel für eine Produktion $X_0 ::= X_1 \ldots X_n$ hat die Form:

$$X_i.a := f(X_{i_1}.a_1, \ldots, X_{i_k}.a_k)$$

mit $0 \leq i, i_1, \ldots, i_k \leq n, a : T \in A_{X_i}$, $a_j : T_j \in A_{X_{i_j}}$ für $T_j, j = 1, \ldots, k$ und einer Funktion $f : T_1 \times \ldots \times T_k \to T$. Entsprechend der Definition von Attributgrammatiken, siehe (Waite und Goos, 1984), ist ein Attribut entweder synthetisiert oder ererbt. Attributierungsregeln definieren Gleichungen, die von $Alg(\Sigma)$ erfüllt sein müssen.

Für jede Produktion $p : X_0 ::= X_1 \ldots X_n$ gilt:

$$\text{Wenn } \mathsf{x}_0 = c_{X_0}(o_{x_0}\mathsf{x}_1, \ldots, \mathsf{x}_n), \text{ dann } a(\mathsf{x}_i) = f(a_{i_1}(\mathsf{x}_{i_1}), \ldots, a_{i_k}(\mathsf{x}_{i_k}))$$

für alle $X_i.a := f(X_{i_1}.a_1, \ldots, X_{i_k}.a_k)$

Eine Bedingung c sieht folgendermaßen aus:

$$f(X_{i_1}.a_1, \ldots, X_{i_k}.a_k) = g(X_{i_{k+1}}.a_{k+1}, X_{i_m}.a_m),$$

wobei $0 \leq i, i_1, \ldots, i_k \leq n$, $a_j : T_j \in A_{X_{i_j}}$, $1 \leq k < m$, $f : T_1 \times \cdots \times T_k \to T$, und $g : T_{k+1} \times \cdots \times T_m \to T$ für einen Typ T.

Jedes gültige Programm π muß diese Bedingungen erfüllen. Daher erfüllt die statische Algebra für jede Produktion $X_0 ::= X_1 \cdots X_n$:

$$\text{Wenn } \mathsf{x}_0 = c_{X_0}(o_{x_0}, \mathsf{x}_1, \ldots, \mathsf{x}_n), \text{ dann } f(a_1(\mathsf{x}_{i_1}), \ldots, a_k(\mathsf{x}_{i_k})) = g(a_{k+1}(\mathsf{x}_{i_{k+1}}), \ldots, a_m(\mathsf{x}_{i_m}))$$

Die ASM $\mathcal{A}$ hat eine spezielle Sorte Task, die die Instruktionenmenge der Sprache repräsentiert. Montages führen Untersorten von Task ein. stop ist eine spezielle Instruktion, die das Ende der Programmausführung signalisiert. Montages definieren eine Erweiterung des abstrakten Strukturbaums zu einem Instruktionsgraphen. Die zweite Komponente einer Montage spezifiziert diese Erweiterung. Aus Sicht der abstrakten Syntax werden die einzelnen Produktionen um die Kommandos erweitert, über denen der entsprechende Instruktionsgraph aufgebaut ist. Kommandos sind neue Terminalsymbole.

Sei $X ::= Y_1 \ldots Y_k$ eine Regel der abstrakten Syntax der Sprache. Der entsprechende Instruktionsgraph ist über Kommandos $\mathsf{T}_1 \ldots \mathsf{T}_j$, $\mathsf{T}_i < \mathsf{Task} < \mathsf{Node}$ aufgebaut. Die Sprachspezifikation

definiert für jede Regel der Syntax den entsprechenden Kommandographen in Abhängigkeit der Nichtterminale Y_i. Für die Kommandos gelten dabei die gleichen Annahmen wie für Knoten des Strukturbaums. Daher stellen sie im Prinzip eine Erweiterung der abstrakten Syntax dar. Aus einer Regel $X ::= Y_1 \dots Y_k$ wird dann die neue Regel $X ::= Y_1 \dots Y_k\ T_1 \dots T_j$, wobei die T_i Terminale des entsprechenden Kommandographen sind. Sei c_X der entsprechende Konstruktor, dann gilt für einen Term $\mathsf{x} = c_X(o, \mathsf{y}_1, \dots, \mathsf{y}_k, c_{t_1}(o_{t_1}), \dots, c_{t_j}(o_{t_j}))$:

$$o_{y_1} = [1|o_x]\ \dots\ o_{y_k} = [k|o_x]\ o_{t_1} = [k+1|o_x]\ \dots\ o_{y_k} = [k+j|o_x]$$

Zusätzlich hat $\mathcal{A}$ Sorten, die Basisdatentypen (deren Semantik algebraisch spezifiziert ist) zusammen mit ihren Operationen definieren. Weitere Sorten definieren Typen (Type) und einen abstrakten Speicher mit einer Menge von Referenzen (Reference).

Eine Montages-Spezifikation definiert immer die folgenden Attribute:

- *initial* : 2^{Task} repräsentiert die Menge der Kommandos, die initial für eine Phrase ausgeführt werden können.
- *terminal* : Task repräsentiert das Kommando, das in einer Phrase als letztes ausgeführt wird.
- *nexttask* : Task $\times$ N $\rightarrow$ Task beschreibt die Nachfolger eines Kommandos im Steuerfluß. Der zweite Parameter ist notwendig, weil es mehrere Nachfolger geben kann.
- *data* : Task $\times$ N $\rightarrow$ Task beschreibt Datenabhängigkeiten zwischen Kommandos, die aus Operationen über Daten resultieren.

Die Funktionen *initial*, *terminal*, *nexttask* und *data* sind über Kommandoknoten definiert und werden in der semantischen Analyse berechnet. *initial*- und *terminal* werden benutzt, um Phrasen in ihren Kontext im Instruktionsgraphen einzubetten.

Neben den statischen Funktionen von Σ definiert die ASM $\mathcal{A}$ Funktionen, die für die Interpretation und damit für die Definition der dynamischen Semantik einer Programmiersprache notwendig sind.

- *ct* : Task ist ein abstrakter Programmzähler, der das aktuell auszuführende Kommando beschreibt.
- *value* : Task $\rightarrow$ Value wird benutzt, um Werte von Kommandos zu modellieren bzw. wir benutzen *value* : Task $\times$ N $\rightarrow$ Value, wenn es während der Interpretation mehrere Instanzen einer Funktion geben kann. Value ist die Vereinigung über den Werten der Basistypen.
- *content* : Reference $\rightarrow$ Value modelliert den Speicher.

Zu jedem Nichtterminal der abstrakten Syntax existiert eine Montage, welche in ihrer vierten Komponente Zustandsübergangsregeln definiert. Diese Regeln beschreiben die Interpretation von Kommandos, die durch diese Montage eingeführt werden. Die Übergangsregeln haben die Form:

if $X(ct) \wedge cond$ **then**
    Updates
endif

Sehen wir die Zustände einer ASM als mehrsortige Algebren an, dann würden wir in der Bedingung überprüfen, ob *ct* aus der Sorte X ist (wir schreiben dafür $X(ct)$).

Die Interpretation von statischen Funktionen ist durch die Attributierung des AST definiert. Ist die Klasse der Programme als algebraische Spezifikation über der Signatur Σ der ASM beschrieben, dann definieren die Attributierungsregeln Gleichungen, die die Klasse der Σ-Algebren einschränken. Für den Initialzustand gelten bei Vorliegen eines konkreten Programms *Prog* folgende Gleichungen über dynamischen Funktionen:

$$\begin{aligned} \text{Reference} &= \emptyset \\ content &= \emptyset \\ ct &= choose(Prog.initial) \end{aligned}$$

Die Funktion *choose* beschreibt die indeterministische Auswahl eines Elements aus einer Menge von Möglichkeiten, hier der Menge der initialen Kommandos des Programms *Prog*.

Montages erlauben die graphisch/textuelle Definition von Attributierungen. Wir führen die Notation anhand eines Beispiels ein. Die Montage in Abb. 4.3 definiert die statische und dynamische Semantik einer Zuweisung. Die erste Komponente definiert die abstrakte Syntax. Die

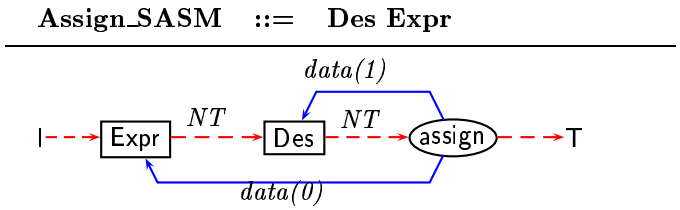


Abbildung 4.3: Beispiel einer Montages für eine Zuweisung

linke Seite der Zuweisung ist ein Designator, die rechte Seite ist ein Ausdruck. Der zweite Teil zeigt den Instruktionsgraphen, der mit einer Zuweisung identifiziert wird. Rechtecke stehen für Teilgraphen (Phrasen), Kreise repräsentieren Kommandos der Sorte Task. Der Graph definiert zwei unterschiedliche Arten von Kanten, die mit den entsprechenden Attributnamen benannt sind: durchgezogene Linien beschreiben Datenabhängigkeiten ($data(i)$), gestrichelte Kanten beschreiben den Steuerfluß ($nexttask(i)$). Zusätzlich gibt es die Kanten I und T, welche die Attribute *initial* und *terminal* für den entsprechenden Teilbaum definieren und über die der Teilbaum in den Kontext eingebunden wird. In der Montages wird das Attribut *defs* für Expr und Des textuell definiert. Attributierungsregeln können auch textuell angegeben werden. Attributnamen stehen dann entweder für ein konkretes Kommando oder für Nichtterminale, die Teilbäume repräsentieren. Hier ist natürlich auch der Bezeichner des Sprachkonzepts erlaubt, welches durch die Montage selbst spezifiziert wird. Es gibt Fälle, in denen mehrere Terminale bzw. Nichtterminale gleichen Namens in einem Sprachkonzept verwendet werden. In diesem

Fall unterscheiden wir diese Nicht-/Terminale durch eine zusätzliche eindeutige Numerierung, wie es auch bei der Definition attributierter Grammatiken üblich ist.

Eine Regel $X ::= Y_1 \ldots Y_k \; T_1 \ldots T_j$ definiert in der graphischen Darstellung Rechtecke für die Y_i und Kreise für die T_j. Ein Kommandoknoten v definiert ein Element einer Sorte T_i und es gilt $v \in \mathsf{T}_1 \cup \ldots \cup \mathsf{T}_j$. Ist v eine Phrase aus Y_i, dann gilt $v \in \mathsf{Y}_1 \cup \ldots \cup \mathsf{Y}_k$ und für v sind die Funktionen *initial* und *terminal* definiert. Aus der graphischen Darstellung leiten sich die folgenden Attributierungsregeln ab.

- Für eine Datenabhängigkeitskante (t, v) mit der Nummer i gilt:

$$\begin{array}{lcl} v \in \mathsf{T}_1 \cup \ldots \cup \mathsf{T}_j & \Rightarrow & data(t,i) = v \\ v \in \mathsf{Y}_1 \cup \ldots \cup \mathsf{Y}_k & \Rightarrow & data(t,i) = v.terminal \end{array}$$

  Datenabhängigkeitskanten von Phrasen auf Terminale oder von Phrasen auf Phrasen sind nicht erlaubt.

- Für eine Steuerflußkante (v, v') mit der Nummer i gilt:

$$\begin{array}{lcl} v \in \mathsf{T}_1 \cup \ldots \cup \mathsf{T}_j \wedge v' \in \mathsf{T}_1 \cup \ldots \cup \mathsf{T}_j & \Rightarrow & nexttask(v,i) = v' \\ v \in \mathsf{T}_1 \cup \ldots \cup \mathsf{T}_j \wedge v' \in \mathsf{Y}_1 \cup \ldots \cup \mathsf{Y}_k & \Rightarrow & nexttask(v,i) = choose(v'.initial) \\ v \in \mathsf{Y}_1 \cup \ldots \cup \mathsf{Y}_k \wedge v' \in \mathsf{T}_1 \cup \ldots \cup \mathsf{T}_j & \Rightarrow & nexttask(v.terminal,i) = v' \\ v \in \mathsf{Y}_1 \cup \ldots \cup \mathsf{Y}_k \wedge v' \in \mathsf{Y}_1 \cup \ldots \cup \mathsf{Y}_k & \Rightarrow & nexttask(v.terminal,i) = \\ & & \qquad choose(v'.initial) \end{array}$$

- Für eine Steuerflußkante (I, v) und $x = c_{X_0}(o, \mathsf{y}_1, \ldots, \mathsf{y}_k, \mathsf{t}_1, \ldots, \mathsf{t}_j)$ gilt:

$$\begin{array}{lcl} v \in \mathsf{T}_1 \cup \ldots \cup \mathsf{T}_j & \Rightarrow & x.initial = v \\ v \in \mathsf{Y}_1 \cup \ldots \cup \mathsf{Y}_k & \Rightarrow & x.initial = choose(v.initial) \end{array}$$

- Für eine Steuerflußkante (v, T) und $x = c_{X_0}(o, \mathsf{y}_1, \ldots, \mathsf{y}_k, \mathsf{t}_1, \ldots, \mathsf{t}_j)$ gilt:

$$\begin{array}{lcl} v \in \mathsf{T}_1 \cup \ldots \cup \mathsf{T}_j & \Rightarrow & x.terminal = v \\ v \in \mathsf{Y}_1 \cup \ldots \cup \mathsf{Y}_k & \Rightarrow & x.terminal = v.terminal \end{array}$$

In unserem Beispiel wird die definierende Typinformation von *Expr* und *Des* aus dem Attribut *Assign.defs* abgeleitet, das die Definitionen aus dem Kontext der *Assign*-Phrase beschreibt. Die Attributierung des Instruktionsgraphen wird entsprechend dem obigen Schema erzeugt. Es gelten die Gleichungen:

$$\begin{array}{rcl} E_x = \{ \; \mathsf{x} = c_{Assign}(o_x, \mathsf{des}, \mathsf{expr}, \mathsf{assign}) & \Rightarrow & \mathsf{expr}.defs = \mathsf{x}.defs \\ \mathsf{x} = c_{Assign}(o_x, \mathsf{des}, \mathsf{expr}, \mathsf{assign}) & \Rightarrow & \mathsf{des}.defs = \mathsf{x}.defs \\ \mathsf{x} = c_{Assign}(o_x, \mathsf{des}, \mathsf{expr}, \mathsf{assign}) & \Rightarrow & \mathsf{x}.initial = \mathsf{expr}.initial \\ \mathsf{x} = c_{Assign}(o_x, \mathsf{des}, \mathsf{expr}, \mathsf{assign}) & \Rightarrow & \mathsf{x}.terminal = \mathsf{assign} \\ \mathsf{x} = c_{Assign}(o_x, \mathsf{des}, \mathsf{expr}, \mathsf{assign}) & \Rightarrow & nexttask(\mathsf{expr}.terminal, 0) = choose(\mathsf{des}.initial) \\ \mathsf{x} = c_{Assign}(o_x, \mathsf{des}, \mathsf{expr}, \mathsf{assign}) & \Rightarrow & nexttask(\mathsf{des}.terminal, 0) = \mathsf{assign} \\ \mathsf{x} = c_{Assign}(o_x, \mathsf{des}, \mathsf{expr}, \mathsf{assign}) & \Rightarrow & data(\mathsf{assign}, 0) = \mathsf{expr}.terminal \\ \mathsf{x} = c_{Assign}(o_x, \mathsf{des}, \mathsf{expr}, \mathsf{assign}) & \Rightarrow & data(\mathsf{assign}, 1) = \mathsf{dest}.terminal \; \} \end{array} \tag{4.1}$$

Der dritte Teil einer Montage definiert die Menge von Bedingungen C_{Assign}. In unserem Beispiel wird verlangt, daß der Name des Designators im aktuellen Kontext definiert ist. Schließlich wird im vierten Teil der Montage die ASM-Regel zur Interpretation des Zuweisungs-Kommandos definiert. $addr(data(\mathsf{assign}, 1))$ bestimmt die konkrete Referenz, deren Inhalt von assign in den Wert von $value(data(\mathsf{assign}, 0))$ (genauer von $value(\mathsf{Expr}.terminal)$) geändert wird.

Eine Montages-Spezifikation definiert für jedes konkrete Programm der Sprache einen Programmgraphen $G(E, V)$, den man als Repräsentation der Semantik des Programms ansehen kann. Die Knotenmenge V ist durch die Kommandouniversen Task_i beschrieben, die aus dem Programm abgeleitet werden. Jeder Knoten ist Element eines Kommandouniversums, das im Prinzip den semantischen Typ eines Knotens festlegt. Die Kantenmenge E beschreibt Steuer- und Datenflußinformation, sie definiert die Funktionen *data* und *nexttask*. Diese graphische Repräsentation bildet die Grundlage für die Spezifikation von Transformationen durch bedingte Graphersetzungsregeln. Die Notation, die wir dabei verwenden, stammt von DÖRR (1995) und wird in Anhang A eingeführt.

4.4 Zusammenfassung

In diesem Kapitel haben wir die Teile von abstrakten Zustandsmaschinen, die wir in unseren Spezifikationen verwenden, formal eingeführt. Zusätzlich haben wir eine mehrsortige Sicht auf ASMs beschrieben, die wir im Rest der Arbeit verwenden. Mit unserer modifizierten Semantik von Montages haben wir gegenüber der Originaldefinition Restriktionen bei der Definition statischer Semantik aufgelöst und die Möglichkeit eröffnet, existierende Werkzeuge für attributierte Grammatiken zur Generierung der semantischen Analyse einzusetzen. Ein weiterer Unterschied besteht darin, daß wir die dynamische Semantik über dem abstrakten Strukturbaum angeben, da eine Sprache üblicherweise über genau dieser Struktur definiert wird. Diese Sicht deckt sich mit dem Bild des Übersetzerbauers, der die lexikalische und syntaktische Analyse einer Programmiersprache als reine Textverarbeitungsaufgabe ansieht, die nichts mit Semantik, sondern höchstens mit Pragmatik zu tun hat. Außerdem muß sich eine Spezifikation der Semantik über der konkreten Syntax mit unnötigen syntaktischen Details auseinandersetzen und implizit Abstraktionen vornehmen. Die Konzentration auf die semantisch relevanten Details ist durch die abstrakte Syntax schon vorgegeben.

5 Eine universelle Sprache zur Spezifikation dynamischer Semantik

Alle imperativen Programmiersprachen werden auf von-Neumann-Architekturen implementiert. Daher sind diese Sprachen alle über demselben semantischen Kern definiert und wir können eine universelle Sprache *AL* definieren, mit der die Semantik imperativer Programmiersprachen beschrieben werden kann.

In diesem Kapitel untersuchen wir zuerst existierende ASM-Spezifikationen von Programmiersprachen und identifizieren das zugrundeliegende abstrakte Modell für die Interpretation. Dann geben wir ein abstraktes Maschinenmodell für imperative Quell- und Zwischensprachen an, das wir aus der Analyse in Abschnitt 2.1.3 gewonnen haben. Dieses Modell umfaßt alle Konzepte, die zur operationellen Definition der dynamischen Semantik imperativer Sprachen zwingend notwendig sind. Mit Hilfe dieser Vorarbeiten definieren wir dann die konkrete Sprache *AL*.

In erster Linie sind wir an einer minimalen, aber ausreichend mächtigen Sprache zur Spezifikation der dynamischen Semantik imperativer Quellsprachen interessiert. Diese Sprache erweitern wir jedoch sofort um einige Sprachkonzepte, z. B. Funktionen, mit denen die Spezifikation von Sprachen einfacher und auch verständlicher wird, deren Abbildung auf die Minimalsprache jedoch sehr aufwendig wäre. Die Sprache *AL* enthält keine semantischen Konzepte wie z. B. Klassen, Vererbung oder Module, die aus Gründen des Systementwurfs eingeführt wurden. Diese zusätzlichen Abstraktionskonzepte haben vor allem Auswirkungen für die statische Semantik. Ihre dynamische Semantik kann mit Hilfe der Basiskonzepte beschrieben werden. Die Wahl von *AL* garantiert, daß dies immer möglich ist.

5.1 Eine Teilklasse von ASMs

Existierende ASM-Spezifikationen für Sprachen beschreiben Steuerflußinformation und Datenabhängigkeiten und definieren eine Interpretation für die Befehle der Sprache. Sprachkonzepten werden ASM Universen zugeordnet. Da wir Universen und Sorten synonym sehen können, siehe 4.1, reden wir im folgenden von Sorten. Die konkreten Instanzen eines Sprachkonzepts, in einem konkreten Programm, definieren die Elemente der entsprechenden Sorte Task.

In Kapitel 2 haben wir schon festgestellt, daß Steuer- und Datenflußrelationen adäquat zur Beschreibung von Programmiersprachsemantiken sind. Daher führen wir für die Klasse SASM ausgezeichnete Funktionsnamen ein, mit denen Steuerfluß- und Datenabhängigkeiten, sowie eine Abarbeitungsreihenfolge über Kommandos modelliert werden. *ct* ist ein abstrakter Programmzeiger, der während der Interpretation eines Programms entsprechend vorberechneter Steuerflußinformation, repräsentiert durch eine Funktion *nexttask*, weitergeschaltet wird. Außerdem verlangen wir, daß eine ASM der Klasse SASM ausgezeichnete Funktionen *initial* und *terminal* definiert. *initial* beschreibt die Menge von Kommandos die beim Start der ASM potentiell zuerst ausgeführt werden, *terminal* beschreibt das Kommando, mit der die ASM endet. Wir benutzen eine Funktion *value*, die jedem Kommando einen Wert zuordnet. Bei der Definition berücksichtigen wir schon, daß es mehrere Instanzen für ein Kommando, aber nur

eine textuelle Repräsentation im Programmgraphen gibt und modellieren das mit Hilfe des zweiten Parameters von *value*. Für die Interpretation von Operationen benötigt man manchmal die Werte anderer Berechnungen. Die dadurch auftretenden Datenabhängigkeiten werden durch die Funktion *data* beschrieben.

Definition 5.1 (SASM-Signatur)
Die Klasse SASM umfaßt ASMs mit einer Menge von Kommandosorten[1]

$$\mathsf{Task} = \mathsf{Task}_1 \cup \ldots \cup \mathsf{Task}_k$$

Die Funktionenmenge von $\Sigma_{\mathcal{A}}$ enthält auf jeden Fall folgende Funktionen:

$$\begin{aligned} ct &: \rightarrow \mathsf{Task} \\ value &: \mathsf{Task} \times \mathrm{N} \rightarrow \mathsf{Value} \\ nexttask &: \mathsf{Task} \times \mathrm{N} \rightarrow \mathsf{Task} \\ data &: \mathsf{Task} \times \mathrm{N} \rightarrow \mathsf{Task} \\ initial &: 2^{\mathsf{Task}} \\ terminal &: \mathsf{Task} \end{aligned}$$

Es gilt $\mathsf{Value} \cap \mathsf{Task} = \emptyset$.

Die dynamische Semantik einer Programmiersprache hängt im wesentlichen von der Interpretation der Kommandos ab. Diese Interpretation ist durch die Regeln der ASM (ASM-Programme) beschrieben. Wir schränken daher auch die Menge der zulässigen SASM-Programme ein. In einem SASM-Programm hängen alle Regeln von dem Programmzeiger *ct* und seinem aktuellen Zustand ab.

Definition 5.2 (SASM-Programm)
In einem SASM-Programm dürfen nur Regeln vorkommen, die die folgende Gestalt besitzen:

if $T_i(ct)$ **then**
  do in-parallel
    $r_1, \ldots, r_n$
  enddo
endif

$r_1, \ldots, r_n$ sind beliebige ASM Regeln, $T_i(ct)$ ist in der originalen ASM-Sicht ein Basisprädikat und in der Sortensicht eine Funktion, die überprüft, ob das Argument aus der Sorte gleichen Namens ist. Zu jeder Sorte Task_i *darf es nur eine Regel geben, welche die Interpretation definiert. Wir werden solche SASM Regeln im folgenden als Kommandos bezeichnen, wenn Verwechslungen mit Kommandosorten nicht zu befürchten sind.*

Wir haben noch weitere Forderungen an die Klasse SASM:

1. Ein Kommando wird nur dann ausgeführt, wenn der globale Programmzeiger *ct* zur entsprechenden Kommandosorte gehört. Damit die Semantik von *ct* immer eindeutig

1) Wir betrachten hier nur endliche Mengen von Kommandosorten. Allerdings kann man sich vorstellen, daß eine Sprache die dynamische Definition von Operationen erlaubt. In diesem Fall wäre die Menge der Instruktionstypen potentiell unendlich.

bestimmt ist, fordern wir für eine SASM, daß die Kommandosorten von $\mathcal{A}$ disjunkt sind, d.h.

$$[\![\mathsf{Task}_i]\!] \cap [\![\mathsf{Task}_j]\!] = \emptyset \quad \text{für} \quad \forall i, j \in I, i \neq j$$

Bemerkung:Diese Forderung spiegelt das intuitive Verständnis wieder, daß die Semantik eines Kommandos eindeutig sein muß. Soll Indeterminismus beschrieben werden, dann wird dieser explizit innerhalb der Regel eines Kommandotyps modelliert. ♢

2. Steuerflußinformation und Datenabhängigkeiten sind statische Informationen, daher sind *nexttask* und *data* statische Funktionen im Sinne der ASM-Terminologie.
3. Steuer- und Datenflußinformation werden getrennt modelliert. Deshalb sind Aktualisierungsregeln der Form $ct := data(ct, i)$ nicht erlaubt.

Damit ist die Klasse SASM vollständig definiert.

5.2 Ein abstraktes Maschinenmodell

Jede Programmiersprache definiert eine abstrakte Maschine. Diese Maschinen haben gemeinsame semantische Konzepte, die für die Beschreibung von Berechnungen eines Programms benötigt werden. Diese grundlegenden Konzepte sind für Quell-, Zwischen- und Zielsprachen identisch.

- **Werte und Ausdrücke**: Grundlage von Berechnungen eines Programms sind arithmetische und logische Operationen auf entsprechenden Werten. Zielmaschinen, auf denen Berechnungen letztendlich ausgeführt werden, definieren Daten wie *Integer, Float, Bool*, zusammen mit den entsprechenden Operationen. Werte und Operationen können durch Algebren spezifiziert werden.

  Eine abstrakte Modellierung dieser Basisdaten durch Z oder R, wie das in den meisten Formalismen der Fall ist, ist nicht adäquat, da die Spezifikation dann nicht mehr alle Information der Programmiersprache (z. B. Minimum, Maximum, Arithmetik) widerspiegelt.

- **Speicher**: Zentraler Bestandteil des Maschinenmodells ist der Speicher, in dem die Ergebnisse von Berechnungen gespeichert werden. Den Elementen des Speichers sind Werte zugeordnet, die geändert werden können. Der Speicher repräsentiert den Zustand eines Programms. Abstrahiert man von dem konkreten, adressierbaren Speicher von Zielmaschinen, dann kann man den Speicher als eine Ansammlung von Objekten betrachten, die jeweils einen Wert haben. Neue Objekte können erzeugt und der Wert eines Objekts kann gelesen oder durch einen anderen Wert ersetzt werden. Jedem Objekt ist eine eindeutige (symbolische) Referenz zugeordnet.

- **Typen und Operationen**: Objekte können unterschiedliche Eigenschaften haben. Die Einführung von Typen über Objekten repräsentiert diese Unterschiede und definiert eine Partitionierung des Objektraums. Auf den Werten der Objekte sind bestimmte Operationen definiert, die je nach Typ des Objekts unterschiedlich sind. Jedem Typ ist eine Wertemenge zugeordnet, zusammen mit anwendbaren Operationen über diesen Werten. Datenabhängigkeiten entstehen durch Anwendung von Operationen.

Zwischen Objekten können auch strukturelle Beziehungen existieren. Eine abstrakte Modellierung sind strukturierte Datentypen, die mehrere Objekte zusammenfassen. Zur Erzeugung neuer strukturierter Typen, benötigen wir Typkonstruktoren.

- **Steuerfluß**: In sequentiellen wie auch in parallelen Maschinenmodellen gibt es eine Reihenfolge über Berechnungen. Das heißt, es existiert Steuerflußinformation, die auf der Zielmaschine durch einen Befehlszähler und Sprünge an unterschiedliche Stellen des Programms dargestellt ist. Steuerfluß kann sich aus Datenabhängigkeiten bei Berechnungen ableiten oder explizit formuliert sein. Aus der Berechenbarkeitstheorie wissen wir, daß eine goto-Sprache mit bedingtem Sprung und den natürlichen Zahlen als Datentyp turingmächtig ist. Sequentielle Ausführung und bedingte Verzweigung und ein Befehlszähler reichen also zur Modellierung von deterministischem Steuerfluß aus.

  Desweiteren müssen wir in der Lage sein, indeterministische Ausführungen zu beschreiben, weil existierende Programmiersprachen Indeterminismus nutzen, um Freiheiten für den Übersetzer zu schaffen. Demzufolge muß es bei der Beschreibung der Semantik und auf Zwischensprachebene möglich sein, Information über Indeterminismus explizit zu beschreiben. Es gibt mehrere Möglichkeiten. Man könnte alle möglichen Reihenfolgen aufzählen und eine davon auswählen. Das ist die „ brute force“ Methode, die jedoch weder praktisch realisierbar, noch für einen Leser übersichtlich und verständlich ist. Alternativ könnte man Indeterminismus bei der Berechnung von Teilausdrücken beschreiben, indem man Einschränkungen an erlaubte Abarbeitungsreihenfolgen, z. B. in Form von Datenabhängigkeiten, definiert, anstatt die Steuerflußinformation explizit zu machen. Da wir Freiheiten jedoch explizit im Programm repräsentieren wollen, bevorzugen wir eine dritte Möglichkeit und betrachten die indeterministische Ausführung explizit als semantisches Konzept einer Programmiersprache.

Zusammen ergeben die identifizierten Konzepte ein sehr abstraktes Modell zur Beschreibung von imperativen Quell-, Zwischen- und Zielsprachen. Im vorigen Abschnitt haben wir mit der Klasse SASM schon eine Spezialisierung der abstrakten Zustandsmaschinen angegeben, die wir zur Spezifikation benutzen. Das abstrakte Maschinenmodell schränkt die Klasse SASM weiter ein, und wir können damit eine Spezifikationssprache auf der Basis von ASMs ableiten, die speziell zur Spezifikation der operationellen Semantik imperativer Programmiersprachen geeignet ist.

Die Eigenschaften des abstrakten Maschinenmodells legen zusätzliche Sorten und Funktionen von SASMs fest.

- Die Sorte Reference modelliert Speicheradressen.
- Die Sorte Value beschreibt Basiswerte.
- Die Sorte Type induziert eine Partitionierung von Value. Konstruktoren und Operationen auf Value und Type sind algebraisch spezifiziert.
- Die bedingte Verzweigung und die indeterministische Ausführungsreihenfolge über Kommandos sind durch spezielle Kommandos beschrieben, die den Steuerfluß verändern.

Bisher definiert eine SASM nur minimale Eigenschaften einer Spezifikationssprache, die zur Spezifikation imperativer Programmiersprachen geeignet ist. Zusätzlich gibt es jedoch Sprachkonzepte, die in jeder Programmiersprache vorkommen, die aber nur aufwendig in *AL* spezifiziert werden können. Das Vorhandensein dieser Konzepte erleichtert die Spezifikation und den Aufbau einer Bibliothek von Sprachkonzepten immens.

Jede imperative Sprache besitzt Abstraktionen für Funktionen und Prozeduren. Die minimale Kernsprache wäre mächtig genug, diese Konzepte zu beschreiben. Aber wir müßten einen Laufzeitkeller nachimplementieren und Funktionen definieren, die Prozeduraufrufe und Rücksprünge beschreiben. Das ist sehr aufwendig und die explizite Definition bringt keine Vorteile. Aus diesem Grund nehmen wir Funktionen mit Parametern als Konzepte in die Sprache auf, stellen einen Laufzeitkeller, den entsprechenden Typkonstruktor und die notwendigen Hilfsfunktionen wie z. B. *firsttask* zur Bestimmung des ersten Kommandos einer Funktion zur Verfügung. Aus Gründen der effizienten Implementierung partitionieren wir den Speicher in Keller- und Haldenobjekte, die unterschiedlich verwaltet werden.

Im nächsten Abschnitt definieren wir eine konkrete Ausprägung in Form der Sprache *AL*, die wir in unserem Rahmen zur Semantikdefinition verwenden.

5.3 Die konkrete Spezifikationssprache *AL*

Unser Ziel ist es, semantische Transformationen syntaktisch zu beschreiben und damit einen Kalkül aufzubauen, der durch eine Bibliothek von Ersetzungsregeln implementiert wird. Das ist jedoch nur möglich, wenn wir eine konkrete Syntax für Semantikspezifikationen haben. Die oben eingeführte Metasprache definiert Einschränkungen an diese konkrete Semantiksprache. Aus unserem Anwendungskontext ergeben sich zusätzliche Anforderungen. In diesem Abschnitt definieren wir den Kern der Sprache *AL*. Dieser Kern ist absichtlich sehr klein und einfach gehalten, da die Sprache *AL* in reale Zwischensprachen übersetzt werden soll. Übersetzungen sind für eine kleine Sprache, die nahe an Zwischensprachen liegt, einfacher zu definieren und zu verifizieren als für eine komplexe Sprache mit Konzepten, die in Zwischensprachen nicht vorkommen.

Die Operationen von *AL* sind durch die Sorte Task repräsentiert. Die Operationen der Basisdatentypen, Speicheroperationen, Steuerflußinstruktionen und Anweisungen zur Ein- und Ausgabe definieren jeweils Untersorten von Task. Wir definieren zuerst ein Typsystem für *AL*, welches parametrisierte Basisdatentypen, Paare und Zeichenketten sowie Konstruktoren für Referenzen, Mengen und Funktionen enthält und beschreiben die Basisdatentypen, die *AL* zur Verfügung stellt. Dann führen wir einen konkreten Speicher ein und definieren die Operationen auf diesem Speicher. Danach geben wir die Steuerflußkonstrukte von *AL* an und beschreiben die Operationen zur Ein- und Ausgabe.

Wir beschreiben die Sprache *AL* im folgenden informell. Eine formale Spezifikation der dynamischen Semantik ist in Anhang B angegeben.

5.3.1 Das Typsystem von *AL*

Die Typsysteme existierender, imperativer (objektorientierter) Programmiersprachen müssen auf das Typsystem von *AL* abgebildet werden können. Dazu gehören die Basisdatentypen, strukturierte Datentypen wie Verbunde bzw. Klassen und Reihungen sowie Funktionen und Referenzen.

Die Typen in *AL* sollten so einfach wie möglich und so abstrakt wie nötig gehalten sein. Unser Typsystem stellt einen Mittelweg zwischen realen Zwischensprachen und existierenden Quellsprachen dar. Es gibt die Basistypen und einen strukturierten Datentyp mit benamten Elementen. Die Sorte Type ist durch eine mehrsortige Algebra $\overline{Type}$ über den Sorten Task und

Pair definiert. Ein Paar

$$pair : \mathsf{Type} \times \mathsf{Selector} \to \mathsf{Pair}$$

assoziiert einen Selektor mit einem Typ. Ein Selektor ist entweder ein Name (String) oder ein Integer-Wert

$$[\![\,\mathsf{Selector}\,]\!] = [\![\,\mathsf{String}\,]\!] \cup [\![\,\mathsf{Int}\,]\!].$$

Die Sorte String definiert Zeichenketten beliebiger, aber endlicher Länge, Int definiert Integer-Werte. Wir gehen später noch genauer auf die Sorte Int ein.

Wir beschreiben zuerst den Aufbau von Typtermen in AL. Die Signatur von $\overline{Type}$ ist:

$$\begin{aligned}
\Sigma_{\overline{Type}} = \{ \quad \mathsf{Set} &< \mathsf{Type} \\
\mathit{int} &: \mathsf{Type} \\
\mathit{bool} &: \mathsf{Type} \\
\mathit{float} &: \mathsf{Type} \\
\mathit{string} &: \mathsf{Type} \\
\bot &: \mathsf{Type} \\
\top &: \mathsf{Type} \\
\mathit{func} &: \mathsf{Type}^* \times \mathsf{Type} \times \mathsf{Task} \to \mathsf{Type} \\
\uparrow &: \mathsf{Type} \to \mathsf{Type} \\
\mathit{set} &: \mathsf{Pair} \times \mathsf{Set} \to \mathsf{Set} \ \}
\end{aligned}$$

Die Konstanten *int*, *float*, *bool* und *string* der Typalgebra repräsentieren die Namen der Basisdatentypen *Int*, *Float Bool* und *String*. Es gilt $\mathsf{BasicType} = \{\mathit{int}, \mathit{float}, \mathit{bool}, \mathit{string}\}$. $\top$ benennt den Fehlertyp und $\bot$ den undefinierten Typ.

func ist der Konstruktor für einen Funktionstyp. Das erste Argument beschreibt die Parameter, das zweite Argument definiert den Ergebnistyp und das Task-Argument legt das erste auszuführende Kommando der Funktion fest. Eine Funktion ist durch das Paar $\langle \mathit{Funktionstyp}, \mathit{Funktionsname} \rangle$ beschrieben. Für den Prozedurtyp einer imperativen Programmiersprache ist das Ergebnis vom Typ $\bot$. $\uparrow$ erzeugt einen Referenztyp, und die Menge der Referenztypen ist durch *RefType* beschrieben. Die Funktion

$$\mathit{firsttask} \ : \ \mathsf{String} \to \mathsf{Task}$$

assoziiert einen Funktionsnamen mit dem ersten auszuführenden Kommando einer Funktion.

Der Konstruktor *set* definiert den Mengentyp Set für den $\mathsf{Set} \subseteq \mathsf{Type}$ gilt. Mengen sind hier Mengen von Paaren. Sie werden zur Modellierung von Verbunden (engl.: *records*) oder Reihungen (engl.: *arrays*) eingesetzt. Wir schreiben $set(t_0, n_0), \ldots, (t_k, n_k))$ als Abkürzung für $set((t_1, s_1), set((t_2, s_2), \ldots, set(t_k, n_k) \ldots)$.

Bemerkung: Wir modellieren eine Liste als Menge über Paaren (*Element, Position*). ♢

Trotz der einfachen Struktur unserer Typen lassen sich auch objektorientierte Strukturen und alle imperativen Typen beschreiben. Dabei definiert z. B. die Abbildung eines Verbundobjekts in AL-Objekte, ob die Reihenfolge, in der Objekte im Speicher abgelegt werden, von Interesse ist oder nicht. Jedem Kommando eines AL-Programms ist ein Typ zugeordnet, der auch $\bot$ sein kann. Das modellieren wir durch die Funktion

$$\mathit{type} : \mathsf{Task} \to \mathsf{Type}.$$

Im folgenden geben wir den Typen aus *AL* eine Semantik. Die *int* und *float* entsprechenden Basisdatentypen *Int* und *Float* sind in *AL* parametrisiert. Man könnte sich auch unterschiedliche Ausprägungen für den Datentyp *Bool* vorstellen. Zum Beispiel könnten boolesche Werte speicherbar oder nicht speicherbar sein. Aber wir haben uns in *AL* dafür entschieden, daß *Bool* fest definiert ist. Die Parametrisierung von *Int* und *Float* schränkt zwar die Menge der möglichen Integer- oder Gleitkomma-Typen ein, allerdings können alle relevanten Sprachen beschrieben werden. Die Einschränkung der Typen erlaubt die Verwendung parametrisierter, vordefinierter und als korrekt nachgewiesener Konvertierungsfunktionen für die Übersetzung von Datentypen.

- Eine konkrete Instanz von *int* wird durch drei Parameter charakterisiert. Diese sind *minint*, *maxint* und *arithmetic*. Dabei bestimmen *minint* und *maxint* den Zahlenbereich, d. h. die kleinste bzw. größte ganze Zahl. Durch den booleschen Parameter *arithmetic* wird angegeben, ob mit Überlauf oder mit Ringarithmetik gerechnet werden soll. Der Typkonstruktor für *int* ist daher erweitert:

  $$int \quad : \quad \mathrm{N} \times \mathrm{N} \times \mathsf{Bool} \rightarrow \mathsf{Type}$$

  Die folgenden Projektionen geben Auskunft über die konkrete Ausprägung des Datentyps *Int*.

  $$\begin{aligned} minint(int(n, m, a)) &\hat{=} n \\ maxint(int(n, m, a)) &\hat{=} m \\ arithmetic(int(n, m, a)) &\hat{=} a \end{aligned}$$

  Ferner sind für einen konkreten Datentyp *Int* die folgenden Operationen definiert:

  $$\begin{aligned} binary\_operation &: \mathsf{Int} \times \mathsf{Int} \rightarrow Int \\ unary\_operation &: \mathsf{Int} \rightarrow \mathsf{Int} \end{aligned}$$

  Wobei $binary\_operation \in \{+, -, *, pow, div, mod\}$ und $unary\_operation \in \{-, abs\}$.

  Eine algebraische Spezifikation des generischen Typs *Int* mit entsprechenden Konvertierungsfunktionen kann man in (Heberle und Heuzeroth, 1997) nachlesen. Eine spezielle *Int*-Instanz, die in Quellsprachen häufig vorkommt, ist *machine(int)*. In diesem Fall enthält der Datentyp *Int* die üblichen Operationen ohne Überlauf und Annahmen über die Arithmetik. Eine automatische Abbildung in eine andere *Int*-Instanz kann nicht definiert werden.

- Die Spezifikation von *Float* erfolgt gemäß dem Standard IEEE 754-1985, (IEEE, 1985). Der Konstruktor erhält als Parameter die Anzahl der Bits, die für die Darstellung einer Gleitkommazahl verwendet werden sollen.

  $$float \quad : \quad \{32, 64, 128\} \rightarrow \mathsf{Type}$$

  Mit dieser Definition lassen sich nur Gleitkommazahlen bestimmter Genauigkeit definieren[2]. Das ist jedoch unproblematisch, weil mit Hilfe einer Bibliothek Gleitkommazah-

2) Die Festlegung auf IEEE-Gleitkommazahlen ist eine Entwurfsentscheidung. Ebenso hätten wir uns auf eine Parametrisierung mit Anzahl der Bits für die Mantisse und den Exponenten, die Basis und die Position des Punkts innerhalb der Mantisse festlegen können.

len beliebiger Genauigkeit nachimplementiert werden können. Das entspricht genau der Lösung bei Sprachen für numerische Anwendungen.

- Der Datentyp *Bool* ist wie üblich definiert. Es gibt die Konstanten *false* und *true*, sowie die Operationen $\wedge$, $\vee$ und $\neg$. Aus der Logik weiß man, daß schon weniger Operatoren, z. B. $\wedge$ und $\neg$ ausreichen würden. Aber auch hier haben wir der Bequemlichkeit Rechnung getragen.

Jeder Datentyp definiert Operationen und damit Untersorten von Task. Zu den Operationen zählen auch die Konstanten eines Datentyps. Für den *Int*-Datentyp haben wir die Sorten

$$\mathsf{IntConst},\ \mathsf{IntAdd},\ \mathsf{IntSub},\ \mathsf{IntMult},\ \mathsf{IntDiv},\ \mathsf{IntPow},\ \mathsf{IntMod},\ \mathsf{IntAbs},\ \mathsf{IntNeg} \subseteq \mathsf{Task}$$

Die Sorten für die anderen Datentypen sind analog definiert, wobei wir davon ausgehen, daß die Vergleichsoperationen jeweils Erweiterungen der Datentypen im algebraischen Sinne sind.

5.3.2 Speicher

Das Ziel von *AL* ist es, maschinenunabhängig die Semantik zu beschreiben. Aus diesem Grund berücksichtigt *AL* keine Ressourcenbeschränkungen. Der Speicher wird als potentiell unendlich angesehen und enthält initial keine Objekte. Speicheradressen werden durch symbolische Referenzen (Reference) modelliert, d. h., ein neues Objekt wird durch Erzeugen einer neuen Referenz auf ein solches Objekt angelegt. Zu Objekten, deren Typ ein Grundtyp ist, werden Typ und Wert im Speichermodell verwaltet. Im Fall von strukturierten Typen werden außer der Typinformation noch Referenzen auf die entsprechenden Teilstrukturen vermerkt, die über Selektoren angesprochen werden können.

AL definiert Kommandos, die Speicher verändern bzw. Projektionen in den Speicher realisieren. Diese Kommandos sind für alle Typen der Sorte Type definiert.

$$\mathsf{New},\ \mathsf{Load},\ \mathsf{Store} \subseteq \mathsf{Task}$$

Ein New-Kommando erzeugt eine neue Instanz eines bestimmten Typs, d. h. ein neues Objekt im Speicher, und liefert eine Referenz auf dieses Objekt. Ein Store-Kommando ersetzt den Wert eines Objekts durch einen neuen Wert. Das entsprechende Objekt und der neue Wert werden von Kommandos berechnet und definieren Datenabhängigkeiten. Load liest den Wert eines Objekts aus. Die Sorte Load ist noch einmal unterteilt in

$$\mathsf{Load} = \mathsf{LoadBasic} \cup \mathsf{LoadStruct},$$

und beschreibt den lesenden Zugriff auf ein strukturiertes oder ein Basisobjekt.

Objekte werden auf dem Laufzeitkeller oder auf der Halde verwaltet. Für das Anlegen von Schachteln für lokale, globale Variablen oder Parameter auf dem Laufzeitkeller gibt es unterschiedliche Operationen.

$$\mathsf{NewGlobalFrame}, \mathsf{NewLocalFrame}, \mathsf{NewParamFrame} \subseteq \mathsf{Task}$$

Mit der Alloc-Operation werden Objekte auf der Halde angelegt.

$$\mathsf{Alloc} \subseteq \mathsf{Task}$$

Prinzipiell hätte es ausgereicht, den Speicher komplett dynamisch zu modellieren. Allerdings wäre dann Information über statisch erzeugbare Objekte nicht in *AL* ausdrückbar und hätte für die Erzeugung effizienten Codes nicht zur Verfügung gestanden.

5.3.3 Steuerfluß und Datenabhängigkeiten

In *AL* können sowohl Steuerfluß als auch Datenabhängigkeiten definiert werden.

$$\mathit{nexttask} : \mathsf{Task} \times \mathrm{N} \to \mathsf{Task}$$
$$\mathit{data} : \mathsf{Task} \times \mathrm{N} \to \mathsf{Task}$$

Abgesehen von dem letzten Kommando eines Programms, hat jedes Kommando Nachfolger im Steuerfluß. *nexttask* modelliert die Menge der möglichen Nachfolger. In unseren Spezifikationen geben wir den unterschiedlichen Nachfolgern $\mathit{nexttask}(t, i)$ eines Kommando t manchmal Namen, wenn das zu einem besseren Verständnis beiträgt. In diesem Fall gibt es eine eindeutige Abbildung von i auf einen Namen. *AL* erlaubt auch die implizite Definition von Steuerfluß anhand von Datenabhängigkeiten. In diesem Fall wird die durch *data* beschriebene Information benutzt, um in einem Vorberechnungsschritt eine Reihenfolge im Steuerfluß zu bestimmen, die mit den existierenden Datenabhängigkeiten verträglich ist.

AL-Programme können zwar als Terme über der abstrakten Syntax angesehen werden, wobei Attribute die Datenabhängigkeiten und die Steuerflußinformation repräsentieren. Jedoch definieren Daten- und Steuerflußabhängigkeiten Relationen, deren natürliche Repräsentation Graphen sind.

In *AL* gibt es zusätzlich Kommandos, die speziellen Steuerfluß modellieren.

$$\mathsf{One},\ \mathsf{All} \subseteq \mathsf{Task}$$

One beschreibt die Case-Anweisung. Eine Menge von Kommandos beschreibt die Alternativen, ein Kommando berechnet den Schlüsselwert. Jede Alternative hat eine Markierung, und es wird zu der Alternative übergegangen, deren Markierung mit dem Schlüsselwert übereinstimmt. All definiert die indeterministische Ausführungsreihenfolge über einer Liste von Kommandos.

One und All sind die einzigen Steuerflußoperationen, die man in *AL* unbedingt benötigt. Sie machen *AL* zu einer *goto*-Sprache und sichern zusammen mit dem unendlichen Speicher die Turingmächtigkeit. Weitere Kommandos sind der Funktionsaufruf und der Rücksprung aus einer Funktion.

$$\mathsf{Call},\ \mathsf{Return} \subseteq \mathsf{Task}$$

Jede Funktion (dazu zählt auch das Hauptprogramm) ist durch einen Kommandographen beschrieben, der die Anweisungen der Funktion modelliert. Ein Call-Kommando beschreibt die Verzweigung im Steuerfluß zum ersten Kommando in dem entsprechenden Kommandographen. Die Funktion *firsttask* : String → Task, setzt Funktionsnamen mit dem ersten Kommando in Kommandographen in Beziehung[3].

AL erlaubt die Definition rekursiver Funktionen. Die Kommandos können Zwischenergebnisse speichern und abhängig von der Aufruftiefe unterschiedliche Werte haben. Wir modellieren das durch den zusätzlichen Parameter der Funktion *value*, der die Aufruftiefe beschreibt. Ein Funktionsaufruf erhöht die Aufruftiefe, ein Rücksprung aus einer Funktion verringert sie. Im Prinzip erzeugen wir bei jedem Funktionsaufruf eine neue Instanz der gerufenen Funktion. Diese Modellierung ähnelt der Benutzung temporärer Variablen, die Zwischenergebnisse einer Berechnung speichern und die beim Verlassen einer Funktion einfach vergessen werden. Die Werte von Objekten können über den Keller oder die Halde zugegriffen werden.

3) Damit modelliert *firsttask* im Prinzip eine Dispatch-Tabelle

5.3.4 Ein- und Ausgabe

Wir müssen die Möglichkeit haben, die Kommunikation mit der Umgebung eines Programms zu beschreiben. Wir sehen Ein- und Ausgaben als potentiell unendliche Ströme von Zeichenfolgen an, die Ereignisse oder Daten kodieren.

$$\begin{aligned} \mathit{input} &:\rightarrow \mathsf{Char}^* \\ \mathit{output} &:\rightarrow \mathsf{Char}^* \end{aligned}$$

Initial ist der Eingabestrom vorbelegt und der Ausgabestrom leer. Es gibt in AL zwei Kommandos, die Ein- bzw. Ausgabe verändern.

$$\mathsf{Read},\ \mathsf{Write} \subseteq \mathsf{Task}$$

Ein Read-Kommando liest das erste Element der Eingabe und entfernt es aus dem Strom. Das Write-Kommando hängt ein, durch ein anderes Kommando berechnetes, Zeichen ans Ende des Ausgabestroms an.

5.4 Zusammenfassung

Unter praktischen Gesichtspunkten unterstützt AL die Konzepte existierender Zwischensprachen. In unserem Maschinenmodell können Freiheiten, wie z. B. indeterministische Ausführungsreihenfolgen, ausgedrückt werden, die für Optimierungen ausgenutzt werden können. Das Typsystem von AL ist sehr einfach, aber alle imperativen Typen können darauf abgebildet werden. Auch unterschiedlichste Ausprägungen von Basisdatentypen können in AL spezifiziert werden. Vom theoretischen Standpunkt definiert AL eine turingmächtige Sprache. Die Sprache ist sehr klein und ähnelt traditionellen Zwischensprachen in Übersetzern. Daher ist die Abbildung in eine konkrete Zwischen- oder Zielsprache einfach zu definieren und zu verifizieren. Wäre AL komplexer, dann wäre diese Abbildung ein richtiger Übersetzungsschritt. Die Sprache AL ist nicht minimal. Die Einführung von Funktionen mit Parametern und der Laufzeitkeller sind eine Entwurfsentscheidung, die wir zur Vereinfachung der initialen Spezifikation und aus Gründen der effizienten Abbildung getroffen haben.

6 Modulare Übersetzungsverifikation

In diesem Kapitel beschreiben wir unseren Ansatz zur Spezifikation von Programmiersprachsemantiken und erklären, wie damit strukturelle Induktion über operationellen Semantiken zum Nachweis der Korrektheit von Übersetzungen angewendet werden kann. Wir beweisen allgemeine Aussagen über die Korrektheit von Übersetzungen und vereinfachen damit die Verifikation.

Bevor wir über den Nachweis der Korrektheit von Übersetzungen reden können, muß klar sein, was Korrektheit einer Übersetzung überhaupt bedeutet. In Abschnitt 6.1 definieren wir Korrektheit einer Übersetzung über der Ausführung von Programmen und den beobachtbaren Auswirkungen aus Sicht der Umgebung. Wir führen den Begriff Sprachkonzept in Abschnitt 6.2 ein und haben damit die Möglichkeit, exakt über die Bestandteile einer Sprache zu sprechen. In Abschnitt 6.3 zeigen wir, wie die SASM-Spezifikation einer Sprache strukturell über den Sprachkonzepten definiert wird. Für die Verifikation argumentieren wir über diesen strukturellen Aufbau der Sprachsemantik. Die Einführung generischer Sprachkonzepte mit Minimalanforderungen an ihre Teilkonzepte und die daraus resultierende lokale Sicht auf Sprachkonzepte ergeben eine praktikable Modularisierung des Korrektheitsbeweises, siehe Abschnitte 6.4.1 und 6.4.2. Als Ergebnis haben wir allgemeine Übersetzungstheoreme bewiesen, die den Korrektheitsbeweis für die Übersetzung einer Sprache auf die Verifikation lokaler Übersetzungsregeln herunterbrechen.

6.1 Der Korrektheitsbegriff in termini von ASMs

Wir formalisieren nun den in Abschnitt 2.3.1 informell eingeführten Begriff von korrekter Übersetzung. Die Korrektheit einer Übersetzung wird über der Simulation von abstrakten Zustandsmaschinen definiert. Die Definition der Simulation von abstrakten Zustandsmaschinen entspricht semantisch der Definition aus (Zimmermann und Gaul, 1997). Wir haben die Formalisierung jedoch für unsere Zwecke angepaßt und haben zusätzlich einen Satz zum Nachweis von Simulationen angegeben, den wir später zur Verifikation von Transformationen ausnutzen.

Das Verhalten eines Programms wird von den Funktionen der ASM definiert. Im allgemeinen Fall sind jedoch nicht alle Funktionen von außen beobachtbar. Ein Beobachter kann zwei Programme nicht unterscheiden, deren Ein-/Ausgabeverhalten identisch ist. Daher sind für die Definition des Korrektheitsbegriffs nur die beobachtbaren Funktionen relevant, also z. B. nur die Ein- und Ausgabeströme. Für den Nachweis der Simulation muß man jedoch den kompletten Zustandsraum der beiden beteiligten abstrakten Zustandsmaschinen betrachten.

Die Menge der beobachtbaren Funktionen induziert eine Untersignatur Ω von Σ. Die Beobachtbarkeitseigenschaft über Funktionen definiert Äquivalenzklassen über Zuständen. Zwei Zustände q_1, q_2 sind beobachtbar gleich, wenn sich die Interpretation der beobachtbaren Funktionen nicht unterscheidet.

$$q_1 \equiv_\Omega q_2 \Leftrightarrow \forall f \in \Omega : [\![f]\!]_{q_1} = [\![f]\!]_{q_2}.$$

Definition 6.1 (Beobachtbare Zustandsübergangsrelation)
Sei $\Psi : Alg(\Sigma) \to Alg(\Omega)$ *der Restriktionsfunktor, der Zustände auf beobachtbare Zustände abbildet, wobei*

$$\Psi(q_1) = \Psi(q_2) \Leftrightarrow q_1 \equiv_\Omega q_2,$$

dann beschreibt $\longrightarrow_\Omega$: $Alg(\Omega) \times Alg(\Omega)$ *die beobachtbaren Zustandsübergänge der Zustandsübergangsrelation* $\to$: $Alg(\Sigma) \to Alg(\Sigma)$, *wenn gilt*

$$\begin{array}{l}
\forall q, q' \in Alg(\Sigma) : \Psi(q) \longrightarrow_\Omega \Psi(q') \\
\Leftrightarrow \\
q \not\equiv_\Omega q' \wedge \exists\, q_j, q_{j+1} \in Alg(\Sigma) : q_j \longrightarrow q_{j+1} \wedge \\
\forall \overline{q} \in Alg(\Sigma) : q \xrightarrow{*} \overline{q} \xrightarrow{*} q_j \wedge q \equiv_\Omega \overline{q} \\
\wedge \forall \hat{q} \in Alg(\Sigma) : q_{j+1} \xrightarrow{*} \hat{q} \xrightarrow{*} q' \wedge \hat{q} \equiv_\Omega q'
\end{array}$$

Abbildung 6.1 veranschaulicht beobachtbares Verhalten.

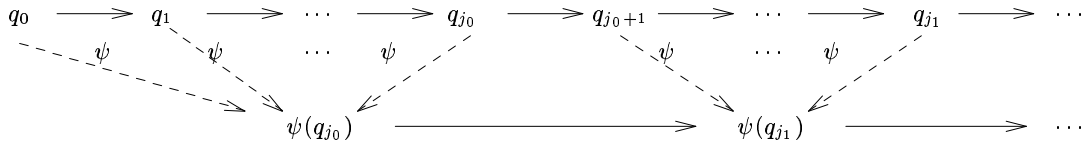


Abbildung 6.1: Beobachtbares Verhalten

Die folgende Definition legt den für uns relevanten Simulationsbegriff über beobachtbarem Verhalten fest.

Definition 6.2 (1:1-Simulation beobachtbaren Verhaltens bis auf Ressourcenfehler)
Seien $\mathcal{A}_1 = (\Sigma_1, Alg(\Sigma_1), \longrightarrow_1, I_1)$ *und* $\mathcal{A}_2 = (\Sigma_2, Alg(\Sigma_2), \longrightarrow_2, I_2)$ *zwei ASMs und* Ω_1, Ω_2 *die Mengen der beobachtbaren Funktionen und den dadurch induzierten Funktionen* Ψ_1 *und* Ψ_2. *Die ASM* $\mathcal{A}_2$ ***simuliert 1:1 das beobachtbare Verhalten*** *von* $\mathcal{A}_1$, *geschrieben* $\mathcal{A}_2 \prec_{\overline{\rho}} \mathcal{A}_1$, *falls es eine partielle injektive Funktion* $\overline{\rho} : Alg(\Omega_2) \to Alg(\Omega_1)$ *gibt mit*

$$\begin{array}{l}
\forall i \in \Psi_1(I_1), \forall i' \in \Psi_2(I_2), \forall q' \in Alg(\Omega_2) : \\
\left(\overline{\rho}(i') = i \wedge i' \xrightarrow{*}_{\Omega_2} q'\right) \Longrightarrow \left(\exists\, q \in Alg(\Omega_1) : i \xrightarrow{*}_{\Omega_1} q \wedge \overline{\rho}(q') = q\right)
\end{array}$$

Die injektive Funktion $\overline{\rho}$ bildet die beobachtbaren Zustände bzw. beobachtbare Zustandsübergänge von $\mathcal{A}_2$ auf $\mathcal{A}_1$ ab. Die Simulationsbeziehung bzgl. $\overline{\rho}$ zwischen $\mathcal{A}_1$ und $\mathcal{A}_2$ ist dann gegeben, wenn es ausgehend von Initialzuständen $i' \in \Psi_2(I_2)$, die in Beziehung zu Initialzuständen $i \in \Psi_1(I_1)$ stehen, für alle Ableitungsfolgen $i' \xrightarrow{*}_{\Omega_2} q'$ in $\mathcal{A}_2$, auch eine Ableitungsfolge $i \xrightarrow{*}_{\Omega_1} q$ gibt und $\overline{\rho}(q') = q$ gilt. Das heißt, q' beschreibt das gleiche beobachtbare Verhalten wie q. Entsprechend Definition 6.1 ist der beobachtbare Übergang $i \xrightarrow{*}_{\Omega_1} q$ nur definiert, wenn es ein $\overline{q} \in Alg(\Sigma_2)$ mit $q' = \Psi_2(\overline{q})$ gibt. Die Abbildung 6.2 veranschaulicht die Simulationsbeziehung. Die Simulationsbeziehung ist so gewählt, daß Zielprogramme deterministischer sein können als die entsprechenden Quellprogramme. Auch nichtterminierende Programme werden durch die Definition von Simulation erfaßt. Außerdem berücksichtigen wir, daß das Zielprogramm auf einer realen Maschine abläuft und somit Ressourcenbeschränkungen unterliegt. Zum Beispiel könnte eine Ausführung des Zielprogramms wegen eines Überlaufs

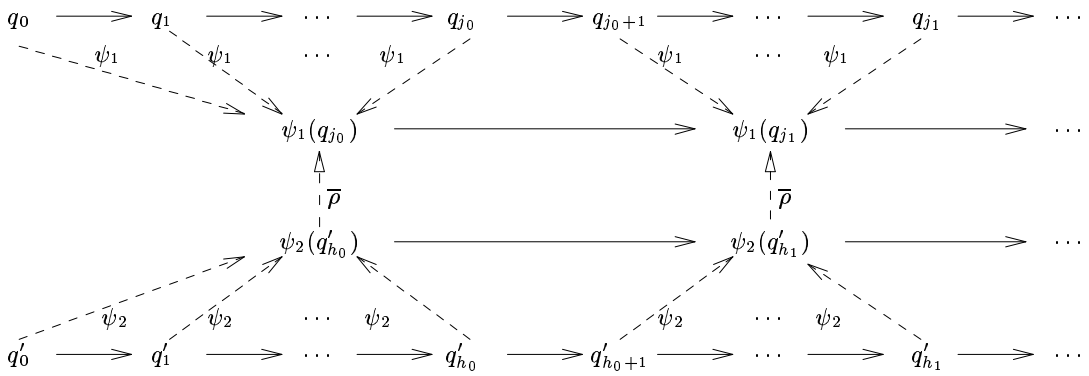


Abbildung 6.2: Simulation über beobachtbaren Zuständen

des Laufzeitkellers terminieren, obwohl das die Ausführung des Quellprogramms nicht tut. In unserer Definition der Simulation ist dieser Fall durch die Partialität der Funktion $\overline{\rho}$ erfaßt.

Indem wir nur beobachtbare Zustände für die Simulationsbeziehung betrachten, erhalten wir die Möglichkeit für Optimierungen, weil die Reihenfolge von Übergängen zwischen nicht beobachtbaren Zuständen geändert werden darf. In der Transformationsphase wird dieser Vorteil normalerweise nicht ausgenutzt, da Optimierungen, die Code umordnen, dort nicht vorkommen.

Bemerkung: Für unsere spezielle Anwendung ist in Zusammenhang mit dem Beobachtbarkeitsbegriff wichtig, daß die Quellsprache vorgibt, welches Verhalten beobachtbar ist. Das heißt, die Quellsprachdefinition bestimmt die Untersignatur $\Omega \subseteq \Sigma$, aus der die Funktion $\overline{\rho}$ abgeleitet wird. Eine Zielsprache könnte mehr Teile des Zustands als beobachtbar definieren als die entsprechende Quellsprache. Für den Korrektheitsnachweis einer Übersetzung der Quell- in die Zielsprache ist der Beobachtbarkeitsbegriff der Quellsprache ausschlaggebend. Daher wird $\Omega_2 \subseteq \Sigma_2$ bzw. die Funktion Ψ_2 passend zur Quellsprache gewählt. In dieser Arbeit gehen wir davon aus, daß sowohl in der Quellsprache, als auch in der Zielsprache nur das Ein-/Ausgabeverhalten beobachtbar ist. ♢

Nach dieser Erläuterung sind wir in der Lage, die Korrektheit einer Übersetzung formal anzugeben.

Definition 6.3 (Korrektheit einer Übersetzung formal)
*Die Übersetzung $\mathcal{C}$ einer Sprache $\mathcal{L}_1$ mit Semantik $\mathcal{A}_{\mathcal{L}_1}$ in eine Sprache $\mathcal{L}_2$ mit Semantik $\mathcal{A}_{\mathcal{L}_2}$ ist **korrekt** gdw.* $\forall \pi \in \mathcal{L}_1 : \mathcal{A}_{\mathcal{L}_2}(\mathcal{C}(\pi)) \prec_{\overline{\rho}} \mathcal{A}_{\mathcal{L}_1}(\pi)$

Für den Nachweis der Simulation des beobachtbaren Verhaltens von $\mathcal{A}_1$ durch $\mathcal{A}_2$ argumentieren wir über beobachtbare Zustandsübergänge. Allerdings können wir beobachtbare Zustandsübergänge nur unter Betrachtung konkreter Zustandsübergänge angeben. Aus diesem Grund definieren wir eine feinere Relation $\rho \subseteq Alg(\Sigma_2) \times Alg(\Sigma_1)$ und weisen die Simulationsbeziehung bezüglich dieser Relation nach. Anschließend zeigen wir, daß die Relation ρ eine Verfeinerung darstellt, die die Simulation induziert.

Definition 6.4 (ρ-Simulation abstrakter Zustandsmaschinen)
*Die ASM $\mathcal{A}_2 = (\Sigma_2, Alg(\Sigma_2), \longrightarrow_2, I_2)$ ρ-**simuliert** die ASM $\mathcal{A}_1 = (\Sigma_1, Alg(\Sigma_1), \longrightarrow_1, I_1)$,*

geschrieben $\mathcal{A}_2 \prec_\rho \mathcal{A}_1$*, falls für die Relation* $\rho \subseteq Alg(\Sigma_2) \times Alg(\Sigma_1)$ *gilt*

$$\forall i_1 \in I_1, \forall i_2 \in I_2 \forall q_2 \in Alg(\Sigma_2) :$$
$$\Big(i_2 \,\rho\, i_1 \;\wedge\; q_2 \in dom(\rho) \;\wedge\; i_2 \xrightarrow{*} q_2\Big) \Longrightarrow \Big(\exists\, q_1 \in Alg(\Sigma_1) : i_1 \xrightarrow{*}_1 q_1 \;\wedge\; q_2 \,\rho\, q_1\Big)$$

Die Definition der ρ-Simulation definiert eine n:m-Simulation ($\prec_\rho$) über den Zuständen der beiden ASMs. Sie wird zu einer 1:1-Simulation ($\prec^1_\rho$), wenn wir fordern, daß ρ eine injektive Funktion ist.

Die Abbildung 6.3 veranschaulicht den Beweis der Simulation beobachtbaren Verhaltens.

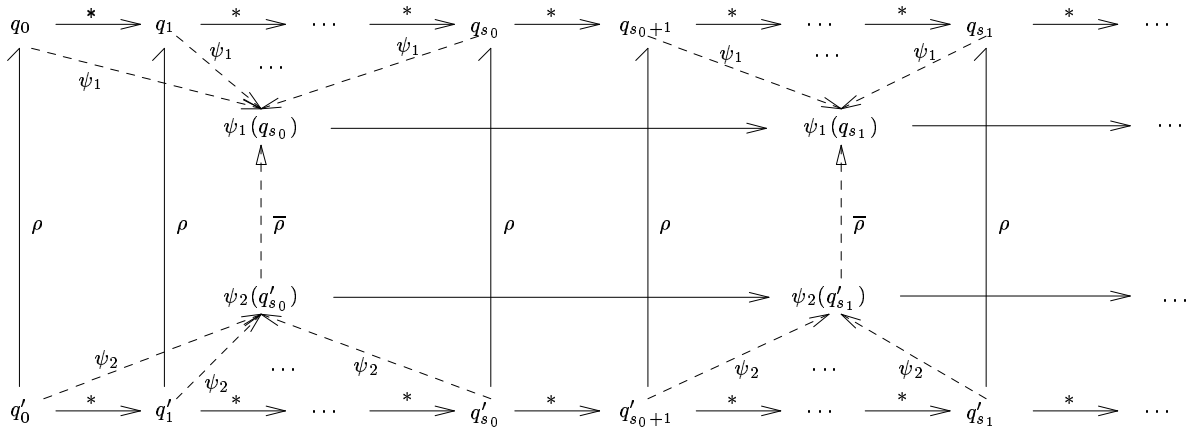


Abbildung 6.3: Simulationsbeweis mit Hilfe einer Relation ρ

Der folgende Satz besagt, wann sich aus der ρ-Simulation auch die Simulation beobachtbaren Verhaltens ableiten läßt.

Satz 6.5 (Nachweis der Simulation)
Für zwei abstrakte Zustandsmaschinen $\mathcal{A}_1$ *und* $\mathcal{A}_2$*, mit Abbildungen* Ψ_1, Ψ_2*, die Zustände auf beobachtbare Zustände abbilden, gilt* $\mathcal{A}_2 \prec_{\overline{\rho}} \mathcal{A}_1$*, wenn* $\mathcal{A}_2 \prec_\rho \mathcal{A}_1$ *und dabei die Relation* $\rho \subseteq Alg(\Sigma_2) \times Alg(\Sigma_1)$ *die folgenden Eigenschaften hat:*

$$q \;\rho\; q' \Rightarrow \overline{\rho}\,(\Psi_2(q)) = \Psi_1(q') \tag{6.1}$$

$$\Psi_2(q) \longrightarrow_{\Omega_2} \Psi_2(q') \Rightarrow \exists\, \overline{q} \in \Psi_2^{-1}(q) \cap dom(\rho) \exists\, \hat{q} \in \Psi_2^{-1}(q') \cap dom(\rho) \exists\, \overline{q}', \hat{q}' \in Alg(\Sigma_1) :$$
$$\overline{q} \longrightarrow_2 \hat{q} \;\wedge\; \overline{q}\,\rho\,\overline{q}' \;\wedge\; \hat{q}\,\rho\,\hat{q}' \;\wedge\; \Psi_1(\overline{q}') \longrightarrow_{\Omega_1} \Psi_1(\hat{q}') \tag{6.2}$$

Beweis Die Anforderung (6.1) besagt, daß der beobachtbare Teil zweier Zustände, die in Beziehung ρ stehen, mit $\overline{\rho}$ aufeinander abgebildet wird. Mit (6.2) ist ρ fein genug, um für alle beobachtbaren Zustandsübergänge auf der Zielmaschine zu folgern, daß es einen entsprechenden beobachtbaren Zustandsübergang auch auf der Quellmaschine gibt. Es ist damit ausgeschlossen, daß ρ beobachtbare Zustandsübergänge „übersieht“. Die beiden Eigenschaften zusammen sichern zu, daß ρ eine Verfeinerung von $\overline{\rho}$ ist, die alle beobachtbaren Zustandsübergänge der Zielmaschine erfaßt.

Der Beweis wird durch Induktion über die Anzahl der Zustandsübergänge $q \xrightarrow{+}_2 q'$ zwischen Zuständen q, q' aus $dom(\rho)$ geführt, die benachbart sind. Das heißt, für die es keinen Zustand $q'' \in dom(\rho)$ gibt mit $q \xrightarrow{+}_2 q'' \xrightarrow{+}_2 q'$.

Induktionsanfang: $n = 0$
Aufgrund der ρ-Simulation gibt es zu jedem Initialzustand i_2 von $\mathcal{A}_2$ einen entsprechenden Initialzustand i_1 in $\mathcal{A}_1$. Mit Anforderung (6.1) bildet $\overline{\rho}$ dann auch den beobachtbaren Anteil von i_2 auf den beobachtbaren Anteil von i_1 ab.

Induktionshypothese: Die Behauptung gilt für Zustandsfolgen $q_1, q_2, \ldots, q_n$ der Länge n, mit benachbarten $q_i \in dom(\rho)$.

Induktionsschluß: $n \curvearrowright n+1$
Sei $q\,\rho\,\overline{q}$ mit $\overline{\rho}(\Psi_2(q)) = \Psi_1(\overline{q})$ und $q \xrightarrow{+}_2 q'$ wobei q' ein bzgl. ρ benachbarter Zustand von q ist. Sei weiterhin $\overline{q} \xrightarrow{+}_1 \overline{q}'$ und $q'\,\rho\,\overline{q}'$. Dann haben wir zu zeigen, daß es zwischen diesen Zuständen nur höchstens einen beobachtbaren Zustandsübergang geben kann, denn sonst würden von ρ nicht alle beobachtbaren Zustandsübergänge, die wir für eine 1:1-Simulation beobachtbaren Verhaltens benötigen, erfaßt werden.

1. q und q' sind beobachtbar unterschiedlich ($\Psi_2(q) \longrightarrow_{\Omega_2} \Psi_2(q')$): Mit (6.1) gilt für einen bzgl. ρ benachbarten Zustand $\overline{q}'$ von $\overline{q} \in Alg(\Sigma_1)$ sofort $\overline{\rho}(\Psi_2(q')) = \Psi_1(\overline{q}')$. Wegen (6.2) gilt auch $\Psi_1(\overline{q}) \longrightarrow_{\Omega_1} \Psi_1(\overline{q}')$ und daraus folgt die 1:1-Simulation beobachtbaren Verhaltens.

2. q und q' sind beobachtbar gleich ($\Psi_2(q) = \Psi_2(q')$). In diesem Fall gilt wegen (6.1) für einen bzgl. ρ benachbarten Zustand $\overline{q}'$ von $\overline{q} \in Alg(\Sigma_1)$ mit $q'\rho\,\overline{q}'$, daß $\overline{\rho}(\Psi_2(q')) = \Psi_1(\overline{q}) = \Psi_1(\overline{q}')$. Also ist das beobachtbare Verhalten von $\overline{q}'$ und $\overline{q}'$ gleich.

Die Aussage gilt für Zustandsfolgen beliebiger, aber endlicher Länge. Nun weisen wir über einen Stetigkeitsbeweis nach, daß die Aussage auch für unendliche Zustandsfolgen gilt. Dafür verwenden wir einige Aussagen über unendliche Folgen von Zuständen und die an der Simulation beteiligten Funktionen, die in Abschnitt A.2 beschrieben sind. Wir zeigen, daß es zu jeder unendlichen Zustandsfolge s aus $\mathcal{A}_2$ eine Folge s' aus $\mathcal{A}_1$ gibt, wobei für die entsprechenden Folgen beobachtbarer Zustände $\delta(\widetilde{\Psi}_2(s))$ und $\delta(\widetilde{\Psi}_1(s'))$ gilt: $\widetilde{\overline{\rho}}(\delta(\widetilde{\Psi}_2(s))) = \delta(\widetilde{\Psi}_1(s'))$.

Sei s_i ein Präfix von s mit der Länge $i \in \mathrm{N}$, dann gilt mit Fakt A.2 $s = \bigsqcup_{i=0}^{\infty} s_i$. Wir haben schon gezeigt, daß zu jeder endlichen Folge s_i eine endliche Folge s_i' existiert und die Aussage für s_i und s_i' gilt. Da $\widetilde{\overline{\rho}} \circ \widetilde{\Psi_2}$ und Ψ_1 stetig sind und $s_i \sqsubseteq s_j$ für $i < j$, gilt auch $s_i' \sqsubseteq s_j'$ für $i < j$ und es gibt eine eindeutig definierte Folge $s' = \bigsqcup_{i=0}^{\infty} s_i'$. Für diese Folge gilt:

$$
\begin{aligned}
\delta(\widetilde{\Psi_1}(s')) &= \delta\left(\widetilde{\Psi_1}\left(\bigsqcup_{i=0}^{\infty} s_i'\right)\right) \\
&= \bigsqcup_{i=0}^{\infty} \delta(\Psi_1(s_i')) && \text{wegen der Stetigkeit von } \Psi_1 \text{ und } \delta \\
&= \bigsqcup_{i=0}^{\infty} \widetilde{\overline{\rho}}(\delta(\widetilde{\Psi_2}(s_i))) && \text{weil die Aussage für alle endlichen Folgen gilt} \\
&= \widetilde{\overline{\rho}}\left(\delta\left(\Psi_2\left(\bigsqcup_{i=0}^{\infty} s_i\right)\right)\right) && \text{weil } \widetilde{\overline{\rho}}, \delta, \Psi_2 \text{ stetig sind} \\
&= \widetilde{\overline{\rho}}(\delta(\widetilde{\Psi_2}(s))) && \text{mit der Definition von } s
\end{aligned}
$$

$\Diamond$

6.2 Sprachkonzepte

Wir betrachten in dieser Arbeit imperative und objektorientierte Programmiersprachen. Die Semantik einer Programmiersprache ist über der abstrakten Syntax definiert, die durch eine kontextfreie Grammatik $G = (N, T, P, Z)$ angegeben wird. Die abstrakte Syntax definiert die semantischen Einheiten der Programmiersprache und wir können auf dieser Ebene den bisher intuitiv verwendeten Begriff Sprachkonzept genau definieren.

Definition 6.6 (Syntax eines Sprachkonzepts)
*Die Typen der abstrakten Syntax definieren die semantischen Einheiten einer Sprache. Sie entsprechen den Terminalen T und Nichtterminalen N der kontextfreien Grammatik $G = (N, T, P, Z)$. Wir nennen diese semantischen Einheiten **Sprachkonzepte**. Die Syntax eines Sprachkonzepts* X *ist durch die Grammatik $G_X = (N_X, T_X, P_X, X)$ definiert, wobei N_X alle Nichtterminale sind, die von der Wurzel X aus erreichbar sind, T_X die erreichbaren Terminale darstellt und P_X die Menge der Produktionen zu N_X ist. Nichtterminale beschreiben zusammen mit ihren Produktionen solche Sprachkonzepte, die aus Teilkonzepten aufgebaut sind. Terminale $t \in T$ repräsentieren die **Grundkonzepte** der Sprache L und stellen die Basisfälle einer Sprachdefinition dar.*

Ein Sprachkonzept X repräsentiert die Menge der abstrakten Strukturbäume mit Wurzel X, die durch die Grammatik G_X definiert werden. Wenn wir im folgenden von einer Instanz des Sprachkonzepts X sprechen, dann ist damit ein abstrakter Strukturbaum gemeint, der aus der Wurzel X abgeleitet werden kann.

Wir wollen formale Beweise über Transformationen führen. Deshalb müssen wir auch die Bedeutung eines Sprachkonzepts formal definieren. Jedes Sprachkonzept entspricht einem Knotentyp der abstrakten Syntax, mit dem statische und dynamische, semantische Information assoziiert ist.

Definition 6.7 (Sprachkonzept)
Ein Sprachkonzept X *ist ein Tripel $(G_X, [\![]\!]_{static}, [\![]\!]_{dyn})$. Dabei ist*

1. *G_X die Syntax des Sprachkonzepts,*
2. *$[\![]\!]_{static}$ die statische Semantik von $L(G_X)$ und*
3. *$[\![]\!]_{dyn}$ die dynamische Semantik von $L(G_X)$.*

Entspricht ein Sprachkonzept X einem Nichtterminal der abstrakten Syntax, dann benötigt man zur Definition der statischen und dynamischen Semantik $[\![]\!]_{static_X}$ und $[\![]\!]_{dyn_X}$ die statische und dynamische Semantik der Teilkonzepte, über denen X definiert ist. $[\![]\!]_{static}$ ist durch eine Menge von Attributierungsregeln spezifiziert. $[\![]\!]_{dyn}$ ordnet jedem Wort aus $L(G_X)$ eine ASM zu und benutzt dafür $[\![]\!]_{static}$. Sprachkonzepte sind durch eine Menge von Montages spezifiziert.

Bemerkung: Manche Sprachkonzepte haben nur statische semantische Eigenschaften. Zum Beispiel geht die Definition von Typen nur als statische semantische Information in die dynamische Semantik anderer Sprachkonzepte ein, hat selbst aber keine dynamische Semantik. ♢

Mit unserer jetzigen Definition sind Sprachkonzepte für genau eine Sprache definiert. Die Betrachtung imperativer Sprachen hat jedoch gezeigt, daß die meisten Sprachen gleiche oder zumindest sehr ähnliche Sprachkonzepte verwenden. Ein Sprachkonzept, welches in unterschiedli-

chen Sprachen vorkommt, unterscheidet sich höchstens in den unterschiedlichen Ausprägungen seiner Teilsprachkonzepte[1]. In unserem Schleifenbeispiel (3.1) waren die semantischen Details der Abbruchbedingung und des Rumpfs sind für die lokale Semantik der Schleife nicht wichtig. Die Semantik der Schleife konnte *generisch* und damit unabhängig von den Teilkonzepten angeben werden. Wir haben nur bestimmte Minimalanforderungen für die Teilkonzepte angenommen. Im folgenden nutzen wir solche generischen Definitionen von Sprachkonzepten für die Sprachspezifikation und modularisieren damit den Korrektheitsnachweis für Übersetzungen.

In den nächsten Abschnitten erklären wir unser Vorgehen im Detail.

6.3 Strukturelle Definition dynamischer Semantik

Wir benutzen die Klasse SASM zur Definition von Semantik. Damit wissen wir zwar, wie eine Semantikdefinition in Form einer abstrakten Zustandsmaschine aussieht, über die Definition dieser ASM haben wir aber noch nichts ausgesagt. Im folgenden beschreiben wir die Konstruktion der Abbildung $[\![\,]\!]_{dyn}$.

Legen wir die Grammatik $G = (N, T, P, Z)$ zugrunde, dann ordnet die Spezifikation der dynamischen Semantik jedem abstrakten Strukturbaum mit einer Wurzel $\mathsf{n} \in N \cup T$ eine abstrakte Zustandsmaschine $\mathcal{A}_\mathsf{n}$ zu. Sei $\mathsf{n} ::= \mathsf{n}_1 \ \ldots \ \mathsf{n}_k$ eine Produktion aus P, dann definieren die $\mathcal{A}_{\mathsf{n}_i}$ Teile von $\mathcal{A}_\mathsf{n}$. Da die Semantik induktiv über der Struktur der abstrakten Syntax definiert ist, kann die Semantik von $\mathcal{A}_\mathsf{n}$ direkt auf der Semantik der $\mathcal{A}_{\mathsf{n}_i}$ aufgebaut werden. Die Basisfälle der Semantikdefinition bilden die ASM-Semantiken für die Terminalsymbole $\mathsf{t} \in T$.

Die statische Semantik $[\![\,]\!]_{static_\mathsf{n}}$ ist durch eine attributierte Grammatik AG_n definiert, die unter Verwendung der AGs der Teilkonzepte n_i konstruiert wird. Wir setzen bei der Definition der dynamischen Semantik voraus, daß die statische Semantik und alle Kontextinformationen, die man zur Definition der dynamischen Semantik benötigt, in einem vorhergehenden Analyseschritt korrekt berechnet wurden. Im einzelnen Fall gehen wir nicht weiter auf die Berechnungsstrategie ein. Zur statischen Semantik gehört zum Beispiel die Information über Typen von Variablen. Die Definition einer Variablen, d. h. die Erzeugung eines neuen Objekts, wird in der dynamischen Semantik beschrieben.

Die Informationen der statischen Semantik gehen als statische Funktionen in die einzelnen ASMs $\mathcal{A}_\mathsf{n}$ ein. Zum Beispiel wird ein Attribut, das den statischen Typ eines Knotens beschreibt, in der dynamischen Semantik durch die statische Funktion *type* : $\mathsf{Node} \to \mathsf{Type}$ repräsentiert. Benötigt man Kontextinformation für die Definition der dynamischen Semantik eines Knotens n, dann geht diese mit in den Initialzustand von $\mathcal{A}_\mathsf{n}$ ein. Da wir im folgenden immer über die dynamische Semantik sprechen, schreiben wir $[\![\,]\!]$ anstatt $[\![\,]\!]_{dyn}$ und unterscheiden zwischen dynamischer und statischer Semantik bei Bedarf.

Wir gehen davon aus, daß die Semantik der Quellsprache durch eine Montages-Spezifikation angegeben ist. Jedem Startsymbol eines Sprachkonzepts wird durch die Sprachspezifikation eine Montage zugeordnet, die beschreibt wie $[\![\,]\!]_{static}$ und $[\![\,]\!]_{dyn}$ unter Verwendung der Semantiken der Teilkonzepte, aus denen das Sprachkonzept aufgebaut ist, konstruiert wird. Der Kompositionsmechanismus für dynamische Semantik ist über Daten- und Steuerflußinformation aufgebaut. Jede Montage legt eine Menge ausgezeichneter Kommandos *initial* fest, die im Steuerfluß gesehen die ersten auszuführenden Kommandos eines Sprachkonzepts sind, und

1) Eventuell unterscheidet sich auch nur die konkrete Syntax, während die assoziierte Semantik dieselbe ist

beschreibt mit *terminal* das letzte Kommando der Abarbeitung. Zur Festlegung dieser ausgezeichneten Aufgaben ist es unter Umständen notwendig, die Teilkonzepte rekursiv bis hinab zu den Grundkonzepten zu betrachten. Wird ein Sprachkonzept nun als Teil eines anderen Sprachkonzepts verwendet, dann werden die initialen und terminalen Kommandos über Steuer- und Datenabhängigkeiten in den Kontext eingebunden. Ein Datum, das im Kontext eines Konzepts S' mit Startsymbol S' berechnet wurde, wird über das Ergebnis von $S'.terminal$ zugegriffen. Mit diesem Konstruktionsprinzip läßt sich eine induktive Definition der Funktionen *data* und *nexttask* ableiten, aus der sich für einen Strukturbaum die konkreten Steuer- und Datenflußinformationen berechnen. Der einem Programm entsprechende Daten- und Steuerflußgraph über Kommandos ist Teil des Initialzustands der abstrakten Zustandsmaschine, welche die dynamische Semantik beschreibt. Die Regeln zur Interpretation der Kommandos sind ebenfalls durch die Menge der Montages gegeben.

Für einen konkreten AST eines Programms ergibt sich die komplette abstrakte Zustandsmaschine induktiv durch Vereinigung ($\uplus$, siehe Abschnitt 4.2 und A.1.1) der ASMs, welche die Semantik von Teilbäumen beschreiben und der ASM, die die Komposition beschreibt. Voraussetzung für eine sinnvolle Vereinigung ist die Konsistenz der Interpretation von Funktionen, die in mehreren der beteiligten ASMs vorkommen.

Beispiel 6.1 Die Montage aus Abbildung 4.3 auf Seite 71 definiert folgende Vorschrift zur Konstruktion einer ASM $\mathcal{A}_{\mathsf{Assign}}$.

- Die Konstruktionsvorschrift für die Semantik der Teilkonzepte ist durch die Montages Expr und Des spezifiziert.
- Die Montage Assign selbst definiert eine ASM $\mathcal{A}_{\mathsf{assign}}$ mit Sorten

  $$\mathsf{Assign} < \mathsf{Task}, \mathsf{Reference}, \mathsf{Type}, \mathsf{Value}$$

  und folgender Signatur:

  $$\begin{aligned} \Sigma = \{ \quad & \mathit{data} : && \mathsf{Task} \times \mathrm{N} \to \mathsf{Task} \\ & \mathit{nexttask} : && \mathsf{Task} \times \mathrm{N} \to \mathsf{Task} \\ & \mathit{value} : && \mathsf{Task} \times \mathrm{N} \to \mathsf{Value} \\ & \mathit{intial} : && 2^{\mathsf{Task}} \\ & \mathit{terminal} : && \mathsf{Task} \ \} \end{aligned}$$

  Entsprechend der Definition von Montages (vgl. Abschnitt 4.3) erhalten wir für einen Term $\mathsf{x} = c_{Assign}(o_x, \mathsf{des}, \mathsf{expr}, \mathsf{assign})$, der eine Zuweisung über den Teiltermen eines Designators des und eines Ausdrucks expr beschreibt, die Einschränkungen (4.1) an den Zustandsraum (genauer an die statischen Funktionen) von $\mathcal{A}_{\mathsf{assign}}$.
- Die Regelmenge ist durch die Regel zur Interpretation von Assign-Kommandos definiert.

Die Konstruktionsvorschrift für die ASM $\mathcal{A}_{\mathsf{Assign}}$ ist dann:

$$\mathcal{A}_{\mathsf{Assign}} = \mathcal{A}_{\mathsf{assign}} \uplus \mathcal{A}_{\mathsf{Expr}} \uplus \mathcal{A}_{\mathsf{Des}}.$$

Die ASMs $\mathcal{A}_{\mathsf{Expr}}$ und $\mathcal{A}_{\mathsf{Des}}$ werden für Terme $\mathsf{des} = c_{Des}(o_{\mathsf{des}}, \dots)$ und $\mathsf{expr} = c_{Expr}(o_{\mathsf{expr}}, \dots)$ entsprechend ihrer Montages-Spezifikation konstruiert. ♢

6.4 Korrekte Übersetzung von Sprachkonzepten

Die Semantik einer Sprache kann durch Montages modular über einzelnen Sprachkonzepten angegeben werden. In diesem Abschnitt zeigen wir, wie die Zerlegung in Sprachkonzepte ausgenutzt werden kann, um die Übersetzung einer Sprache und die Verifikation dieser Übersetzung zu modularisieren. Die korrekte Übersetzung eines Sprachkonzepts ist wie üblich gegenüber der Semantik der Quelle und des Ziels definiert.

Definition 6.8 (Korrekte Übersetzung eines Sprachkonzepts)
Sei $\mathsf{S} = (G_S, [\![\,]\!]_{static}, [\![\,]\!]_{dyn})$ *ein Sprachkonzept,* n *ein Strukturbaum von* $L(G_S)$ *und* $\mathcal{C}(\mathsf{n})$ *eine Übersetzung von* n *mit Semantik* $[\![\,\mathcal{C}(\mathsf{n})\,]\!]'$. S *wird genau dann von* $\mathcal{C}$ *korrekt übersetzt, wenn für alle Strukturbäume* $\mathsf{n} \in L(G_S)$ *gilt:*

$$[\![\,\mathcal{C}(\mathsf{n})\,]\!]' \prec_{\overline{p}} [\![\,\mathsf{n}\,]\!].$$

Aus dieser Definition können wir gleich eine hinreichende Bedingung für die korrekte Übersetzung einer kompletten Sprache ableiten.

Lemma 6.9 (Korrekte Übersetzung von Sprachen)
Eine Sprache $\mathcal{L}$ *mit abstrakter Syntax* $G = (N, T, P, Z)$ *wird durch eine Abbildung* $\mathcal{C}$ *korrekt in eine Sprache* $\mathcal{L}'$ *übersetzt, wenn das Sprachkonzept* Z *von* $\mathcal{C}$ *korrekt übersetzt wird.*

Für den Beweis nutzen wir das folgende Fakt.

Fakt 6.1 Die Semantik einer Sprache mit abstrakter Syntax $G = (N, T, P, Z)$ ist das Sprachkonzept $\mathsf{Z} = (G, [\![\,]\!]_{static}, [\![\,]\!]_{dyn})$.

Beweis Die Aussage folgt direkt aus der Definition 6.8 und Fakt 6.1. Das Sprachkonzept Z repräsentiert alle Programme der Sprache, die aus der Wurzel Z abgeleitet werden können. Mit Definition 6.3 folgt dann, daß die Sprache $\mathcal{L}$ von $\mathcal{C}$ korrekt übersetzt wird. ♢

Auf den ersten Blick scheint damit schon die gewünschte Modularisierung der Übersetzungsverifikation erreicht. Daß dem nicht so ist, wird klar, wenn wir uns noch einmal genau ansehen, wie die Semantik von Sprachkonzepten konstruiert wird. Eine Montage S beschreibt eine Konstruktionsvorschrift für eine konkrete ASM, in Abhängigkeit der Konstruktionsvorschriften der Teilkonzepte von S. Die Spezifikation selbst ist damit zwar generisch, aber die ASM kann nur konstruiert werden, wenn man die konkreten Vorschriften für die Teilkonzepte kennt. Wir müssen also alle Montages zu G_S betrachten, wenn wir die Semantik eines Terms von S betrachten. In Beispiel 6.1 läßt sich nur eine Konstruktionsvorschrift für die Semantik von Zuweisungen ableiten, wenn bekannt ist, was die Definitionen für $\mathsf{e}.\mathit{initial}$, $\mathsf{e}.\mathit{terminal}$, $\mathsf{d}.\mathit{initial}$ und $\mathsf{d}.\mathit{terminal}$ sind. Das setzt sich fort für die Semantikdefinition von Ausdrücken und Designatoren. Für die Verifikation einer Übersetzung müssen wir daher immer den gesamten Teilbaum mit allen Teilkonzepten betrachten. Für das Sprachkonzept Z bedeutet das, daß wir eine Übersetzung gegenüber der kompletten Sprachsemantik verifizieren müssen.

Eigentlich wollen wir in Beweisen über die „lokale“ Semantik von Sprachkonzepten argumentieren und dabei von Zustandsübergängen abstrahieren, die durch Teilkonzepte definiert werden. Für einen aus $X ::= Y\ Z$ abgeleiteten Term x sind wir an folgender Sicht auf die entsprechende Zustandsfolge interessiert.

$$q_0 \xrightarrow{*}_{\mathsf{x}} q_j \xrightarrow{\mathcal{A}_\mathsf{Y}} q_{j+1} \xrightarrow{*}_{\mathsf{x}} q_k \xrightarrow{\mathcal{A}_\mathsf{Z}} q_{k+1} \xrightarrow{*}_{\mathsf{x}} \dots$$

6.4.1 Generische Sprachkonzepte mit Minimalanforderungen an ihre Teilkonzepte

[Generische Sprachkonzepte mit Minimalanforderungen an ihre Teilkonzepte] Der Montages-Ansatz, wie wir ihn bisher verwenden, hat zwei Nachteile. Erstens muß man für jedes Nichtterminal über dessen komplette ASM-Semantik argumentieren, wenn man die Korrektheit von Übersetzungen beweisen will und zweitens können Sprachkonzepte aus einer existierenden Spezifikation nur unverändert wiederverwendet werden. Unverändert bedeutet inklusive der Teilkonzepte, über denen das Sprachkonzept definiert ist. Wir lösen beide Probleme, indem wir Sprachkonzepte generisch definieren und für die eingesetzten Teilkonzepte keine oder nur minimale Annahmen machen.

Beispiel 6.2 Die Semantik der Hintereinanderausführung von Anweisungen Stat; Stats können wir generisch ohne zusätzliche Annahmen an die konkreten Semantiken von Stat und Stats spezifizieren. Als Zustandsfolge betrachten wir dann:

$$q_0 \xrightarrow{\mathcal{A}_{\mathsf{Stat}}} q_1 \xrightarrow{\mathcal{A}_{\mathsf{Stats}}} q_2$$

♢

In Abschnitt 3.3 haben wir schon das Beispiel der Schleife vorgestellt.

Beschreiben wir diese minimalen Eigenschaften für Teilkonzepte ebenfalls mit abstrakten Zustandsmaschinen, dann bleiben wir erstens im selben Formalismus und zweitens können wir unsere Simulationsbeweistechniken ausnutzen, um korrekte Instantiierungen zu beschreiben. Wir betrachten die Semantik eines Sprachkonzepts als abstrakte Zustandsmaschine, die andere abstrakte Zustandsmaschinen „aufruft“, deren Verhalten nicht explizit bekannt ist.

Definition 6.10 (Generisches Sprachkonzept mit ASMs als Minimalanforderungen) *Ein Sprachkonzept* S*, das an die Semantik seiner Teilkonzepte* $\mathsf{S}_1 \ldots \mathsf{S}_k$ *nur minimale Anforderungen* $\mathcal{A}_1, \ldots, \mathcal{A}_k$*, in Form abstrakter Zustandsmaschinen stellt, heißt* ***generisches Sprachkonzept*** *mit ASM-Minimalanforderungen und wird mit* $\mathsf{S}(S_1 < \mathcal{A}_1, \ldots, S_k < \mathcal{A}_k)$ *bezeichnet.*

Für die Minimalanforderung $\mathcal{A}_i$ *gilt, daß die Funktionen* $f \in \Sigma_{\mathcal{A}_i}$*, über die eine Minimalanforderung* $\mathcal{A}_i$ *in* S *eingebunden wird, beobachtbar sind, d. h.* $f \in \Omega_{\mathcal{A}_i}$.

Die lokale Semantik eines generischen Sprachkonzepts ergibt sich durch Einsetzen der Minimalanforderung:

$$[\![\, \mathsf{S} \,]\!](\mathcal{A}_1, \ldots, \mathcal{A}_k) = [\![\, S \,]\!] \uplus \mathcal{A}_1 \uplus \ldots \uplus \mathcal{A}_k$$

Die Definition eines generischen Sprachkonzepts mit widersprüchlichen Minimalanforderungen $\mathcal{A}_i$, d. h. die Vereinigung $\mathcal{A}_1 \uplus \ldots \uplus \mathcal{A}_k$ ist nicht gültig und konsistent (vgl. 4.2), macht keinen Sinn, weil dann $[\![\, \mathsf{S} \,]\!](\mathcal{A}_1, \ldots, \mathcal{A}_k)$ keine Semantik hat. Wir gehen daher im folgenden davon aus, daß die von uns betrachteten generischen Sprachkonzepte widerspruchsfrei sind.

Fakt 6.2 Ein Grundkonzept (abgeleitet aus den Terminalen einer Sprache) ist ein spezielles generisches Sprachkonzept ohne Parameter.

Es steht dem Sprachentwerfer frei, beliebige Sprachkonzepte zu definieren und zu komponieren und so eine beliebig sinnlose Sprachspezifikation zu erstellen. Es ist unmöglich festzustellen, ob eine bestimmte Semantik intendiert war oder nicht. Wir können im allgemeinen Fall nicht einmal die Einhaltung von Minimalanforderungen bei der Instantiierung von Sprachkonzepten automatisch überprüfen. Wir sind jedoch nur an der Übersetzung von Sprachen interessiert, deren Semantik **wohlgeformt** ist. Die folgende Definition von wohlgeformter Instantiierung eines Sprachkonzepts gibt uns ein Kriterium für die Wohlgeformtheit einer Sprachdefinition.

Definition 6.11 (Wohlgeformte Instanz eines generischen Sprachkonzepts)
Eine Instanz $\mathsf{S}(\mathsf{s}_1, \ldots, \mathsf{s}_k)$ *eines generischen Sprachkonzepts* $\mathsf{S}(S_1 < \mathcal{A}_1, \ldots, S_k < \mathcal{A}_k)$ *ist* ***wohlgeformt****, wenn:*

(A) Es gibt injektive Funktionen $\overline{\rho}_1, \ldots, \overline{\rho}_k$ *mit*

$$\forall i, 1 \leq i \leq k : [\![\mathsf{s}_i]\!] \prec_{\overline{\rho}_i} \mathcal{A}_i$$

(B) Die Zustandsräume der abstrakten Zustandsmaschinen $[\![\mathsf{S}]\!](\mathcal{A}_1, \ldots, \mathcal{A}_k)$ *und* $[\![\mathsf{s}_i]\!]$ *stehen nur über die Funktionen* $\overline{\rho}_i$ *in Beziehung und sind ansonsten disjunkt.*

(C) Mit den Relationen ρ_i*, aus denen wir die Funktionen* $\overline{\rho_i}$ *ableiten (vgl. Satz 6.5), gilt:*

$$\forall q \in I_{[\![\mathsf{s}_i]\!]} \exists\, q' \in I_{\mathcal{A}_i} : q \,\rho\, q' \wedge \forall q' \in I_{\mathcal{A}_i} \exists q \in I_{[\![\mathsf{s}_i]\!]} : q \,\rho\, q'$$

(D) Die Menge der Aktualisierungsregeln aus $[\![\mathsf{S}(\mathsf{s}_1, \ldots, \mathsf{s}_k)]\!]$ *ist konsistent.*

Die formale Semantik $[\![\mathsf{S}]\!]([\![\mathsf{s}_1]\!], \ldots, [\![\mathsf{s}_k]\!])$ *ergibt sich durch Vereinigung der ASM* $[\![\mathsf{S}]\!]$ *mit den durch* $\overline{\rho}_i$ *angepaßten ASMs* $[\![\mathsf{s}_i]\!]$*.*

Die Anforderung (A) besagt, daß alle konkreten Teilkonzepte, die eingesetzt werden, auch die Semantik der Minimalanforderungen simulieren. $[\![\mathsf{s}_i]\!] \prec_{\overline{\rho}_i} \mathcal{A}_i$ ist genau die Simulationsbeziehung, wie wir sie für die Korrektheit von Übersetzungen nachweisen. Die Funktion $\overline{\rho}_i$ wird jeweils durch die beobachtbaren Funktionen $\Omega_{\mathcal{A}_i}$ der Minimalanforderung $\mathcal{A}_i$ induziert. Die Zielsprach-ASM, hier das konkrete Sprachkonzept $[\![\mathsf{s}_i]\!]$, könnte mehr beobachtbare Zwischenzustände definieren. Für den Nachweis von $[\![\mathsf{s}_i]\!] \prec_{\overline{\rho}_i} \mathcal{A}_i$ sind jedoch die beobachtbaren Funktionen von $\mathcal{A}_i$ relevant. Wir lassen im folgenden die Indizierung in $\prec_{\overline{\rho}_i}$ weg, wenn klar ist, auf welche beobachtbaren Funktionen sie sich bezieht. Die Simulationsbeziehung zwischen einer Minimalanforderung und dem entsprechenden konkreten Teilkonzept garantiert zusammen mit der Anforderung (B), daß die Ausführung einer konkreten ASM für das Teilkonzept nur Auswirkungen auf den Kontext hat, die auch von der Minimalanforderung beschrieben werden. Wird im Initialzustand von $[\![\mathsf{s}_i]\!]$ für die Interpretation einer Funktion ein bestimmter Wert angenommen, der jedoch im Kontext von $[\![\mathsf{s}_i]\!]$ verändert werden könnte, dann ist im allgemeinen nicht sicher, daß die Voraussetzung an $[\![\mathsf{s}_i]\!]$ auch nach Einbau in den Kontext noch zugesichert ist. (C) garantiert, daß in diesem Fall auch für die Minimalanforderung $\mathcal{A}_i$ eine entsprechende Annahme gemacht wird. Zusätzlich sichert die Anforderung (C) zu, daß durch die Instantiierung keine Inkonsistenzen bei der Interpretation von Funktionen auftreten, die nicht auch auf Ebene der Minimalanforderungen vorhanden sind. Die Anforderung (D) ist für unseren Ansatz nicht zwingend notwendig, jedoch sind widersprüchliche Aktualisierungen ein Zeichen für falsche Spezifikationen an deren Übersetzung wir nicht interessiert sind.

Für die Einhaltung von (C) können wir ein hinreichendes und einfach zu überprüfendes Kriterium angeben.

Lemma 6.12
Eine abstrakte Zustandsmaschine $\mathcal{A}$ ist unabhängig von dem Kontext, in dem sie aufgerufen wird, wenn im Initialzustand keine Annahmen über dynamische Funktionen $f \in \Sigma$ gemacht werden. Wenn also für dynamische Funktionen f gilt: $\forall \overline{x} \in dom(f) : f(x) = \perp$, wobei $\perp$ keine Information repräsentiert.

Beweis Macht eine abstrakte Zustandsmaschine keine Annahmen über die Initialisierung ihrer Funktionen, dann ist jede beliebige Initialisierung gültig und es kann damit keine Initialisierung durch den Kontext geben, die zu inkonsistentem Verhalten führt. ♢

Ist das hinreichende Kriterium nicht erfüllt, dann ist es eine zusätzliche Beweisverpflichtung sicherzustellen, daß die Initialzustände für alle $[\![\mathsf{s}_i]\!]$ auch im Gesamtkontext erfüllt sind.

Da wir in unseren Anwendungen nur SASMs betrachten und die Konstruktion von $[\![\mathsf{S}]\!]([\![\mathsf{s}_1]\!], \ldots, [\![\mathsf{s}_k]\!])$ durch die Vereinigung von SASMs definiert ist, gibt das folgende Lemma ein hinreichendes Kriterium für die Konsistenz der Regelmenge.

Lemma 6.13 (Konsistenz der Vereinigung von Regelmengen)
Seien $\mathcal{A}_1$ und $\mathcal{A}_2$ zwei SASMs, deren Aktualisierungsregeln jeweils konsistent sind, dann ist die Regelmenge von $\mathcal{A}_1 \uplus \mathcal{A}_2$ konsistent, wenn

1. *Die Mengen von Sorten aus $\mathcal{A}_1$ und $\mathcal{A}_2$, die Kommandos beschreiben, disjunkt sind oder*
2. *die Aktualisierungsregeln zur Interpretation von Kommandos T, die sowohl in $\mathcal{A}_1$ als auch in $\mathcal{A}_2$ vorkommen, identisch sind.*

Beweis Eine SASM definiert für jede Kommandosorte genau eine Aktualisierungsregel. Gibt es keine gemeinsamen Kommandosorten in $\mathcal{A}_1$ und $\mathcal{A}_2$, dann sichert die Form der SASM-Regeln und die Konsistenz der einzelnen Regelmengen sofort die Konsistenz der vereinigten Regelmenge. Gibt es gemeinsame Kommandosorten, dann sichert die 2. Voraussetzung des Lemmas, daß die Aktualisierungsregeln eines Kommandos sich nicht verändern und daher die vereinigte Regelmenge konsistent ist. ♢

Fakt 6.3 Ein Grundkonzept ist wohlgeformt, da es keine Minimalanforderungen gibt, die zu erfüllen sind.

Fakt 6.4 Ein konsistentes konkretes Sprachkonzept ist wohlgeformt.

Fakt 6.5 $\mathsf{S}(\mathcal{A}_1, \ldots, \mathcal{A}_k)$ ist eine wohlgeformte Instanz eines generischen Sprachkonzepts $\mathsf{S}(S_1 < \mathcal{A}_1, \ldots, S_K < \mathcal{A}_k)$, falls die Vereinigung $\mathcal{A}_1 \uplus \ldots \uplus \mathcal{A}_k$ gültig und konsistent ist. Sind die Minimalanforderungen eines generischen Sprachkonzepts widersprüchlich, dann gibt es keine wohlgeformten Instanzen dieses Sprachkonzepts.

Beispiel 6.3 Sei Call ein Sprachkonzept, das einen Funktionsaufruf generisch in der Berechnung seiner Parameter definiert (siehe Abschnitt 9.6.2). Die Minimalanforderung $\mathcal{A}_{Arguments}$ definiert für die Auswertung der Parameter eine feste Berechnungsreihenfolge. Eine Instantiierung mit einem Konzept Args, das die Parameter in beliebiger Reihenfolge auswertet, wäre nicht wohlgeformt. Wir können nämlich nicht für alle möglichen Berechnungsfolgen von $[\![\mathsf{Args}]\!]$ eine entsprechende Berechnungsfolge in $\mathcal{A}_{Arguments}$ finden und daher nicht die Simulation nachweisen. ♢

In unseren Anwendungen ist vor allem das Kriterium (A) für die Wohlgeformtheit von Instanzen eines Sprachkonzepts relevant. Die Anforderung (B) erfüllen wir durch entsprechende Wahl von Funktionsnamen und durch die Konstruktion der Trägermenge. (C) kann üblicherweise aus der Erfüllung von (A) gefolgert werden, weil wir bei der Definition der Semantik eines Arguments, außer an den abstrakten Befehlszähler, keine Anforderungen an dynamische Funktionen der entsprechenden ASM stellen.

Satz 6.14 (Simulation eines generischen Sprachkonzepts)
Ist $\mathsf{S}(\mathsf{s}_1, \ldots, \mathsf{s}_k)$ *eine wohlgeformte Instanz von* $\mathsf{S}(S_1 < \mathcal{A}_1, \ldots, S_k < \mathcal{A}_k)$*, dann gilt:*

$$[\![\, \mathsf{S} \,]\!]([\![\, \mathsf{s}_1 \,]\!], \ldots, [\![\, \mathsf{s}_k \,]\!]) \prec_{\overline{\rho}} [\![\, \mathsf{S} \,]\!](\mathcal{A}_1, \ldots, \mathcal{A}_k)$$

Beweis Die Semantik $[\![\, \mathsf{S} \,]\!](\mathcal{A}_1, \ldots, \mathcal{A}_k)$ ist, wie in Abschnitt 6.3 beschrieben, strukturell definiert, wobei die Minimalanforderungen $\mathcal{A}_i$ selbst als Argumente eingesetzt werden. Da $\mathsf{S}(\mathsf{s}_1, \ldots \mathsf{s}_k)$ eine wohlgeformte Instanz ist, garantiert Anforderung (A), daß die durch $\overline{\rho}_i$ angepaßte ASM genau die Schnittstelle der Minimalanforderung erfüllt und daher eingesetzt werden kann. Die Parameter eines generischen Sprachkonzepts sind über Steuerfluß und Datenabhängigkeiten in die Gesamt-ASM eingebunden. Das heißt, an einem bestimmten Punkt wird eine Teil-ASM $[\![\, \mathsf{s}_i \,]\!]$ „aufgerufen“ ($ct \in [\![\, \mathsf{s}_i.\mathit{initial} \,]\!]$) und nach Beendigung der Ausführung steht das Ergebnis in $[\![\, \mathit{value}(\mathsf{s}_i.\mathit{terminal}) \,]\!]$. Da $[\![\, \mathsf{s}_i \,]\!] \prec_{\overline{\rho}_i} \mathcal{A}_i$ für alle Teilkonzepte gilt und die Interaktion mit der Umgebung wegen (C) keinen Einfluß auf die Berechnung hat, wird dieses Ergebnis auch von $\mathcal{A}_i$ berechnet.

Formal beweisen wir die Aussage durch Induktion über die Anzahl der Zustandsübergänge. Für die Funktion $\overline{\rho}$, die Zustände von $[\![\, \mathsf{S} \,]\!](\mathcal{A}_1, \ldots, \mathcal{A}_k)$ und $[\![\, \mathsf{S} \,]\!]([\![\, \mathsf{s}_1 \,]\!], \ldots, [\![\, \mathsf{s}_k \,]\!])$ in Beziehung setzt, gilt $\overline{\rho} \supseteq \overline{\rho}_1 \cup \ldots \cup \overline{\rho}_k$

Induktionsanfang: $n = 0$
Die beiden Maschinen unterscheiden sich in den Teilen des Zustands, die von den $\mathcal{A}_i$ bzw. $[\![\, \mathsf{s}_i \,]\!]$ definiert werden. Die Funktion $\overline{\rho}$ wurde unter Ausnutzung der $\overline{\rho}_i$ so konstruiert, daß sie genau die unterschiedlichen Teile in Beziehung setzt.

Induktionshypothese: Die Behauptung gilt für Zustandsfolgen der Länge n.

Induktionsschluß: $n \curvearrowright n+1$
Wir müssen zwei Fälle unterscheiden. Entweder wird ein Zustandsübergang durch den Rahmen S oder durch die einem Parameter entsprechende Zustandsmaschine verursacht.

1. Bei einem Übergang, bei dem eine Aktualisierungsregel R des Rahmens gefeuert wird, haben wir

   $$\begin{array}{ccc} q & \xrightarrow{R} & \overline{q} \\ {\scriptstyle\overline{\rho}}\uparrow & & \uparrow{\scriptstyle\overline{\rho}} \\ q' & \xrightarrow{R} & \overline{q'} \end{array}$$

   zu zeigen. Stehen die Zustände q und q' in Beziehung, dann werden auf der Quell- und der Zielmaschine exakt die gleichen Aktualisierungen ausgeführt. Die Minimalanforderungen von S machen alle Auswirkungen auf den Kontext der Parameter beobachtbar. Daher garantiert die Funktion $\overline{\rho}$, daß die Funktionen, die für die Aktualisierungen in R relevant sind, auch die gleiche Interpretation haben. Das gilt natürlich auch nach Ausführen derselben Regel R auf der Quell- und der Zielmaschine und daher folgt $\overline{\rho}(\overline{q'}) = \overline{q}$.

2. Wird die ASM eines Parameters ausgeführt, dann ist

$$\begin{array}{ccc} q & \xrightarrow{\mathcal{A}_i} & \overline{q} \\ \overline{\rho}\uparrow & & \uparrow\overline{\rho} \\ q' & \xrightarrow{[\![\,\mathsf{s}_i\,]\!]} & \overline{q'} \end{array}$$

zu zeigen. Laut Voraussetzung gilt $[\![\,\mathsf{s}_i\,]\!] \prec_{\overline{\rho}_i} \mathcal{A}_i$. Der konkrete Parameter simuliert das beobachtbare Verhalten der Minimalanforderung und speziell auch alle Auswirkungen, die $\mathcal{A}_i$ auf den Kontext hat. Damit folgt sofort $\overline{\rho}(\overline{q'}) = \overline{q}$.

♢

Die Definition von Wohlgeformtheit können wir auf komplette Sprachspezifikationen übertragen.

Definition 6.15 (Wohlgeformte Sprachdefinition)
Eine Sprachdefinition heißt wohlgeformt, wenn alle ihre Sprachkonzepte (beschrieben durch die Nichtterminale und Terminale der abstrakten Syntax) wohlgeformt sind.

6.4.2 Korrekte Übersetzung von Sprachkonzepten mit Minimalanforderungen

Sind Transformationen strukturerhaltend, d. h. baut ein Zielsprachkonzept auf Teilkonzepten auf, welche die gleichen (modulo Simulation) Minimalanforderungen erfüllen wie die Teilkonzepte des Quellkonzepts, dann können wir Übersetzungen generisch formulieren und lokal die Transformation eines Sprachkonzepts verifizieren. Wir setzen dabei nur die korrekte Abbildung der Teilkonzepte voraus. Letztendlich führt das zu einem Korrektheitsnachweis durch strukturelle Induktion.

Definition 6.16 (Korrekte strukturerhaltende Übersetzung)
$\mathcal{C}(\mathsf{S}(S_1 < \mathcal{A}_1, \ldots, S_k < \mathcal{A}_k)) = \mathsf{S}'(S'_1 < \mathcal{A}'_1, \ldots, S'_k < \mathcal{A}'_k)$ *ist eine* ***korrekte strukturerhaltende*** *Übersetzung, wenn gilt:*

$$\forall i, 1 \leq i \leq k : \mathcal{A}'_i \prec_{\overline{\rho}} \mathcal{A}_i \tag{6.3}$$

$$[\![\,\mathsf{S}'\,]\!](\mathcal{A}'_1, \ldots, \mathcal{A}'_k) \prec_{\overline{\rho}} [\![\,\mathsf{S}\,]\!](\mathcal{A}_1, \ldots, \mathcal{A}_k) \tag{6.4}$$

Die Beziehung zwischen den Minimalanforderungen der Quelle und des Ziels entspricht der Intuition, daß die Semantik des übersetzten Programms nicht allgemeiner sein kann als die Semantik der Quelle. In unseren Anwendungen sind die Minimalanforderungen des Quell- und des Zielsprachkonzepts üblicherweise identisch oder bisimulieren sich, falls die zugrundeliegenden Maschinenmodelle von S und S' unterschiedlich sind.

Bei der Übersetzung einer Sprache sind strukturerhaltende Übersetzungen von Vorteil, da dann die Verifikation über strukturelle Induktion geführt werden kann. Eine spezielle Form strukturerhaltender Übersetzungen stellen bestimmte Optimierungen dar, die Parameter eines generischen Sprachkonzepts eliminieren. In diesem Fall hat man nachzuweisen, daß der eliminierte Parameter, durch die leere Anweisung simuliert werden kann, also keine Zustandsänderung bewirkt. Ein Beispiel einer solchen Optimierung ist die Elimination toten Codes. Allerdings

kann nicht immer strukturerhaltend übersetzt werden, z. B. wenn Optimierungen eine Umordnung des Codes bewirken. In diesem Fall läßt sich die Übersetzung eines Sprachkonzepts nicht mehr induktiv über der Übersetzung der Teilkonzepte definieren und die Verifikation muß über der kompletten Semantik des Sprachkonzepts durchgeführt werden. Allgemein gilt, daß Übersetzungen, die nicht strukturerhaltend sind, mehrere induktive Übersetzungsschritte zusammenfassen und daher komplett verifiziert werden müssen.

6.4.3 Allgemeine Korrektheitsaussagen

Im folgenden zeigen wir, wie die Benutzung generischer Sprachkonzepte die Verifikation von Übersetzungen vereinfacht. Wir wollen die Korrektheit einer Übersetzung $\mathcal{C}$ mit $\mathcal{C}(\mathsf{S}(\mathsf{s}_1, \ldots, \mathsf{s}_k)) = \mathsf{S}'(\mathsf{s}'_1, \ldots, \mathsf{s}'_k)$ nachweisen. In unserem Rahmen können wir dabei die folgenden Annahmen machen.

$$\mathsf{S}(\mathsf{s}_1, \ldots, \mathsf{s}_k) \text{ ist eine wohlgeformte Instanz von } \mathsf{S}(S_1 < \mathcal{A}_1, \ldots, S_k < \mathcal{A}_k) \tag{6.5}$$

$$\mathsf{S}'(\mathsf{s}'_1, \ldots, \mathsf{s}'_k) \text{ ist eine wohlgeformte Instanz von } \mathsf{S}'(S_1 < \mathcal{A}'_1, \ldots, S_k < \mathcal{A}'_k) \tag{6.6}$$

$$\mathsf{S}'(S_1 < \mathcal{A}'_1, \ldots, S_k < \mathcal{A}'_k) \text{ ist eine korrekte strukturerhaltende Übersetzung von } \mathsf{S}(S_1 < \mathcal{A}_1, \ldots, S_k < \mathcal{A}_k) \tag{6.7}$$

$$\forall i, 1 \leq i \leq k : [\![s'_i]\!] \prec_{\overline{\rho}} [\![s_i]\!] \tag{6.8}$$

(6.5) und (6.6) gelten immer, da wir nur an der Übersetzung wohlgeformter Quell- in wohlgeformte Zielsprachen interessiert sind. (6.7) muß gelten, wenn wir generische Sprachkonzepte für die Übersetzung ausnutzen wollen. (6.8) ist die Induktionshypothese, wenn wir die Übersetzung einer kompletten Sprache betrachten und daher auf jeden Fall nachweisen müssen, daß alle Teilkonzepte korrekt übersetzt werden.

Für den Nachweis der Korrektheit zerlegen wir die Übersetzung $\mathcal{C}$ in $\mathcal{C}_2 \circ \mathcal{C}_1$.

$$\mathsf{S}(\mathsf{s}_1, \ldots, \mathsf{s}_k) \xrightarrow{\mathcal{C}_1} \mathsf{S}(\mathsf{s}'_1, \ldots, \mathsf{s}_k)' \xrightarrow{\mathcal{C}_2} \mathsf{S}'(\mathsf{s}'_1, \ldots, \mathsf{s}'_k)$$

Wir übersetzen also zuerst die Teilkonzepte s_i und dann das Sprachkonzept S. Diese Zerlegung impliziert die Einführung einer neuen logischen Zwischensprache zu Beweiszwecken. Haben wir die Korrektheit von $\mathcal{C}_1$ und $\mathcal{C}_2$ nachgewiesen, dann folgt die Korrektheit von $\mathcal{C}$ aus dem Satz 2.2, S. 37 über die vertikale Komposition von korrekten Teiltransformationen.

Wir beweisen zuerst einige Hilfssätze.

Satz 6.17 (Transitive Fortsetzung von Instantiierungen)
Sind $\mathsf{S}(\mathsf{s}_1, \ldots, \mathsf{s}_k)$ *und* $\mathsf{S}(\mathsf{s}'_1, \ldots, \mathsf{s}'_k)$ *wohlgeformte Instanzen eines generischen Sprachkonzepts* $\mathsf{S}(S_1 < \mathcal{A}_1, \ldots, S_k < \mathcal{A}_k)$ *und gilt (6.8), dann folgt daraus:*

$$[\![\mathsf{S}]\!]([\![\mathsf{s}'_1]\!], \ldots, [\![\mathsf{s}'_k]\!] \prec_{\overline{\rho}} [\![\mathsf{S}]\!]([\![\mathsf{s}_1]\!], \ldots, [\![\mathsf{s}_k]\!]$$

Beweis Da beide Instantiierungen wohlgeformt sind, erfüllen sowohl die s_i als auch die s'_i die Anforderungen, die vom Kontext S an die Teilkonzepte gestellt werden. Zusätzlich simulieren die $[\![\mathsf{s}'_i]\!]$ das beobachtbare Verhalten der $[\![\mathsf{s}_i]\!]$. Da sich die $[\![\mathsf{s}'_i]\!]$ sowohl lokal als auch im Kontext, wie die $[\![\mathsf{s}_i]\!]$ verhalten, folgt die Behauptung.

Formal würde man den Beweis analog zu Satz 6.14 durch Induktion über die Anzahl der Zustandsübergänge von $[\![\mathsf{S}]\!]([\![\mathsf{s}'_1]\!], \ldots, [\![\mathsf{s}'_k]\!])$ führen. ♢

Der folgende Satz etabliert die Korrektheit des zweiten Übersetzungsschritts $\mathcal{C}_2$.

Satz 6.18 (Instantiierung einer strukturerhaltenden Übersetzung)
Ist $\mathsf{S}'(S'_1 < \mathcal{A}'_1, \ldots, S'_k < \mathcal{A}'_k)$ *eine korrekte strukturerhaltende Übersetzung von* $\mathsf{S}(S_1 < \mathcal{A}_1, \ldots, S_k < \mathcal{A}_k)$ *und* $\mathsf{S}'(\mathsf{s}'_1, \ldots, \mathsf{s}'_k)$ *eine wohlgeformte Instanz von* $\mathsf{S}'(S_1 < \mathcal{A}'_1, \ldots, S_k < \mathcal{A}'_k)$, *dann gilt:*

$$[\![\,\mathsf{S}'\,]\!]([\![\,\mathsf{s}'_1\,]\!], \ldots, [\![\,\mathsf{s}_k\,]\!]') \prec_{\overline{\rho}} [\![\,\mathsf{S}\,]\!]([\![\,\mathsf{s}'_1\,]\!], \ldots, [\![\,\mathsf{s}'_k\,]\!]).$$

Beweis Wegen $[\![\,\mathsf{s}_i\,]\!] \prec_{\overline{\rho}_{i'}} \mathcal{A}'_i \prec_{\overline{\rho}_i} \mathcal{A}_i$ folgt mit der Wohlgeformtheit von $\mathsf{S}'(\mathsf{s}'_1, \ldots, \mathsf{s}'_k)$ auch die Wohlgeformtheit von $\mathsf{S}(\mathsf{s}'_1, \ldots, \mathsf{s}_k)$. Die Simulationsbeziehung $[\![\,\mathsf{S}'\,]\!](\mathcal{A}'_1, \ldots, \mathcal{A}'_i) \prec_{\overline{\rho}} [\![\,\mathsf{S}\,]\!](\mathcal{A}_1, \ldots, \mathcal{A}_i)$ wurde unter Annahme der Minimaleigenschaften $\mathcal{A}_i$ bzw. $\mathcal{A}'_i$ gezeigt. Mit der Transitivität der $\overline{\rho}$-Simulation folgt sofort die Behauptung. ♢

Der nächste Satz etabliert die Korrektheit des kompletten Übersetzungsschritts $\mathcal{C} = \mathcal{C}_2 \circ \mathcal{C}_1$.

Satz 6.19 (Korrekte Übersetzung mit generischen Sprachkonzepten)
Für eine Übersetzung $\mathcal{C}$ *mit* $\mathcal{C}(\mathsf{S}(\mathsf{s}_1, \ldots, \mathsf{s}_k) = \mathsf{S}'(\mathsf{s}'_1, \ldots, \mathsf{s}'_k)$ *gilt mit den Bedingungen (6.5)–(6.8)*

$$[\![\,\mathsf{S}'\,]\!]([\![\,\mathsf{s}'_1\,]\!], \ldots, [\![\,\mathsf{s}'_k\,]\!]) \prec_{\overline{\rho}} [\![\,\mathsf{S}\,]\!]([\![\,\mathsf{s}_1\,]\!], \ldots, [\![\,\mathsf{s}_k\,]\!]).$$

Beweis $\mathsf{S}(\mathsf{s}_1, \ldots, \mathsf{s}_k)$ ist wohlgeformt und wegen (6.6) und (6.7) gilt $[\![\,\mathsf{s}'_i\,]\!] \prec_{\overline{\rho}} \mathcal{A}_i$ für alle $i, 1 \leq i \leq k$. Deshalb ist auch $[\![\,\mathsf{S}\,]\!]([\![\,\mathsf{s}'_1\,]\!], \ldots, [\![\,\mathsf{s}'_k\,]\!]$ wohlgeformt. Zusammen mit (6.8) ist dann Satz 6.17 anwendbar und $\mathcal{C}_1$ ist korrekt. (6.6) und (6.7) sind die Voraussetzungen für die Anwendung von Satz 6.18. Damit ist auch $\mathcal{C}_2$ korrekt und die Behauptung folgt durch die vertikale Komposition von korrekten Abbildungen. ♢

Gehen wir davon aus, daß für die Konstruktion eines korrekten Übersetzers eine Bibliothek vordefinierter generischer Sprachkonzepte und korrekter Transformationen dieser Konzepte verwendet werden, dann muß der Konstrukteur eines Übersetzers nur noch die Wohlgeformtheit der beteiligten Sprachen zeigen.

6.5 Zusammenfassung

In diesem Kapitel haben wir formal definiert, was korrekte Übersetzung von Programmen bedeutet, wenn die Semantik der Quell- und der Zielsprache operationell beschrieben ist. Eine Übersetzung $\mathcal{C}(\pi)$ eines Programms π ist demnach korrekt, wenn für alle zulässigen Eingaben das beobachtbare Verhalten des Quellprogramms (in unserem Kontext das Ein-/Ausgabeverhalten) durch die Semantik des Zielprogramms simuliert wird.

Für die Verifikation einer Übersetzung haben wir ein induktives Beweisprinzip über den Aufbau von Programmen aufgestellt. Dabei nutzen wir aus, daß die operationelle Semantik von Programmen mit unserem Dialekt von Montages kompositionell definiert ist. Fordern wir zusätzlich noch, daß bei der Definition eines Sprachkonzepts explizit die Minimalanforderungen an seine Teilkonzepte angegeben werden, dann kann über die Übersetzung des Sprachkonzepts lokal argumentiert werden, ohne die Teilkonzepte betrachten zu müssen. Wir weisen die Korrektheit der Übersetzung eines Sprachkonzepts unter der Annahme nach, daß die Teilkonzepte bestimmte Bedingungen erfüllen und ebenfalls korrekt übersetzt werden. In der konkreten Anwendung nutzen wir die im vorigen Abschnitt definierten Sätze 6.17 und 6.18. Die korrekte Übersetzung einer Sprache leitet sich aus dem Nachweis der korrekten Übersetzung aller Sprachkonzepte ab, falls die Quell- und die Zielsprache wohlgeformt sind.

Durch die Verwendung generischer Sprachkonzepte ist es erstmals möglich, diese Aussagen über die Wohlgeformtheit einer Sprachspezifikation zu machen. Bei einer wohlgeformten Sprachspezifikation müssen alle Instanzen generischer Sprachkonzepte die geforderten Mindestanforderungen an die Teilkonzepte erfüllen.

Generische Sprachkonzepte mit Minimalanforderungen sind die Grundlage für den Aufbau einer Bibliothek von Sprachkonzepten und korrekten Übersetzungen. Die Wiederverwendung vordefinierter Komponenten vereinfacht die Konstruktion einer korrekten Übersetzungsspezifikation, weil für die Konzepte einer Bibliothek die Voraussetzungen zur Anwendung der Übersetzungstheoreme schon nachgewiesen wurden. Stammt das Quellsprachkonzept $\mathsf{S}(S_1 < \mathcal{A}_1, \ldots, S_k < \mathcal{A}_k)$, das Zielsprachkonzept $\mathsf{S}'(S'_1 < \mathcal{A}'_1, \ldots, S'_k < \mathcal{A}'_k)$ und die Transformation $\mathcal{C}$ mit $C(\mathsf{S}) = \mathsf{S}'$ aus einer verifizierten Bibliothek, dann gewährleistet schon die Wohlgeformtheit der Instantiierungen $\mathsf{S}(\mathsf{s}_1, \ldots, \mathsf{s}_k)$ und $\mathsf{S}'(\mathsf{s}'_1, \ldots, \mathsf{s}'_k)$ die Simulation.

7 Benutzung der Sprache *AL* zur Verifikation von Übersetzern

Mit den im vorigen Kapitel eingeführten Techniken können wir zwar den Korrektheitsnachweis für Übersetzungen modularisieren und Teile wiederverwenden, jedoch müssen wir für jedes Quellsprach-/Zielsprachpaar den Nachweis erneut führen. Untersucht man Quell- und Zielsprachsemantiken und Übersetzungen zwischen den Sprachen, dann stellt man fest, daß die bei der Übersetzung auftretenden abstrakten Zwischensprachen jeweils ähnliche Eigenschaften haben. Sie beschreiben nämlich genau die Semantik der zu übersetzenden Sprachen in Termen von bestimmten Kernkonzepten. Die Identifikation dieser Konzepte führte zur Definition der Sprache *AL*. In Abschnitt 3.4 haben wir schon dargelegt, wie *AL* in unserem Übersetzerrahmen eingesetzt wird. Wir benutzen *AL* zur Definition der dynamischen Semantik imperativer Programmiersprachen und auch als Zwischensprache, die in Übersetzern auftritt. Verwendet man *AL* als konkrete Zwischensprache im Übersetzer, dann reduziert das die Anzahl der Übersetzer von $n * m$ auf $n + m$, wenn n Quellsprachen QS auf m Zwischenrepräsentationen ZR abgebildet werden, da entweder die Abbildung $QS \rightarrow AL$ oder $AL \rightarrow ZR$ wiederverwendet werden kann.

AL ist zwar insofern eine universelle Sprache zur Spezifikation dynamischer Semantik, daß alle imperativen Sprachen damit beschrieben werden können, aber komfortabel kann damit nur bedingt spezifiziert werden. *AL* unterstützt weder Konzepte, die in Quellsprachen zur einfachen Umsetzung des Entwurfs eines Systems eingeführt wurden, noch berücksichtigt sie spezielle Zielmaschinenkonzepte. Aus diesem Grund setzen wir eine Bibliothek ein, die sowohl die Spezifikation von Sprachkonzepten als auch die Konstruktion einer korrekten Übersetzung durch vordefinierte Sprachkonzepte und Übersetzungen dieser Sprachkonzepte unterstützt. Zu der Bibliothek gehören auch die Sprachkonzepte aus *AL* und Übersetzungen von *AL* in unterschiedliche Zwischensprachen.

Für die Konstruktion einer korrekten Übersetzungsspezifikation gehen wir wie folgt vor. Wir geben für die Quellsprache QS eine Montages-Spezifikation an und definieren damit eine formale Semantik dieser Sprache. Die Semantik der Quellsprachkonzepte wird mit den vordefinierten Sprachkonzepten der Bibliothek ausgedrückt. Faßt man die bei der Spezifikation verwendeten Sprachkonzepte zusammen, dann definiert das eine spezielle Spezifikationssprache AL^+ zur Beschreibung der dynamischen Semantik der Quellsprache QS. Die Semantik von AL^+ ist durch einen Interpretierer in Form einer abstrakten Zustandsmaschine definiert. Anschließend konstruieren wir eine Übersetzung von AL^+-Programmen in die gewünschte Zwischensprache ZS. Diese Übersetzung ist aus zwei Übersetzungsschritten aufgebaut. Für die Übersetzung $QS \rightarrow AL$ nutzen wir aus, daß die Sprache AL^+ selbst aus Bibliothekskonzepten bzw. aus den Sprachkonzepten der Kernsprache *AL* zusammengesteckt wurde und wir daher die Übersetzung aus vordefinierten und korrekt bewiesenen Transformationen ableiten können. Für den zweiten Schritt benutzen wir ebenfalls vordefinierte Übersetzungen, die Konzepte aus *AL* durch Konzepte einer bestimmten Zwischen- bzw. Zielsprache implementieren. Auch bei dieser Übersetzung können Zwischenschritte auftreten, die neue Zwischensprachen AL^- einführen. Die Abbildung 7.1 illustriert noch einmal das Vorgehen bei der Übersetzung.

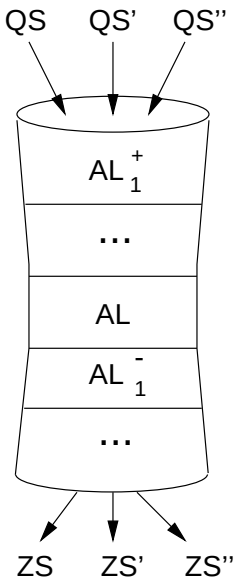


Abbildung 7.1: Übersetzung unter Verwendung von Standardzwischensprachen

Im folgenden stellen wir die einzelnen Schritte unseres Vorgehens und die Rolle von AL im Detail vor. Dabei berücksichtigen wir auch, daß für manche Quellsprachen neue Sprachkonzepte oder Übersetzungen, ohne Unterstützung der Bibliothek, angegeben werden müssen. Diese Übersetzungen müssen dann natürlich verifiziert werden.

7.1 Semantikspezifikation mit *AL* und die Definition von Übersetzungen

Die Montages-Spezifikation einer Quellsprache definiert für eine Produktion der abstrakten Syntax bzw. für ein Sprachkonzept der Quellsprache eine Abbildung auf die abstrakte Syntax unserer Bibliothek. Die Montages-Spezifikation definiert auch die Attribute, die zur Definition der dynamischen Semantik der verwendeten AL^+-Konzepte oder für die Transformation benötigt werden. Die zweite Komponente einer Montage ordnet den entsprechenden abstrakten Teilbäumen AL^+-Graphen zu, die die dynamische Semantik beschreiben. Die Semantik der Kommandos, die in dieser zweiten Komponente angegeben sind, muß bekannt sein, d. h. es dürfen nur Kommandos aus AL^+ verwendet werden. Am Beispiel der Zuweisung in Abbildung 7.2 demonstrieren wir das Prinzip. Die abstrakte Syntax der Quellsprache definiert einen Designator für die linke Seite und einen Ausdruck für die rechte Seite der Zuweisung. Die abstrakte Syntax der Quellsprachzuweisung ist

$QS\_Assign ::= QS\_Designator\ QS\_Expr.$

Der entsprechende sprachunabhängige AL-Graph definiert einen AL_Assign-Knoten, der die sprachunabhängige Repräsentation von QS_Designator und QS_Expr zur Berechnung der Quelle und des Ziels der Zuweisung benutzt. Die Attribute *type, lval* und *defs* beschreiben statische semantische Eigenschaften. Die Bedingungen sichern zu, daß der Designator auch tatsächlich ein Objekt identifiziert und der Wert des Ausdrucks auch an das Objekt zugewiesen werden kann. Die dynamische Semantik von AL_Assign muß in dieser Montage nicht angegeben werden, da AL_Assign durch die Bibliothek schon eine feste Semantik hat. Alle Montages

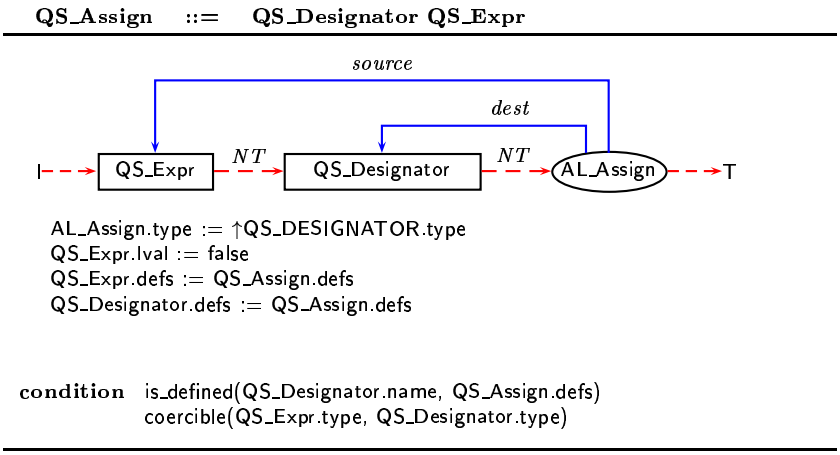


Abbildung 7.2: Eine Montage für die Zuweisung

der Sprachspezifikation zusammen definieren eine Abbildung von Quellsprachprogrammen in AL^+-Graphen.

Bemerkung: Die obige Beschreibung resultiert aus einer Sicht auf die Semantik. In Wahrheit sind die AL^+-Graphen, die ein Programm semantisch repräsentieren, nichts anderes als sprachunabhängige attributierte Strukturbäume. Die Kanten des Graphen sind durch Attribute codiert. Die Abbildung 7.3 zeigt ein Beispiel. ♢

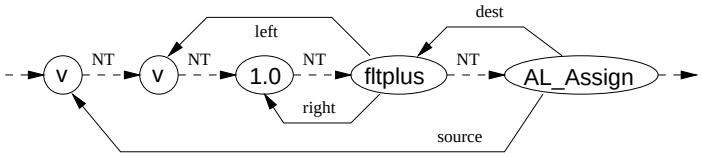


Abbildung 7.3: AL-Graph zu der Zuweisung $v := v + 1.0$

Die Semantik von Sprachkonzepten aus AL^+, die nicht aus dem Kern AL sind, wird durch AL^+-Konzepte bzw. direkt durch AL beschrieben. Daher existiert für jedes Konzept aus AL^+ eine Abbildung (direkt oder über einen Umweg in AL^+) nach AL, die letztendlich die Semantik dieses Konzepts spezifiziert. Diese Abbildung können wir natürlich auch als Spezifikation einer Übersetzung ansehen und können damit aus der Sprachspezifikation einer Quellsprache QS eine komplette und korrekte Transformation $QS \rightarrow AL^+ \rightarrow AL$ ableiten, sofern die Abbildungen $AL^+ \rightarrow AL$ verifiziert ist. Anschließend müssen wir AL noch in die gewünschte Zwischensprache übersetzen.

Fakt 7.1 Die Übersetzung der Sprache AL in eine Zwischensprache ist korrekt und vollständig, wenn für *jede* Instruktion aus AL eine semantikerhaltende Abbildung in die Zwischenrepräsentation existiert.

AL Programme sind Graphen mit einem ausgezeichneten Knoten, der das erste auszuführende Kommando beschreibt. Folgen wir bei der Übersetzung den Steuerflußkanten, wobei wir schon besuchte und übersetzte Knoten nicht weiterbearbeiten, dann definiert das eine Ersetzungsstrategie, die terminiert, wenn alle Knoten und Kanten abgebildet sind. Diese Strategie wenden wir auf jede schwache Zusammenhangskomponente des AL Graphen an. Wir haben schwache

Zusammenhangskomponenten vorliegen, weil Prozeduren durch Teilgraphen repräsentiert sind, deren Anfangsadresse bzw. erstes auszuführende Kommando über einer Tabelle verwaltet wird.

Die mit der Minimalanforderung für Vollständigkeit und Korrektheit ableitbare Übersetzung nutzt nicht alle Zwischensprachinstruktionen, falls diese von höherem Abstraktionsniveau, als die *AL* Instruktionen sind. Für die Konstruktion eines realistischen Übersetzers ist das jedoch nicht akzeptabel. Mit Kenntnis der Abbildung der Quellsprache weiß man jedoch, daß nur bestimmte AL Programme vorkommen können, für die man eine Übersetzung definieren muß. Das können wir ausnutzen, um eine bessere Übersetzung anzugeben, die nicht den Umweg über *AL* geht, sondern AL^+-Konzepte direkt in die Zwischensprache übersetzt, vgl. Abschnitt 3.4.2. Damit sind wir auch in der Lage, Befehle der Zwischensprache, die speziellen und besonders effizient ausführbaren Befehle des Prozessors entsprechen, auszunutzen.

In unserem Übersetzungsrahmen haben wir daher zwei unterschiedliche Arten von Regeln. Einerseits Regeln, die die Semantik neuer komplexer Sprachkonzepte definieren und andererseits Regeln, die explizit Übersetzungen definieren. Eine Montage definiert Regeln der ersten Art. Eine Programmtransformation der zweiten Klasse wird durch eine oder mehrere bedingte Graphersetzungsregeln angegeben, die eine Übersetzung des Steuer- und Datenflußgraphen über Kommandos definieren.

7.2 Die Definition neuer Sprachkonzepte

Bei der Spezifikation von Quellsprachkonzepten müßte man sich mit vielen Zwischensprach- bzw. Zielsprachdetails befassen. Die Erweiterungen der Sprache *AL* verbergen Spezifikations- bzw. Implementierungsdetails und erleichtern damit die Spezifikation von Semantik. Die Erweiterungen beschreiben die Semantik komplexer Konzepte imperativer Programmiersprachen in Termen von *AL*. Jedes neu eingeführte semantische Konzept wird in die Bibliothek aufgenommen und kann anschließend zur Spezifikation von Semantik benutzt werden. Welche Konzepte in der Bibliothek benötigt werden, ergibt sich aus unserer Analyse der Transformationsphase und den Eigenschaften von imperativen Quellsprachen (siehe Kapitel 2, Tabelle 2.1).

In den nächsten Abschnitten geben wir unterschiedliche semantische Konzepte von Quellsprachen an, für die wir Konzepte in der Bibliothek definieren. Hierbei dürfen wir nicht vergessen, daß jede Abstraktion bzw. Erweiterung, die wir im folgenden einführen, einer Übersetzung entspricht, die in der Transformationsphase ein höheres Konzept durch maschinennähere Konzepte implementiert. Diese Übersetzung ist durch die Abbildung nach *AL* definiert und ebenfalls Teil der Bibliothek. Wir setzen voraus, daß alle für die Übersetzung benötigte Information schon bei der Spezifikation der Quellsprache angegeben wird und daher im Programmgraphen explizit repräsentiert ist. Eine Ausnahme sind Informationen für Optimierungen, die in einem separaten Analyseschritt berechnet werden. Die Abbildung 7.4 zeigt die Konzepte von *AL* und AL^+ im Überblick.

7.2.1 Daten und Speicher

In realen Übersetzern werden bei der Speicherabbildung komplexe Datentypen auf die Grunddatentypen der Zwischensprache bzw. der Maschine abgebildet und der Zugriff auf Objekte wird nicht mehr über Namen, sondern durch symbolische oder Relativadressen implementiert. Dabei fließen unter Umständen schon Informationen über die Zielmaschine ein. Dieser letzte Punkt betrifft die Erweiterungen von *AL* zunächst nicht, da *AL* eine maschinenunabhängige

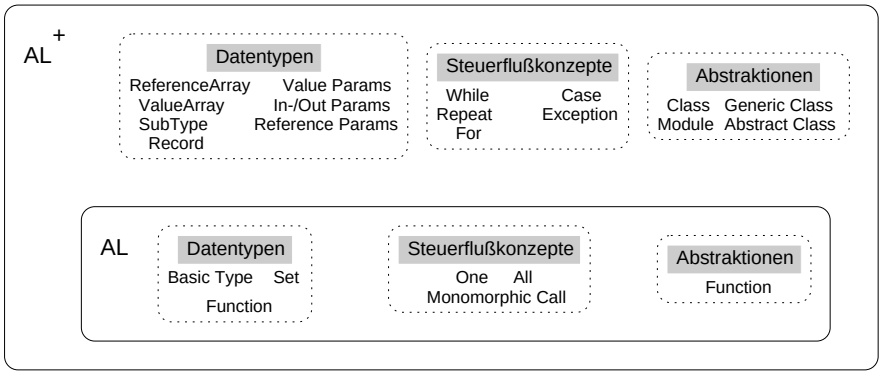


Abbildung 7.4: Sprachkonzepte von AL und AL^+

Beschreibung der dynamischen Semantik von Quellsprachen darstellt und über symbolische Adressen auf Objekte zugegriffen wird.

Komplexere Objekte, wie Reihungen, Verbunde oder Klassen, sind (induktiv) aus weniger komplexen Objekten aufgebaut. Diese Struktur ist ein Aspekt des Typsystems einer Programmiersprache. Zur Beschreibung der statischen Semantik eines neuen Typs führen wir einen neuen Konstruktor der Sorte Type ein. Die dynamische Semantik wird durch neue Kommandos angegeben, die beschreiben, wie neue strukturierte Objekte eines Typs erzeugt werden (New), wie die Werte eines Objekts gelesen (Load) oder verändert (Store) werden. Diese Grundfunktionalität gilt für alle Objekte, unabhängig von ihrem Typ.

In diesem speziellen Fall unterscheidet sich die Semantikspezifikation nicht von der Spezifikation der Transformation, die im Übersetzer durchgeführt wird. Bei der Übersetzung von strukturierten Objekten werden komplexe Typen auf die Typen der Zwischensprache abgebildet und die Funktionalität auf Objekten wird durch Instruktionen über Basistypen implementiert.

Beispiel 7.1 Eine Reihung $Array[size](T)$ modellieren wir durch einen Mengentyp $set((T', 0), \ldots, (T', size-1))$. T' ist gleich T, falls es sich um einen Typ von AL handelt, ansonsten ist T' die entsprechende AL-Modellierung. Die Zuweisung an eine Reihung beschreibt das Kopieren aller Elemente der Reihung, falls es sich um eine Wertesemantik handelt. Ansonsten wird die entsprechende Referenz zugewiesen. Der Zugriff auf die Elemente der Reihung wird unter Verwendung der AL-Konzepte zum Zugriff auf die Elemente einer Menge beschrieben.

♢

7.2.2 Steuerfluß- und Entwurfskonzepte

Die Kernsprache AL definiert als Steuerflußkonzepte neben der Hintereinanderausführung nur eine bedingte Verzweigung, einen einfachen Funktionsaufruf und die indeterministische Ausführungsreihenfolge. Übliche imperative Quellsprachen definieren *for*-, *while*- und *repeat*-Schleifen, eine *Case*-Anweisung und unterschiedliche Arten der Ausnahmebehandlung. Zusätzlich gibt es in imperativen Sprachen unterschiedliche Parameterübergabemechanismen beim Funktionsaufruf, die in AL so nicht vorhanden sind. In der Transformationsphase werden diese komplexen Konstrukte auf die einfacheren Konstrukte der Zielmaschine abgebildet.

Außerdem definieren Quellsprachen Konzepte, die die Konstruktion von Systemen oder die Wiederverwendung von Komponenten erleichtern. Solche Konzepte gibt es normalerweise auf

Zwischen- oder Maschinensprachebene nicht. Allerding haben Entwurfs- oder Abstraktionstechniken von Quellsprachen, wie z. B. die Definition von Modulen, Klassen oder polymorphen Typen und überladenen Operationen, bei genauerer Betrachtung vor allem Auswirkungen in der statischen Semantik.

Beispiel 7.2 Die dynamische Semantik eines monomorphen Funktionsaufrufs besagt, daß zum ersten Kommando der Funktion gesprungen wird, die durch den Funktionsnamen beschrieben ist. Die Zuordnung von Funktionsnamen zu Funktionstypen ist eine statische Information, die während der semantischen Analyse berechnet wird. Die dynamische Semantik eines polymorphen Funktionsaufrufs unterscheidet sich nicht vom monomorphen Aufruf. Es wird ebenfalls zum ersten Kommando der Funktion gesprungen. Was sich semantisch allerdings unterscheidet ist, daß beim polymorphen oder auch beim überladenen Aufruf die Zuordnung von Namen zu Funktionstypen keine Funktion, sondern eine Relation ist. Die Berechnung dieser Relation geschieht ebenfalls zur Übersetzungszeit. In der dynamischen Semantik wird dann nur angegeben, wie der konkrete Typ zu einem Funktionsnamen ausgewählt wird. ♢

Beispiel 7.3 Die Klassen einer objektorientierten Sprache stellen spezielle strukturierte Typen dar. In der dynamischen Semantik wird diese Typinformation, die wiederum statisch ist, zur Erzeugung und Veränderung der entsprechenden Objekte des Typs benutzt. ♢

Generische Klassen sind sogar ein Konzept, dessen Semantik in Sather (Goos, 1997) oder auch C++ (Stroustrup, 1986) syntaktisch erklärt wird. Man ersetzt einfach die Parameter einer Klasse bei der Instantiierung textuell und führt erst anschließend die semantische Analyse durch.

Die von der Quellsprache angebotenen Mechanismen zur Modularisierung von Programmen haben eine komplexere semantische Analyse zur Folge, weil die zu berechnende Typinformation komplexer ist und dadurch auch die Konsistenzprüfung von Programmen schwieriger wird. In der dynamischen Semantik bzw. in der Transformation wirken sich Abstraktionskonzepte meistens nur wenig aus, weil sie unter Verwendung von Typinformation einfach in Termen von Basiskonzepten beschrieben bzw. einfach durch Basiskonzepte implementiert werden können.

7.3 Die Übersetzung in die Zwischensprache

Bisher können wir aus der Definition der AL^+-Konzepte eine korrekte Übersetzung in *AL*-Programme herleiten. Die Übersetzung von *AL* in eine konkrete Zwischensprache haben wir noch nicht berücksichtigt. Außerdem können wir Konzepte der Zielmaschine nicht ausnutzen, falls diese komplexer sind als die Konzepte von *AL*. Diese Konzepte sind auf der Ebene von *AL* schon aufgelöst und durch *AL*-Graphen implementiert. Das Wiederfinden der entsprechenden Muster ist schwierig und kann, zumindest bisher, nicht als geeigneter Weg angesehen werden. Außerdem gehen in die Transformation keine speziellen Übersetzungen ein, die sich von der Semantikdefinition unterscheiden, da die Abbildung komplexer Konzepte bisher unter dem Gesichtspunkt der Semantikdefinition, nicht aber unter Übersetzerbaugesichtspunkten definiert wurde.

AL abstrahiert die Konzepte von Zwischensprachen. Die Abbildung von *AL* besteht vor allem aus

- der Spezialisierung von AL-Konzepten, z. B. mit speziellen Zuweisungsoperationen für die Basisdatentypen,
- der Speicherabbildung durch Einführung konkreter Adressen und
- der Abbildung von Steuerfluß, z. B. durch die Implementierung von Sprungsequenzen oder dem Zusammenfassen von Instruktionen zu Grundblöcken.

Jede einzelne dieser Abbildungen kann wiederverwendet werden und definiert Übersetzungen und Sprachkonzepte für unsere Bibliothek. Um diese neuen Sprachkonzepte von den höheren Konzepten zu unterscheiden benennen wir diesen Teil der Bibliothek mit AL^-.

Für die Erzeugung effizienten bzw. guten Zwischensprachcodes müssen wir zu einem Zeitpunkt in den Übersetzungsprozeß eingreifen, an dem wir die Zuordnung von Quell- auf effiziente Zielsprachkonzepte noch direkt erkennen können. Anstatt eine Übersetzung über den Umweg der Zwischensprache AL zu definieren, führen wir im Prinzip „abkürzende“ Übersetzungen ein, die Quellsprachkonzepte direkt in die passenden Zwischensprachkonzepte abbilden. Die Abbildung 3.6 veranschaulicht die Situation.

Beispiel 7.4 Die Quellsprache definiert eine Bit-Shift-Operation, die es genau so auf der Zielmaschine gibt. Da AL diese Operation nicht kennt, müßte man sie nachimplementieren. In diesem Fall würden wir Bit-Shift-Operationen direkt auf die entsprechende Operation der Zielmaschine abbilden. ♢

Abkürzende Transformationen definieren eine Optimierung der Übersetzung. Die Definition von optimierenden Transformationen führt dazu, daß es neben der Standardübersetzung, die wir in unserem Rahmen über den Weg mit AL ableiten können, zusätzliche Transformationen gibt. Die Definition zusätzlicher Übersetzungen macht jedoch nur Sinn, wenn es ein Kostenkriterium für Zwischensprachprogramme gibt, anhand dessen eine Auswahl unter mehreren anwendbaren Transformationsregeln getroffen werden kann. Da wir jedoch möglichst maschinenunabhängig sein wollen und die Berechnung genauer Kosten für ein Programm die Kenntnis der konkreten Zielmaschine erfordert, können wir nur eine Heuristik angeben. Wir nehmen an, daß eine geringere Codegröße günstiger ist. Für unsere Anwendung ist dieses Kriterium sinnvoll, da eine im Quellprogramm vorhandene Zielmaschinenoperation unverändert und ohne zusätzlichen Code abgebildet werden kann, während bei der Nachimplementierung der Operation in AL normalerweise mehr Code erzeugt wird. Diese Heuristik ist nicht für beliebige Optimierungen geeignet, da eine Aufblähung des Codes unter Umständen in einem anschließenden Optimierungsschritt zu einer kostengünstigeren Variante führt.

Bei der Übersetzung können wir unsere normale Übersetzungsstrategie beibehalten. Die Übersetzung beginnt mit dem ersten Kommando im Steuerfluß und pflanzt sich dann in Tiefensuche entlang der Steuerflußkanten fort. Falls eine optimierte Ersetzungsregel existiert, dann wird diese anstatt der Sprachdefinition angewendet. Alternativ, können wir auch mehrere Übersetzungen mit unterschiedlichen Kosten erlauben, von denen die kostengünstigste ausgewählt wird. Allerdings benötigen wir dann eine Kostenfunktion, die effizient berechnet werden kann. Die Auswahl erfolgt anhand der Kosten der Transformation bzw. des entstehenden Codes.

7.4 Die Verifikation von Übersetzungsregeln

In den letzten Abschnitten haben wir die Syntax bzw. Normalformen unseres Kalküls für Übersetzungen eingeführt und die Struktur zum Nachweis der korrekten Übersetzung einer kompletten Sprache beschrieben. Bei der Einführung eines neuen Sprachkonzepts wird jedoch immer auch eine Übersetzung in die Grundsprache AL definiert. Solche Übersetzungen definieren Regeln des Kalküls und müssen gegenüber einer formalen Semantik verifiziert oder zumindest validiert werden, falls nur eine informelle Semantik existiert. AL ist vom Abstraktionsgrad nahe an realen Zwischensprachen. Aus diesem Grund entsprechen die zu definierenden Abbildungen in die Sprache AL Transformationen, die normalerweise im Übersetzer durchgeführt werden. In unserem speziellen Fall sind diese Abbildungen auch Transformationen auf der Programmiersprachsemantik.

Im folgenden geben wir ein systematisches Vorgehen zur Verifikation von Übersetzungsregeln an und demonstrieren die grundlegenden Techniken am Beispiel. Die Beispieltransformationen sind für die Abbildung imperativer Programmiersprachen in die Sprache AL bzw. für die Abbildung von AL in eine Zwischensprache mit adressierbarem Speicher geeignet.

7.4.1 Ein Schema für Beweise

Alle Beweise in unserem Kalkül laufen nach dem gleichen Schema ab.

Sei die Semantik eines Quellprogramms durch eine ASM $\mathcal{A}_1 = (\Sigma_1, Alg(\Sigma_1), \longrightarrow_1, I_1)$ und die des zugehörigen Zielprogramms durch eine ASM $\mathcal{A}_2 = (\Sigma_2, Alg(\Sigma_2), \longrightarrow_2, I_2)$ definiert. Wir unterscheiden zwei Arten von Transformationen:

- Bei der **Speicherabbildung** wird der **Speicherzustand**, d. h. der Teil des Zustands einer ASM, der Speicherobjekte und Datentypen beschreibt, abgebildet. Wir drücken den Speicherzustand von Quellprogrammen in Termen der Zielsemantik aus und definieren damit eine Uminterpretation von Programmen. Die Kommandos und ihre Attribute, sowie der abstrakte Programmzähler bleiben unverändert.
- Bei der **Programmtransformation** bleibt der Speicherzustandsraum unberührt, aber das Programm wird verändert. Das bedeutet, wir transformieren Kommandos und Attribute über Kommandos sowie den abstrakten Befehlszähler.

Die Speicherabbildung definiert eine injektive Abbildung auf dem Speicherzustandsraum zusammen mit einer Änderung von $\longrightarrow_1$. Die Abbildung der Basisdatentypen wird durch eine Menge von Verfeinerungsregeln beschrieben. Diese Regeln induzieren, abhängig davon, ob von einem genaueren auf einen weniger genauen Datentyp oder umgekehrt abgebildet wird , einen Isomorphismus bzw. Epimorphismus auf den Algebren, die den Quell- und den Zieldatentyp spezifizieren. Zusammen definieren die Speicherabbildung und die Abbildung der Daten eine neue Interpretation von Quellprogrammen und wir haben nachzuweisen, daß für alle Programme die Interpretation im Speicherzustandsraum der Zielsprache die ursprüngliche Semantik im Speicherzustandsraum der Quellsprache simuliert. Bei diesem Transformationsschritt kommen unter Umständen Ressourcenbeschränkungen ins Spiel, weil der Speicher der Zielmaschine endlich ist, während Quellsprachen oft unendlichen Speicher annehmen.

Für die Verifikation einer Programmtransformation gehen wir davon aus, daß der Speicherzustandsraum der beteiligten ASMs $\mathcal{A}_1$ und $\mathcal{A}_2$ identisch ist. Wir beweisen die Simulationsbeziehung zwischen Quell- und Zielprogramm indem wir das Zielprogramm in einer kombinierten

ASM $\mathcal{A}_{1,2} = \mathcal{A}_1 \uplus \mathcal{A}_2$ ausführen, in der sowohl Operationen der Quellsprache, als auch Operation der Zielsprache interpretiert werden können[1].

Die Korrektheit der Transformation ergibt sich durch Induktion über die Anwendung von Transformationsregeln und dem Nachweis der lokalen Simulation. Daraus leitet sich die Simulation über dem kompletten Programm ab. Nach vollständiger Transformation des Quellprogramms können wir die benutzte ASM auf die Zielsprach-ASM reduzieren, da sichergestellt ist, daß keine Kommandos bzw. Operationen der Quelle mehr vorkommen. Dieser Schritt entspricht formal der Bildung des σ-Redukts auf den $\Sigma_{1,2}$-Algebren mit dem Signaturmorphismus $\sigma : \Sigma_2 \to \Sigma_1 \cup \Sigma_2$. Gleichzeitig können die Regeln zur Interpretation von Quellsprachkommandos vergessen werden.

Die Abbildung 7.5 veranschaulicht noch einmal die Schritte des Beweisschemas. Im ersten Schritt bleibt das Programm unverändert, aber die Interpretation ändert sich. Im zweiten Übersetzungsschritt vereinigen wir die Semantiken der Quelle und des Ziels und transformieren das Programm.

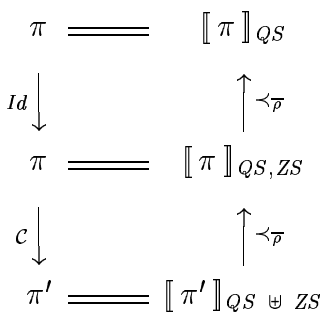


Abbildung 7.5: Verifikation einer Übersetzung

7.4.2 Speicherabbildung: Beispiel Adreßabbildung

Der erste Schritt des Beweisschemas muß bei der Abbildung einer Quellsprache in eine Zwischen- oder Zielsprache auf jeden Fall durchgeführt werden, weil die Speicherzustandsräume von Quelle und Ziel unterschiedlich sind. Wir demonstrieren diesen Übersetzungsschritt am Beispiel der Abbildung von *AL* in eine Zwischensprache mit adressierbarem Speicher. Diese Transformation ist insofern global, daß alle Sprachelemente davon betroffen sind. Der Korrektheitsbeweis wird über strukturelle Induktion geführt. Die Theoreme aus dem vorigen Abschnitt können hierbei nicht angewendet werden.

In *AL* werden bei jedem Aufruf einer Funktion Schachteln für die Parameter und die lokalen Variablen der Funktion erzeugt. Diese Schachteln werden jeweils durch eine Kellerstruktur verwaltet. Die Kellerschachteln sind strukturierte *AL*-Objekte, die beim bzw. vor dem Betreten einer Funktion erzeugt und beim Verlassen wieder vergessen werden. Im Laufzeitsystem von *AL* werden die Schachteln durch die dynamischen Funktionen *params* und *locals* verwaltet. Zusätzlich gibt es noch die ausgezeichnete Struktur *globals* für die globalen Variablen. Objekte werden über ihre symbolischen Referenzen zugegriffen.

1) Wir setzen dabei voraus, daß die Befehlszähler von Quelle und Ziel identisch sind. Ist das nicht der Fall, dann muß dies durch eine entsprechende Uminterpretation, die ebenfalls zu verifizieren ist, sichergestellt werden.

Der Speicher der Zielsprache *MIS* ist eine sequentielle byte-orientierte Struktur. Der Laufzeitkeller und die Halde sind Bereiche auf dieser Struktur und die Objekte werden über Adressen zugegriffen. In diesem Übersetzungsschritt kommen zum ersten Mal Ressourcenbeschränkungen der Zielmaschine ins Spiel, da wir nicht sicher sein können, daß der Laufzeitkeller bzw. die Halde im Speicher der Zielmaschine beliebig groß werden können.

Bei der Speicherabbildung von *AL* gehen wir in zwei Schritten vor. Zuerst interpretieren wir den Zugriff auf Objekte in *AL* um. Das betrifft die Load– und Store-Kommandos und die Kommandos, die Funktionsaufrufe betreffen (NewGlobalFrame, NewLocalFrame, NewParamFrame, Call, Return) sowie das Alloc-Kommando zum Anlegen neuer Objekte auf der Halde. Anstatt ein Objekt durch eine symbolische Referenz anzusprechen, nehmen wir an, daß es durch eine Adresse identifiziert wird. Eine Adresse berechnet sich, wie von der Zielmaschine definiert, aus einer Basisadresse und einer Relativadresse. Diese Uminterpretation beläßt das Programm unverändert. Stattdessen wechseln wir auf der formalen ASM-Ebene vom Speicherzustandsraum der Quellsprach-ASM in den Speicherzustandsraum der Zielsprach-ASM. In einem zweiten Schritt weisen wir mit der neuen Interpretation eine Simulationsbeziehung nach.

Annahmen

Der Adreßraum der Zielmaschine ist durch die Sorte Addr beschrieben. Üblicherweise sind Adressen Bit-Sequenzen, aber dieses Detail ist hier nicht von Interesse. Auf Addr gibt es von der Zielmaschine definierte Funktionen:

$$\boxplus : \mathsf{Addr} \times \mathsf{Int} \to \mathsf{Addr}$$
$$\boxminus : \mathsf{Addr} \times \mathsf{Addr} \to \mathsf{Int}$$

$\boxplus$ addiert eine Relativadresse zu einer Basisadresse, $\boxminus$ berechnet die Differenz von zwei Adressen. Es gelten die folgenden Gleichungen für alle $a, a' \in \mathsf{Addr}$ und $i, j \in \mathsf{Int}$:

$$\begin{aligned} a &= a' \boxplus (a \boxminus a') \\ a \boxminus a' &= 0 \ominus (a' \boxminus a) \\ (a \boxplus i) \boxminus (a \boxplus j) &= i \ominus j \end{aligned}$$

Wir nehmen ohne Beschränkung der Allgemeinheit einen Byte-orientierten Speicher an, der durch die Funktion

$$mem : \mathsf{Addr} \to \mathsf{Byte}$$

mit $\mathsf{Byte} \triangleq \mathsf{Bit}^8$ und $\mathsf{Bit} \triangleq \{0, 1\}$ modelliert ist. Für einen anders adressierten Speicher geht man analog vor.

Die Zielmaschine legt auch die Datentypen *Int, Float* und *Bool* fest und definiert die Sorten Int, Float und Bool. Wir gehen hier davon aus, daß *AL* und die Zielmaschine Basisdatentypen mit gleicher Semantik haben. Ansonsten müßten wir noch einen zusätzlichen Schritt einführen, der die Datentypen ineinander abbildet. Die Zielmaschine gibt die Codierung von Werten eines Typs als Bytefolgen vor. Da Werte in mehreren Bytes codiert sein können, müssen wir die Größe eines Datums kennen. Diese Größe wird von der Zielmaschine durch die Funktion

$$typesize : \mathsf{Type} \to \mathrm{N}$$

festgelegt. Wir benutzen die Funktion $content : \mathsf{Addr} \times \mathsf{Type} \to \mathsf{Byte}^*$ um Werte aus dem Byte-orientierten Speicher zu lesen. *content* ist mit Hilfe der Funktion $cont : \mathsf{Addr} \times \mathrm{N} \to \mathsf{Byte}^*$ definiert:

$$\begin{array}{rcl} cont(a,0) & \triangleq & [mem(a)] \\ cont(a,n+1) & \triangleq & [mem(a)|cont(a \boxplus 1, n)] \\ content(a,t) & \triangleq & cont(a, typesize(t)-1) \end{array} \tag{7.1}$$

Das Makro $\overline{content}$ beschreibt, wie Werte in benachbarten Speicherstellen abgelegt werden. Es ist mit Hilfe des Makros $\overline{cont}$ definiert:

$$\begin{array}{rcl} \overline{cont}(a,0) := e & \triangleq & mem(a) := hd(e) \\ \overline{cont}(a,n+1) := e & \triangleq & mem(a) := hd(e); \\ & & \overline{cont}(a \boxplus 1, n) := tl(e) \\ \overline{content}(a,t) := e & \triangleq & \overline{cont}(a, typesize(t)-1) := e \end{array} \tag{7.2}$$

Die Speicherworte einer Maschine können nicht an beliebigen Adressen beginnen. Sie müssen ausgerichtet sein. Die Ausrichtung wird von der Zielmaschine vorgegeben und ist durch die Funktion $IsAligned : \mathsf{Addr} \times \mathsf{Type} \rightarrow \mathsf{Bool}$ beschrieben.

Bemerkung: Die Abbildung der Datentypen der Quellsprache auf Datentypen der Zwischen- bzw. Zielsprache beeinflußt die Erzeugung von Adressen, weil Typgrößen und Ausrichtung berücksichtigt werden. Trotzdem können wir diesen Übersetzungsschritt separat betrachten, da wir die Adreßerzeugung mit Parametern für Größe und Ausrichtung von Typen definieren. Wir setzen dann für den Korrektheitsbeweis voraus, daß Daten korrekt abgebildet werden. ♢

Für Adressen nehmen wir an, daß Prädikate *greater* und *lower* definiert sind. Ihre Definition hängt davon ab, ob der Laufzeitkeller nach unten oder nach oben wächst. Wächst er nach unten, dann ist $greater(x, addr) \hat{=} (x \boxminus addr) < 0$, sonst $greater \hat{=} (x \boxminus addr) > 0$. *lower* ist analog definiert.

Dynamische Semantik der Zwischensprache

Der Speicher der Zielmaschine ist partitioniert in den Keller und die Halde.

$tos, toh : \mathsf{Addr}$

zeigen den aktuellen Kellerpegel bzw. den aktuellen Pegel der Halde an. Sie werden zur Feststellung eines Speicherüberlaufs (der Pegel des Kellers übersteigt den Pegel der Halde oder umgekehrt) und für die Erzeugung neuer Basisadressen im Keller und auf der Halde benutzt.

$$\begin{array}{rcl} params & : & \mathsf{Int} \rightarrow \mathsf{Addr} \\ locals & : & \mathsf{Int} \rightarrow \mathsf{Addr} \\ globals & : & \rightarrow \mathsf{Addr} \\ caller & : & \mathsf{Int} \rightarrow \mathsf{Addr} \end{array}$$

modellieren zusammen einen Laufzeitkeller für Parameterschachteln und Schachteln für lokale und globale Variablen. Zusätzlich benutzen wir die externen Funktionen:

$$\begin{array}{rcl} NewStackAddr : & \mathsf{Addr} \times \mathsf{Type} \rightarrow \mathsf{Addr} \\ NewHeapAddr : & \mathsf{Addr} \times \mathsf{Type} \times \mathsf{Int} \rightarrow \mathsf{Addr}. \end{array}$$

NewStackAddr berechnet in Abhängigkeit des Kellerpegels und eines Typs eine neue Basisadresse für eine Schachtel. *NewHeapAddr* erzeugt unter Berücksichtigung des aktuellen Haldenpegels eine neue Adresse auf der Halde unter der die entsprechende Anzahl von Elementen eines Typs zusammenhängend gespeichert werden können. Der Zugriff auf Adressen der Halde

funktioniert nach dem gleichen Schema. Ein Alloc-Kommando veranlaßt das Erzeugen einer neuen Haldenadresse und berechnet einen neuen Haldenpegel.

Bemerkung: Aus Platzgründen verzichten wir auf die detaillierte Beschreibung von Ausnahmen. Ausnahmen werden durch Prädikate modelliert, die in einem Zustand erfüllt sein können. Für uns ist hier nur der Ressourcenfehler Speicherüberlauf wichtig, den wir mit Hilfe des aktuellen Keller- und Haldenpegels und dem Wert, um den der Keller bzw. die Halde wächst, feststellen können.

$$memory\_overflow : \mathsf{Addr} \times \mathsf{Int} \times \mathsf{Addr} \rightarrow \mathsf{BOOL}$$

Tritt ein Speicherüberlauf auf, dann wird der Steuerfluß an ein spezielles Fehler-Kommando übergeben. ♢

Die dynamische Semantik der für uns relevanten Instruktionen der Zwischensprache ist in Abbildung 7.6 beschrieben. Wir haben auf die Angabe der Regeln für NewLocalFrame verzichtet, weil sie sich nur in der Aktualisierung der Struktur *locals* anstatt *params* in NewParamFrame unterscheiden.

Wir setzen voraus, daß das Betriebssystem die Anfangsadresse des Laufzeitkellers definiert und die korrekte Ausrichtung der Schachtel der globalen Variablen zugesichert ist. Wir abstrahieren davon mit Hilfe der Funktion *initstack*. Die Anfangsadresse der Halde ist initial durch die Funktion *initheap* gegeben.

New-Kommandos bewirken auf der Zielmaschine keine Veränderung des Speicherzustands, da dieser schon durch das Anlegen von Schachteln für Parameter und lokale Variablen verändert wird. Zur Erinnerung: Entsprechend unserem Beweisschema bleibt das Programm bei der Speicherabbildung unverändert, daher können die New-Kommandos nicht einfach entfallen.

Die Abarbeitung eines NewParamFrame-Kommandos erhöht den Kellerpegel um die Größe der Parameterschachtel und schafft Platz zum Speichern des alten Kellerpegels, der in *acttos* abgelegt wird. Die Anfangsadresse der Parameterschachtel wird in *params* gespeichert. Ein Funktionsaufruf erhöht ebenfalls den Kellerpegel, da der dynamische Vorgänger gesichert werden muß. Außerdem wird zum ersten Kommando der gerufenen Funktion gesprungen, und die Aufruftiefe wird erhöht. Bei Speicheroperationen benutzen wir anstatt symbolischer Adressen nun Relativadressen, die durch die Funktion *reladdr* vorgegeben sind. *reladdr* wird durch die Speicherabbildung im nächsten Abschnitt definiert.

Das Programm selbst ist auf der Zielmaschine im Speicher abgelegt. Daher wird bei einem Funktionsaufruf die Adresse der Rücksprungstelle im Laufzeitkeller abgelegt. In unserer Zwischensprache, ist das Programm noch ein abstrakter Graph und daher ist die Rücksprungstelle ein Kommando. Die Verwaltung erfolgt über die Funktion *caller*. Allerdings berücksichtigen wir bei der Speicherabbildung schon, daß Rückkehradressen später auf dem Laufzeitkeller verwaltet werden und sehen Speicherplatz für sie vor. Die Verwaltung der Rücksprungadresse im Laufzeitkeller wird durch einen zusätzlichen Übersetzungsschritt eingeführt, bei dem die Funktion *caller* eliminiert wird. Für den direkten Vergleich der Semantiken sind in Abb. 7.7 noch einmal die entsprechenden Regeln aus der *AL*-Semantik aufgeführt.

Bemerkung: Wir benutzen dabei $\perp$ als Wert über den keine Aussagen gemacht werden können. Im Gegensatz dazu steht *undef* für einen nicht definierten Wert. ♢

```
if NewGlobalFrame(ct) then
  if mem_overflow(initstack,framesize(type(ct)),initheap) then
    exception := "memory_overflow"
    ct := ErrorTask
  else
    value(ct,reclevel) := initstack
    tos := initstack ⊞ framesize(type(ct))
    toh := initheap
    globals := initstack
    proceed
endif

if NewParamFrame(ct) then
  if mem_overflow((NewStackAddr(tos,type(ct)),
                  framesize(type(ct))+typesize(addr),
                  toh) then
    exception := "memory_overflow"
    ct := ErrorTask
  else
    value(ct,reclevel) := NewStackAddr(tos,type(ct))
    tos:=NewStackAddr(tos,type(ct))⊞framesize(type(ct))
    acttos(reclevel) := tos
    params(reclevel+1) := NewStackAddr(tos,type(ct))
    proceed
  endif
endif

if LoadStruct(ct) then
  value(ct,reclevel) :=
      content(value(ct.source,reclevel) ⊞ reladdr(type(ct),value(ct.selector,reclevel)),type(ct.selector))
  proceed
endif

if Store(ct) then
   content(value(ct.dest,reclevel), type(ct)) := value(ct.source,reclevel)
  proceed
endif

if Alloc(ct) then
  if mem_overflow(tos, −objsize(type(ct.elements), value(ct.size,reclevel)), toh) then
    exception := "memory_overflow"
    ct := ErrorTask
  else
    toh := toh ⊞ NewHeapAddr(toh, type(ct.elements), value(ct.size,reclevel))
    value(ct,reclevel) := NewHeapAddr(toh, type(ct.elements), value(ct.size,reclevel))
    proceed
  endif
endif

if Call(ct) then
  if mem_overflow(tos, typesize(addr), toh) then
    exception := "memory_overflow"
    ct := ErrorTask
  else
    tos := tos + typesize(addr)
    ct := firsttask(ct.id)
    caller(reclevel+1) := ct
    reclevel := reclevel + 1
  endif
endif
```

```
if New(ct) then
  proceed
endif
```

```
if LoadBasic(ct) then
  value(ct,reclevel) :=
     content(value(ct.source,reclevel),type(ct))
  proceed
endif
```

```
if Return(ct) then
  tos := acttos(reclevel-1)
  ct := caller(reclevel).nexttask(0)
  reclevel := reclevel - 1
  params(reclevel) := undef
  locals(reclevel) := undef
endif
```

Abbildung 7.6: Ausschnitt aus der dynamischen Semantik der Zwischensprache

```
if NewParamFrame(ct) then
  extend Reference with r
    type(r) := type(ct)
    value(ct,reclevel) := r
    do forall t : (t ∈ components(type(ct)))
      r.(snd(t)) := value(ct.(snd(t)),reclevel)
    enddo
    params(reclevel+1) := r
  endextend
  proceed
endif

if LoadBasic(ct) then
  value(ct,reclevel) :=
       content(value(ct.source,reclevel))
  proceed
endif

if New(ct) then
   if BasicType(type(ct))
      extend Reference with r
         type(r) := type(ct)
         content(r) := ⊥
         value(ct,reclevel):= r
      endextend
   else
      extend Reference with r
         type(r) := type(ct)
         value(ct,reclevel):= r
         do forall t : (t ∈ components(type(ct)))
            r.(snd(t)) := value(ct.(snd(t)),reclevel)
         enddo
      endextend
   endif
   proceed
endif

if LoadStruct(ct) then
  value(ct,reclevel) := value(ct.source,reclevel).(value(ct.selector,reclevel))
  proceed
endif

if Store(ct) then
  content(value(ct.dest,reclevel)) := value(ct.source,reclevel)
  proceed
endif

if Return(ct) then
  ct := caller(reclevel).nexttask(0)
  reclevel := reclevel - 1
  params(reclevel) := undef
  locals(reclevel) := undef
endif

if Call(ct) then
   ct := firsttask(ct.id)
   caller(reclevel + 1) := ct
   reclevel := reclevel + 1
endif

if Alloc(ct) then
  if numofallocated(ct) = undef then
    extend HeapReference with r
      type(r) := type(ct)
      value(ct,reclevel) := r
      content(r) := ⊥
    endextend
    numofallocated(ct) := 0
    ct := first(ct)
  elsif numofallocated(ct) < value(ct.size,reclevel) then
    r.data(numofallocated(ct)) := value(ct.elements,reclevel)
    numofallocated(ct) := numofallocated(ct) + 1
    ct := elements
  else
    numofallocated(ct) := undef
    proceed
  endif
endif
```

Abbildung 7.7: Ausschnitt aus der dynamischen Semantik von *AL*

Eigenschaften der Speicherabbildung

Die Speicherabbildung muß so definiert sein, daß es in jedem Zustand q des Quellprogramms eine injektive Abbildung der in q zugreifbaren Objekte auf Speicheradressen des Zielprogramms gibt. Da Werte eines Objektes durch mehrere Bytes codiert sein können, die Speicheradressierung unserer Zielmaschine aber Byte-orientiert ist, muß ausgeschlossen werden, daß sich Objekte im Speicher überlappen. Desweiteren muß sichergestellt sein, daß die Ausrichtung der Adressen, in Abhängigkeit vom Typ der Werte, eingehalten wird.

Unsere Zielsprache *MIS* unterscheidet Adressen lokaler und globaler Variablen sowie Adressen von Parametern. In *AL* sind die Objekte einer Funktion durch ein strukturiertes Objekt (Schachtel) beschrieben, das beim Aufruf erzeugt wird, wenn es sich um lokale Variablen handelt bzw. vor dem Aufruf, wenn es sich um Parameter handelt. Schachteln können selbst wieder Schachteln für strukturierte Objekte enthalten. Wir setzen eine Schachtel mit ihrem statischen Typ gleich, der ein Mengentyp *Set* ist.

$$\mathsf{Frame} \mathrel{\hat{=}} \mathsf{Set}.$$

Im folgenden definieren wir Anforderungen an die Abbildung von Objekten einer Schachtel auf Adressen.

Anforderung 7.1 (Relativadressen)
Sei $f : \mathsf{Frame}$ eine beliebige Schachtel, dann müssen für die Funktion

$$\mathit{reladdr} : \mathsf{Frame} \times \mathsf{Selector} \to \mathsf{Int}$$

zur Erzeugung von Relativadressen folgende Eigenschaften gelten:

$$\forall n_1, n_2 \in f : \mathit{greater}(\mathit{reladdr}(f, n_1) + \mathit{size}(\mathit{type}(n_1)), \mathit{reladdr}(f, n_2)) \vee \mathit{greater}(\mathit{reladdr}(f, n_2) + \mathit{size}(\mathit{type}(n_2)), \mathit{reladdr}(f, n_1)) \tag{7.3}$$

$$\forall n \in f : \mathit{IsAligned}(\mathit{reladdr}(f, n), \mathit{type}(n)) \tag{7.4}$$

Dabei ist

$$\mathit{size}(t) = \begin{cases} \mathit{typesize}(t), & \text{wenn } t \text{ aus } \mathsf{BasicType} \\ \mathit{framesize}(t), & \text{wenn } t \text{ aus } \mathsf{Set} \end{cases}$$

Die Anforderungen (7.3) und (7.4) an *reladdr* sichern zu, daß:

1. *reladdr* injektiv ist und sich die erzeugten Relativadressen einer Schachtel nicht überlappen
2. die Relativadressen der Schachtel entsprechend den Typen der Werte, die sie aufnehmen, korrekt ausgerichtet sind.

Damit wird jedem Objekt einer Schachtel offensichtlich eine eindeutige Adresse zugeordnet, und das Schreiben eines Datums an eine Adresse kann nicht Teildaten eines anderen Objekts überschreiben.

Können wir nun noch nachweisen, daß in allen Zuständen q eines Programmlaufs, alle existierenden Schachteln eine eindeutige Basisadresse im Speicher haben, sich nicht überlappen (7.5) und so im Speicher abgelegt sind (7.6), daß neben der Schachtel selbst auch alle ihre Elemente korrekt ausgerichtet sind, dann können wir damit die Korrektheit der Speicherabbildung nachweisen.

Anforderung 7.2 (Speicherbereiche von Schachteln)
Die Speicherabbildung muß sicherstellen, daß Schachteln für Parameter, lokale und globale Variablen unterschiedliche Basisadressen haben, sich nicht überlappen und korrekt ausgerichtet sind. Sei $addr(f)$ die konkrete Adresse einer Schachtel und $stack(q) = \{locals(0), \ldots, locals(reclevel), params(0), \ldots, params(reclevel), globals\}$ die Menge der Anfangsadressen der Schachteln auf dem Laufzeitkeller in einem Zustand q, dann muß gelten:

$$\forall f_1, f_2 \in \mathsf{Frame},\ addr(f_1), addr(f_2) \in stack(q) : f_1 \neq f_2 \Rightarrow \tag{7.5}$$
$$greater(addr(f_1) \boxplus framesize(f_1), addr(f_2)) \vee greater(addr(f_2) \boxplus framesize(f_2), addr(f_1))$$
$$\forall f \in \mathsf{Frame} :\ addr(f) \in stack(q) \Rightarrow IsAligned(addr(f), type(f)) \tag{7.6}$$

Anforderung 7.3 (Schachtelgröße)
Sei n die Komponente der Schachtel f mit maximaler Relativadresse $reladdr(f, n)$, dann gilt:

$$framesize(f) \geq reladdr(f, n) + typesize(type(n)) \tag{7.7}$$

Anforderung 7.4 (Basisadressen)
In der dynamischen Semantik benutzen wir eine Funktion *NewStackAddr*, die in Abhängigkeit eines Kellerpegels und einer Schachtel, eine neue Basisadresse berechnet. Für diese Funktion gilt:

$$x = NewStackAddr(addr, type) \Rightarrow IsAligned(x, type) \wedge greater(x, addr) \tag{7.8}$$

Anforderung 7.5 (Haldenadressen)
Für die Funktion *NewHeapAddr* fordern wir analog

$$x = NewHeapAddr(addr, el\_type, size) \Rightarrow IsAligned(x, el\_type) \wedge greater(x, addr) \tag{7.9}$$

Die Elemente des neuen Haldenobjekts definieren eine Schachtel vom Typ $elements\_type = set(el\_type, 0), (el\_type, 1), \ldots, (el\_type, size - 1))$. Der Aufbau der Schachtel hängt von der, unter Umständen dynamischen, Anzahl der Elemente von Alloc ab. Die Funktion

$$objsize : \mathsf{Type} \times \mathsf{Int} \rightarrow \mathsf{Int}$$

berechnet zu einem Elementtyp und einer Anzahl die Größe der entsprechend *elements_type* beschriebenen Schachtel. Es gilt:

$$objsize(t, s) = framesize(elements\_type) \tag{7.10}$$
$$\forall i, 0 \leq i < size : IsAligned(reladdr(elements\_type, i)) \tag{7.11}$$
$$\forall i, 0 \leq i < size - 1 : \tag{7.12}$$
$$abs(reladdr(elements\_type, i + 1) - reladdr(elements\_type, i)) < typealign(el\_type)$$

$typealign : \mathsf{Type} \rightarrow \mathsf{Int}$ gibt zu einem Typ die Ausrichtung an.

Gleichung (7.11) besagt, daß die Adressen der neuen Haldenelemente korrekt ausgerichtet sind, (7.12) sichert zu, daß die Elemente hintereinander und zusammenhängend im Speicher liegen. In unserer Zwischensprache ist keine Speicherbereinigung vorgesehen. Durch eine entsprechend kompliziertere Definition von *NewHeapAddress* kann dies jedoch mit berücksichtigt werden.

Bemerkung: Die Berechnung von Relativadressen innerhalb eines Frames f bzw. für Instanzen eines Frames f erfolgt anhand der Struktur des Frames. Es gibt Sprachen die verlangen,

daß das Speicherlayout exakt die Reihenfolge bei der Deklaration von Variablen einzuhalten hat. In diesem Fall fordern wir für die Relativadressen von Frames f die zusätzliche Eigenschaft:

$$\forall n_1, n_2 \in f : declared\_before(f, n_1, n_2) \Rightarrow reladdr(f, n_1) < reladdr(f, n_2) \tag{7.13}$$

declared_before ist ein Prädikat, das prüft, ob eine Variable vor einer anderen deklariert wurde. ♢

Die Speicherabbildung impliziert eine Uminterpretation von AL-Programmen. Wir interpretieren Referenzen als Adressen und Selektoren als Relativadressen. Damit werden Load- und Store-Kommandos, die Speicherzugriffe über Referenzen modellieren, zu Adreßberechnungen mit anschließendem Zugriff in den Speicher.

Spezifikation der Speicherabbildung

Für die formale Definition der Speicherabbildung setzen wir voraus, daß der Zustandsraum einer ASM algebraisch durch eine Menge von Gleichungen spezifiziert ist. Die Speicherabbildung ist dann durch ein Tripel $(\sigma, E, .|_{\sigma'})$ beschrieben.

1. $\sigma : \Sigma_1 \to (\Sigma_1 \cup \Sigma_2)$ ist ein Signaturmorphismus, der die Signatur der Quellmaschine in die Vereinigung der Signaturen der Quelle und des Ziels einbettet.
2. E ist eine Menge von Gleichungen, die die Beziehung zwischen den Funktionen der beiden ASMs $\mathcal{A}_1$ und $\mathcal{A}_2$ herstellt und einen vereinigten Zustandsraum von $\mathcal{A}_1$ und $\mathcal{A}_2$ beschreibt.
3. $.|_{\sigma'}$, $\sigma' : \Sigma_2 \to \Sigma_1 \cup \Sigma_2$ ist ein Restriktionsfunktor mit dem wir den vereinigten Zustandsraum von $\mathcal{A}_1$ und $\mathcal{A}_2$ wieder auf den Zustandsraum von $\mathcal{A}_2$ einschränken.

σ bildet Funktionsnamen von Σ_1 auf sich selbst ab und führt nicht vorhandene Zwischensprachfunktionen ein. Das sind einmal die statischen Funktionen *reladdr*, *typesize*, *framesize*, *objsize*, *NewStackAddr*, *NewHeapAddr*, *memory_overflow* und *ErrorTask*, sowie die dynamischen Funktionen *tos*, *toh*, *acttos* und *exception*, die wir in den letzten Abschnitten beschrieben haben. Durch die Gleichungen E wird die Semantik der neuen Funktionen definiert und die Beziehung zu Funktionen aus Σ_1 hergestellt. Wir haben in den letzten Abschnitten die einzelnen Funktionen schon beschrieben. Für *reladdr* gelten (7.3) und (7.4), für *NewStackAddr* (7.8), für *NewHeapAddr* (7.9) und für *framesize* gilt (7.7). *initstack* und *initheap* haben wir nur informell beschrieben, aber die Gleichungen sind klar. Die Trägermenge von Addr ist die Menge aller Adressen des Speicherbereichs und wird dem Programm vom Betriebssystem zugeteilt.

$$\forall a \in [\![\mathsf{Addr}]\!] : [\![greater]\!]([\![initstack]\!], a) \wedge [\![greater]\!](a, [\![initheap]\!]) \tag{7.14}$$

ErrorTask ist ein neues Element von $[\![\mathsf{Task}]\!]$. Im Initialzustand gilt für die dynamischen Funktionen:

$$[\![tos]\!] = undef,\ [\![toh]\!] = undef,\ [\![exception]\!] = undef \tag{7.15}$$

Korrektheit der Speicherabbildung

Die Korrektheit der Speicherabbildung wird für alle Programme π durch Nachweis einer Simulationsbeziehung zwischen der Semantik der Quelle $\mathcal{A}_{AL}(\pi)$ (im folgenden als $\mathcal{A}_1$ bezeichnet)

und der neuen Semantik mit dem Speicherzustandsraum der Zielsprache $\mathcal{A}_{ZR'}(\pi)$ (im folgenden als $\mathcal{A}_2$ bezeichnet) gezeigt.

Für die Verifikation benutzen wir eine injektive Funktion $\rho : Alg(\Sigma_2) \to Alg(\Sigma_1)$ über den Zuständen der Quelle und des Ziels, die durch die Speicherabbildung und die damit verbundene Uminterpretation der AL-Kommandos induziert wird. Wir haben zu zeigen, daß

$$\forall q_0 \in I_2\, \forall q \in Alg(\Sigma_2) : q_0 \xrightarrow{*}_{(2)} q \wedge [\![\, exception \,]\!]_q \neq [\![\, memory\_overflow \,]\!] \Rightarrow \\ \rho(q_0) \in I_1 \wedge \rho(q) \in Alg(\Sigma_1) \wedge \rho(q_0) \xrightarrow{*}_{(1)} \rho(q)$$

Zusätzlich benötigen wir einige Hilfsfunktionen. Die Funktion

$$addr : \mathsf{Reference} \to \mathsf{Addr}$$

stellt die Beziehung zwischen Referenzen und Adressen her.

$$addr(params(i)) = \begin{cases} NewStackAddr(addr(globals) \boxplus globals.size, params(1).align), & \text{wenn } i = 1 \\ NewStackAddr(addr(locals(i-1)) \boxplus locals(i-1).size, params(i).align), & \text{wenn } i > 1 \end{cases}$$

$$addr(locals(i)) = \begin{cases} NewStackAddr(addr(params(1)) \boxplus params(1).size, locals(1).align), & \text{wenn } i = 1 \\ NewStackAddr(addr(params(i)) \boxplus params(i).size, locals(i).align), & \text{wenn } i > 1 \end{cases}$$

$$addr(r) = \begin{cases} initstack \boxplus reladdr(mainframe, type(r).name), & \text{wenn } IsGlobal(r, mainframe) \\ addr(locals(reclevel)) \boxplus reladdr(pframe, type(r).name), & \text{wenn } IsLocal(r, pframe) \\ addr(params(reclevel)) \boxplus reladdr(pframe, type(r).name), & \text{wenn } IsParam(r, pframe) \\ addr(r') \boxplus reladdr(type(r'), type(r).name), & \text{wenn } IsRecordElement(r, r') \\ addr(r') \boxplus reladdr(type(r'), reladdr(r'.1) * value(r, reclevel), & \text{wenn } IsArrayElement(r, r') \end{cases}$$

Die Prädikate *IsGlobal, IsLocal, IsRecordElement* und *IsArrayElement* entscheiden für eine Referenz, ob es sich um eine globale, lokale Variable, einen Parameter oder um ein Teilobjekt eines strukturierten Datentyps handelt. *mainframe* und *pframe* bezeichnen die Schachteln für die Variablen des Hauptprogramms bzw. einer Funktion.

$$ObjRefs(r) \;\hat{=}\; \bigcup_{r_i \in Components(r)} ObjRefs(r_i) \cup \{r\}$$

$$StackRefs \;\hat{=}\; ObjRefs(globals) \cup \\ ObjRefs(locals(0)) \cup \ldots \cup ObjRefs(locals(reclevel)) \cup \\ ObjRefs(params(0)) \cup \ldots \cup ObjRefs(params(reclevel))$$

$$HeapRefs \;\hat{=}\; \bigcup_{r \in HeapReference} ObjRefs(r)$$

ObjRefs berechnet rekursiv die Menge der Referenzen eines Objekts und aller seiner Teilobjekte. *StackRef* beschreibt die Menge aller Referenzen des aktuellen Laufzeitkellers. *HeapRefs* berechnet die Menge der Referenzen auf der Halde.

Satz 7.1 (Speicherabbildung)
Die ASM $\mathcal{A}_2$ simuliert das beobachtbare Verhalten von $\mathcal{A}_1$ bis auf den Ressourcenfehler Speicherüberlauf.

Beweis Wir definieren zuerst die Relation $\rho \subseteq Alg(\Sigma_2) \times Alg(\Sigma_1)$. Es gilt $q' \,\rho\, q$ genau dann, wenn

$$[\![ct]\!]_q = [\![ct]\!]_{q'} \tag{7.16}$$
$$[\![reclevel]\!]_q = [\![reclevel]\!]_{q'} \tag{7.17}$$
$$[\![caller(reclevel)]\!]_q = [\![caller(reclevel)]\!]_{q'} \tag{7.18}$$
$$[\![input]\!]_q = [\![input]\!]_{q'} \tag{7.19}$$
$$[\![output]\!]_q = [\![output]\!]_{q'} \tag{7.20}$$

$$\forall t \in [\![\mathsf{Task} \setminus \mathsf{New}]\!]_q \forall i \in \mathsf{Int}, 0 \le i \le [\![reclevel]\!]_q :$$
$$[\![value]\!]_{q'}(t,i) = \begin{cases} [\![addr]\!]_q([\![value]\!]_q(t,i)), & \text{falls } [\![type]\!]_q(t) \in [\![RefType]\!]_q \\ [\![value]\!]_q(t,i), & \text{sonst} \end{cases} \tag{7.21}$$

$$\begin{aligned} \forall r \in [\![StackRefs]\!]_q : \quad & [\![greater]\!]([\![addr]\!]_q(r), [\![initstack]\!]_{q'}) \wedge \\ & [\![greater]\!]([\![tos]\!]_{q'}, [\![addr]\!]_q(r)) \wedge \\ & [\![content]\!]_q(r) \notin \{\bot, undef\} \\ \Rightarrow \\ [\![content]\!]_{q'}([\![addr]\!]_q(r), [\![type]\!]_q(r)) = & \begin{cases} [\![addr]\!]_q([\![content]\!]_q(r)), & \text{falls} \\ & [\![type]\!]_q(r) \in [\![RefType]\!]_q \\ [\![content]\!]_q(r), & \text{sonst} \end{cases} \end{aligned} \tag{7.22}$$

$$\begin{aligned} \forall r \in [\![HeapRefs]\!]_q : \quad & [\![greater]\!]([\![initheap]\!]_{q'}, [\![addr]\!]_q(r)) \wedge \\ & [\![greater]\!]([\![addr]\!]_q(r), [\![toh]\!]_{q'}) \wedge \\ & [\![content]\!]_q(r) \notin \{\bot, undef\} \Rightarrow \\ [\![content]\!]_{q'}([\![addr]\!]_q(r), [\![type]\!]_q(r)) = & \begin{cases} [\![addr]\!]_q([\![content]\!]_q(r)), & \text{falls} \\ & [\![type]\!]_q(r) \in [\![RefType]\!]_q \\ [\![content]\!]_q(r), & \text{sonst} \end{cases} \end{aligned} \tag{7.23}$$

Für alle anderen Funktionen, die nicht von der Speicherabbildung betroffen sind, ist die Interpretation in der Quelle und im Ziel identisch.

Die Adreßabbildung definiert eine injektive Abbildung von Referenzen auf Adressen. Da die Definition von ρ alle davon betroffenen Funktionen einbezieht und ansonsten die Gleichheit der Interpretationen fordert, definiert ρ eine injektive Funktion von $Alg(\Sigma_2)$ nach $Alg(\Sigma_1)$.

Die Simulation wird durch Induktion über die Anzahl der Zustandsübergänge der Zielmaschine bewiesen. Wir verzichten beim Beweis darauf, Gleichheitsbeziehungen zu erwähnen, die offensichtlich erhalten bleiben, weil die beteiligten Funktionen nicht aktualisiert wurden. Wird eine Beziehung im Beweis nicht explizit angesprochen, dann gilt diese Beziehung auch nach dem Übergang auf der Quell- bzw. Zielmaschine. Tritt auf der Zielmaschine ein Speicherüberlauf auf, dann kann keine Simulationsbeziehung mehr etabliert werden, weil es in AL keine Begrenzung des Speichers gibt und dieser Fehler nicht auftritt. Auch wenn wir diesen Aspekt im folgenden nicht mehr explizit erwähnen, gilt das an allen Stellen, an denen ein Speicherüberlauf auftreten kann. Mit Ausnahme des Alloc-Kommandos wird bei einem Zustandsübergang

auf der Quelle und auf dem Ziel jeweils der Befehlszähler gleich weitergeschaltet. Daher gilt in diesen Fällen (7.16). Ebenso werden *reclevel* und *caller* jeweils gleich aktualisiert daher sind auch (7.17) und (7.18) erfüllt.

Induktionsanfang: $n = 0$
Im Initialzustand stehen die Zustände trivialerweise in Beziehung. Das Programm ist unverändert, kein Kommando wurde ausgeführt und es gibt noch keine Variablen und Parameter, siehe auch Abschnitt B.6. Für $\mathcal{A}_2$ gilt zusätzlich (7.14) und (7.15).

Induktionshypothese: Die Behauptung gilt für Zustandsfolgen der Länge n.

Induktionsschluß:
Sei $q \in Alg(\Sigma_1)$ und $q' \in Alg(\Sigma_2)$ mit $q' \, \rho \, q$. Wir betrachten die Übergänge $q' \longrightarrow \overline{q'}$ und zeigen, daß es dann einen Zustand $\overline{q}$ gibt mit $q \xrightarrow{+} \overline{q}$ und $\rho(\overline{q'}) = \overline{q}$. Beim Beweis unterscheiden wir die Fälle anhand von $[\![ct]\!]_{q'}$.

Fall 1: $[\![ct]\!]_{q'} \in [\![\mathsf{New}]\!]_{q'}$
Auf der Zielmaschine passieren bei der Ausführung eines New-Kommandos, außer dem Weiterschalten des Befehlszählers keine Aktualisierungen.

Bei der Ausführung eines New-Kommandos auf der Quelle wird eine neue Referenz r erzeugt ($[\![\mathit{Reference}]\!]_{\overline{q}} = [\![\mathit{Reference}]\!]_q \cup \{r\}$), die entsprechend der Semantik von ASMs, von allen Elementen der Trägermengen in q verschieden ist. Es gilt $[\![\mathit{value}]\!]_{\overline{q}}([\![ct]\!]_q, [\![\mathit{reclevel}]\!]_q) = r$ und $[\![\mathit{content}(r)]\!]_{\overline{q}} = \bot$, falls ein Basisobjekt erzeugt wurde, ansonsten $[\![\mathit{content}(r)]\!]_{\overline{q}} = \mathit{undef}$. Bei der Erzeugung eines strukturierten Objekts werden zusätzlich die Komponenten mit dem Kopfobjekt verkettet. Außerdem wird der Befehlszähler weitergeschaltet.

Die Ausführung eines New-Kommandos ändert weder die Interpretation von *StackRefs* noch die Interpretation von *HeapRefs*, daher sind (7.22) und (7.23) auch für $\overline{q'}$ und $\overline{q}$ erfüllt. Da (7.21) vorher galt und New-Kommandos dabei explizit nicht berücksichtigt werden, gilt (7.21) auch für die Folgezustände. Von den für ρ relevanten Funktionen ändert sich nur die Interpretation des abstrakten Befehlszählers. Da sowohl in der Quelle, als auch im Ziel der Befehlszähler durch *proceed* weitergeschaltet wird und die Programme von Quelle und Ziel identisch sind gilt $\rho(\overline{q'}) = \overline{q}$.

Fall 2: $[\![ct]\!]_{q'} \in [\![\mathsf{NewGlobalFrame}]\!]_{q'}$
Sofern kein Speicherüberlauf auftritt, bewirkt die Ausführung eines NewGlobalFrame-Kommandos auf der Zielmaschine die folgenden Aktualisierungen:

```
value(ct,reclevel) := initstack
tos := initstack ⊞ framesize(type(ct))
toh := initheap
globals := initstack
proceed
```

Die Adresse des Laufzeitkellers wird initialisiert und der aktuelle Kellerpegel wird um die Größe der Schachtel für die globalen Variablen erhöht.

$$[\![tos]\!]_{\overline{q'}} = [\![initstack \boxplus framesize(type(ct))]\!]_{q'}$$

initstack berechnet die Basisadresse des Kellers, die laut Voraussetzung auch korrekt ausgerichtet ist. Gleichung (7.7) sichert zu, daß alle globalen Objekte auf dem Laufzeitkeller Platz

haben und es gilt $[\![globals]\!]_{\overline{q'}} = [\![initstack]\!]_q$. Wegen (7.3) ist sichergestellt, daß für jede globale Variable (also für jede Komponente der Schachtel) eine eineindeutige Relativadresse berechnet wird. Daher ist $[\![globals]\!]_{\overline{q'}} \boxplus [\![reladdr]\!]_{\overline{q'}}([\![type(ct)]\!]_{q'}, id)$ ein eindeutiger Zugriffspfad für eine globale Variable id. Da der Speicher nicht initialisiert war, kann über die Inhalte von Kelleradressen nichts ausgesagt werden.

$$\forall id \in [\![type(ct)]\!]_{q'} : [\![content]\!]_{\overline{q'}}([\![globals]\!]_{\overline{q'}} \boxplus [\![reladdr]\!]_{\overline{q'}}([\![type(ct)]\!]_{q'}, id)) = \bot$$

In $\mathcal{A}_1$ werden folgende Aktualisierungen ausgeführt.

```
extend Reference with r
  type(r) :=c (t)
  value(ct,reclevel) := r
  do forall t : (t ∈ components(type(ct)))
    r.(snd(t)) := value(ct.(snd(t)),reclevel)
  enddo
  globals := r
endextend
proceed
```

Es wird ein neues strukturiertes Objekt erzeugt, das durch eine neue Referenz r repräsentiert ist. Die Teilobjekte werden mit den Namen der globalen Variablen ($snd(t)$) assoziiert, die neue Referenz wird gespeichert $[\![globals]\!]_{\overline{q}} = r$, und der Befehlszähler wird weitergeschaltet. Nach Durchführung dieser Aktualisierungen gibt es für alle globalen Variablen id einen eindeutigen Zugriffspfad über $[\![globals]\!]_{\overline{q}}$ und es gilt:

$$\forall id \in [\![type(ct)]\!]_q : [\![content]\!]_{\overline{q}}([\![globals.id]\!]_{\overline{q}}) = \bot$$

Die Ausführung des NewGlobalFrame-Kommandos sorgt dafür, daß die durch *StackRefs* berechneten Referenzen auf dem Laufzeitkeller um die Referenzen für die globalen Objekte erweitert werden. Die Konstruktion der Adreßabbildung garantiert, daß es zu jeder globalen Variable, die in der Quelle ein globales Objekt identifiziert, auch eine entsprechende Anfangsadresse in der Zielmaschine gibt, die ein entsprechendes Objekt referenziert. Dies gilt auch für alle Teilobjekte, der globalen Variablen. Da vor Ausführung des aktuellen Kommandos der Quell- und Zielspeicher modulo $\bot$ gleich waren, sind (7.22) und (7.23) erfüllt und es gilt $\rho(\overline{q'}) = \overline{q}$.

Fall 3: $[\![ct]\!]_{q'} \in [\![\mathsf{NewLocalFrame}]\!]_{q'}$
Die Ausführung eines NewLocalFrame-Kommandos auf der Zielmaschine bewirkt

```
value(ct,reclevel) := NewStackAddr(tos, type(ct))
tos := NewStackAddr(tos, type(ct)) ⊞ framesize(type(ct))
locals(reclevel) := NewStackAddr(tos, type(ct))
proceed
```

Auf dem Laufzeitkeller wird eine neue Basisadresse erzeugt und der Kellerpegel wird entsprechend erhöht. Die neue Basisadresse für lokale Variablen ist:

$$[\![locals(reclevel)]\!]_{\overline{q'}} = [\![NewStackAddr(tos, type(ct))]\!]_{q'}$$

Wegen (7.8) ist die neue Basisadresse in einem Bereich der nicht benutzt wird. Anforderung (7.3) stellt sicher, daß für jede lokale Variable (also für jede Komponente der Schachtel) eine eineindeutige Relativadresse berechnet wird und sich die Adressen nicht überlappen. Wegen

(7.4) sind die Adressen auch korrekt ausgerichtet. Gleichung (7.7) sichert zu, daß alle Objekte auf dem neuen Stück im Laufzeitkeller Platz finden. Daher ist

$$[\![locals(reclevel)]\!]_{\overline{q'}} \boxplus [\![reladdr]\!]_{\overline{q'}}([\![type(ct)]\!]_{q'}, id)$$

ein eindeutiger Zugriffspfad für eine lokale Variable *id*.

In $\mathcal{A}_1$ wird ein neues strukturiertes Objekt erzeugt und die Teilobjekte werden mit den Namen der lokalen Variablen (*snd(t)*) assoziiert. Zusätzlich wird die Referenz in *locals(reclevel)* gespeichert und der Befehlszähler wird weitergeschaltet. Nach Durchführung dieser Aktualisierungen identifiziert der Zugriffspfad $[\![locals(reclevel).id]\!]_{\overline{q}}$ für alle lokalen Variablen *id* der aktuellen Funktion f ein eindeutiges Objekt. Die ASM-Regel zur Interpretation eines NewLocalFrame-Kommandos steht in Anhang B. Für diese Objekte gilt:

$$\forall id \in [\![type(ct)]\!]_q : [\![content(locals(reclevel).id]\!]_{\overline{q}}) = \bot$$

Die Argumentation ist ansonsten analog zu Fall 2.

Fall 4: $[\![ct]\!]_{q'} \in [\![$ NewParamFrame $]\!]_{q'}$
Die Ausführung eines *NewParamFrame*-Kommandos auf der Zielmaschine bewirkt

```
tos := NewStackAddr(tos, type(ct)) ⊞ framesize(type(ct))
acttos(reclevel) := tos
params(reclevel+1) := NewStackAddr(tos, type(ct))
proceed
```

Auf dem Laufzeitkeller wird eine neue Basisadresse erzeugt und der Kellerpegel wird entsprechend erhöht. Die neue Basisadresse für die Parameter der nächsten gerufenen Funktion ist:

$$[\![params]\!]_{\overline{q'}}([\![reclevel + 1]\!]_{q'}) = [\![NewStackAddr(tos, type(ct))]\!]_{q'}$$

Wegen (7.8) ist die neue Basisadresse in einem Bereich der nicht benutzt wird. Anforderung (7.3) stellt sicher, daß für jeden Parameter der nächsten gerufenen Funktion (also für jede Komponente der Parameterschachtel) eine eineindeutige Relativadresse berechnet wird. (7.7) sichert zu, daß alle Objekte auf dem neuen Stück im Laufzeitkeller Platz finden. Daher ist

$$[\![params(reclevel + 1)]\!]_{\overline{q'}} \boxplus [\![reladdr]\!]_{\overline{q'}}([\![type(ct)]\!]_{q'}, id)$$

ein eindeutiger Zugriffspfad für Parameter *id*. Zusätzlich wird der alte Kellerpegel in *acttos* gesichert.

Auf der Quellmaschine werden die gleichen Aktualisierungen wie für NewLocalFrame durchgeführt, mit der Ausnahme, daß anstatt *locals(reclevel)* nun *params(reclevel+1)* aktualisiert wird. Nach der Durchführung dieser Aktualisierungen existiert eine neue Parameterschachtel. Die Parameter des Aufrufers werden weiterhin über $[\![params(reclevel)]\!]_q$ zugegriffen, Zuweisungen an die neuen Parameter erfolgen über $[\![ct]\!]_q$ (vgl. statische Semantik von *AL*). $[\![params(reclevel + 1).id]\!]_{\overline{q}}$ identifiziert für alle Parameter *id* der nächsten gerufenen Funktion f ein eindeutiges Objekt und es gilt:

$$\forall id \in [\![type(ct)]\!]_q : [\![content(params(reclevel + 1).id]\!]_{\overline{q}}) = \bot$$

Die Argumentation ist ansonsten analog zu Fall 3 und 4.

Fall 5: $[\![\, ct \,]\!]_{q'} \in [\![\, \mathsf{Call} \,]\!]_{q'}$
Bei einem Funktionsaufruf passieren im Quell- und im Zielprogramm mit einer Ausnahme die gleichen Aktualisierungen (vgl. Abb. 7.6 und 7.7). Auf der Zielmaschine wird noch der aktuelle Kellerpegel um die Größe einer Adresse erhöht. Damit haben wir schon vorgesehen, daß an dieser Stelle die Rücksprungadresse gespeichert werden kann. Die für ρ relevanten Funktionen betrifft dieser Unterschied nicht.

Fall 6: $[\![\, ct \,]\!]_{q'} \in [\![\, \mathsf{Return} \,]\!]_{q'}$
Beim Rücksprung aus einer Funktion wird der Befehlszähler sowohl in der Quell- als auch in der Zielmaschine auf den Nachfolger des Aufruf-Kommandos gesetzt. Die Identität der Programme sichert zu, daß dann $[\![\, ct \,]\!]_{\overline{q}} = [\![\, ct \,]\!]_{\overline{q'}}$ gilt.

Auf der Zielmaschine werden die Basisadressen vergessen, mit denen bisher auf die lokalen Variablen und Parameter der Funktion zugegriffen wurde. Der Kellerpegel wird auf den Stand zurückgesetzt, der vor dem Erzeugen der Parameterschachtel und der Schachtel für die lokalen Variablen galt.

```
tos := acttos(reclevel-1)
```

Zugriffe in den Keller erfolgen immer unterhalb des Kellerpegels[2]. Aus diesem Grund sind die lokalen Variablen und Parameter auf der Zielmaschine nicht mehr zugreifbar, nachdem aus der Funktion zurückgesprungen wurde.

Eine entsprechende Aktion passiert auf der Quellmaschine. Die lokalen Variablen und die aktuellen Parameter werden vom Keller genommen, indem die Aufruftiefe um 1 reduziert wird, und die Referenzen auf die entsprechenden Schachteln werden weggeworfen.

```
params(reclevel) := undef
locals(reclevel) := undef
```

Damit existieren sie zwar noch in der Sorte Reference, aber die Objekte können nicht mehr zugegriffen werden, da der Zugriff auf Kellerobjekte nur über die Schachteln aus *locals, globals* und *params* möglich ist. Es gilt

$$[\![\, StackRefs \,]\!]_{q'}) = [\![\, StackRefs \,]\!]_q) \setminus ([\![\, ObjRefs(params(reclevel)) \,]\!]_q \cup [\![\, ObjRefs(locals(reclevel)) \,]\!]_q$$

Mit der obigen Argumentation über den Laufzeitkeller gilt (7.22) auch für $\overline{q'}$ und $\overline{q}$. Wegen $\rho(q') = q$ folgt dann sofort $\rho(\overline{q'}) = \overline{q}$. Wir haben ρ bewußt so definiert, daß die Speicherfunktionen von Quelle und Ziel nur auf den Objekten in Beziehung stehen, an die schon einmal ein Wert zugewiesen wurde. Während die Quellmaschine jedes Mal einen undefinierten Inhalt eines neu erzeugten Objekts festlegt, kann dies auf der Zielmaschine sehr wohl ein definierter Wert sein, der an eine ehemals existierende Kellerschachtel zugewiesen wurde.

Fall 7: $[\![\, ct \,]\!]_{q'} \in [\![\, \mathsf{Alloc} \,]\!]_{q'}$
In $\mathcal{A}_2$ werden $[\![\, ct.size \,]\!]_q$ Elemente und das Kopfobjekt in einem Schritt erzeugt, indem Platz für ein Objekt der Größe $[\![\, objsize(type(ct.elements), ct.size) \,]\!]_{q'}$ auf der Halde reserviert wird. Die Anforderungen (7.10)–(7.12) garantieren, daß genügend Platz auf der Halde reserviert

2) Werden neue Schachteln für Parameter oder lokale Variablen erzeugt, dann wird der Kellerpegel entsprechend erhöht. Wird der Kellerpegel beim Rücksprung aus einer Funktion erniedrigt, dann werden die Basisadressen ebenfalls zurückgesetzt.

wird, um $[\![ct.size]\!]_q$ Elemente aufzunehmen, die alle korrekt ausgerichtet und nacheinander im Speicher liegen. Die Anforderung (7.9) an *NewHeapAddr* garantiert, daß das erste Element korrekt ausgerichtet ist und die neue Anfangsadresse über dem bisherigen Haldenpegel liegt. Damit haben wir eine injektive Funktion, die Haldenreferenzen aus AL auf Speicheradressen abbildet.

Ein Alloc-Kommando auf der Quellmaschine erzeugt zuerst ein Kopfobjekt und anschließend in einer Schleife $[\![value(ct.size, reclevel)]\!]_q$ Teilobjekte vom Typ $[\![type(ct.elements)]\!]_q$, die mit dem Kopfobjekt verknüpft werden. In einem letzten Schritt wird der Initialzustand des Alloc-Kommandos wieder hergestellt, damit die nächste Ausführung wieder korrekt ablaufen kann.

Sei nun $q, \ldots, q_0, \ldots, q_{[\![ct.size]\!]_q - 1}, \ldots, \overline{q}$ eine Zustandsfolge und $\overline{q}$ sei der Zustand in dem

$$[\![ct.numofallocated]\!]_{\overline{q}} = [\![ct.size]\!]_q$$

gilt, dann gilt $\rho(\overline{q'}) = \overline{q}$.

Der Nachweis, daß die Ausführung von Alloc auf der Zielmaschine genügend Speicher ausfaßt, um alle erzeugten Objekte eines Aufrufs auf der Quellmaschine unterzubringen, wird formal durch Induktion über den Aufbau des Typterms geführt, der sich aus $[\![type(ct.elements)]\!]_{q'}$ und $[\![value(ct.size, reclevel)]\!]_{q'}$ ableiten läßt. $[\![ct.elements]\!]_{q'}$ ist dabei ein New-Kommando bzw. ein Teilgraph, falls ein strukturiertes Objekt erzeugt werden soll.

Fall 8: $[\![ct]\!]_{q'} \in [\![\mathsf{LoadBasic}]\!]_{q'}$
Da $\rho(q') = q$ gilt, ist (7.21) erfüllt. Mit (7.22) folgt dann für $\overline{q}$ und $\overline{q'}$ sofort (7.21) und damit $\rho(\overline{q'}) = \overline{q}$.

Fall 9: $[\![ct]\!]_{q'} \in [\![\mathsf{LoadStruct}]\!]_{q'}$
Die Argumentation in diesem Fall unterscheidet sich nicht von Fall 8. Auf der Zielmaschine wird die Speicheradresse für den Zugriff auf eine Komponente eines strukturierten Objekts berechnet, indem zu der Basisadresse des Objekts die Relativadresse des entsprechenden Selektors addiert wird. Ein Zugriff auf eine Komponente eines AL-Objekts passiert in der Quelle über das Kopfobjekt und einen Selektor. Da die Zuordnung von Referenzen zu Adressen genau so definiert ist und die Inhalte von Objekten auf der Quelle und dem Ziel übereinstimmen, gilt anschließend auch

$$[\![value]\!]_{\overline{q'}}([\![ct]\!]_{q'}, [\![reclevel]\!]_{q'}) = [\![value]\!]_{\overline{q}}([\![ct]\!]_q, [\![reclevel]\!]_q)$$

und damit $\rho(\overline{q'}) = \overline{q}$. Es macht dabei keinen Unterschied, ob Keller- oder Haldenobjekte zugegriffen werden.

Fall 10: $[\![ct]\!]_{q'} \in [\![\mathsf{Store}]\!]_{q'}$
Auf der Zielmaschine können Werte in mehreren Speicherworten codiert sein. Daher wird die Zuweisung an ein Speicherobjekt mit Hilfe des Makros $\overline{content}$ Byte für Byte realisiert. Die Funktion *content* der Zielmaschine liest entsprechend dem Typ der Adresse die passende Anzahl Bytes aus und setzt den Wert zusammen.

In AL werden Werte ungeteilt an ein Speicherobjekt zugewiesen Laut Voraussetzung ist die Codierung der Werte korrekt. Vor der Ausführung des Store-Kommandos stehen die Werte (inklusive Referenzen) von Kommandos gemäß (7.21) in Beziehung. In der Quelle wird der Inhalt eines Objekts durch die Aktualisierung von $[\![content]\!]_q$ verändert. Daher gilt (7.22) und (7.23) über den Zuständen $\overline{q'}$ und $\overline{q}$ und damit $\rho(\overline{q'}) = \overline{q}$.

Fall 11: $[\![ct]\!]_{q'} \in [\![\mathsf{Read}]\!]_{q'} \cup [\![\mathsf{Write}]\!]_{q'}$
Bei der Ausführung eines Write-Kommandos wird an den Ausgabestrom der Wert $[\![value(ct.source, reclevel)]\!]'_q$ angehängt. Vor der Ausführung sind wegen (7.20) die Ausgabeströme und wegen (7.21) auch die Werte gleich. Deshalb folgt sofort $\rho(\overline{q'}) = \overline{q}$.

Bei der Ausführung eines Read-Kommandos wird das erste Element des Eingabestroms als Wert des Kommandos zugewiesen und aus dem Strom entfernt. Das passiert sowohl auf der Quelle als auch im Ziel. Da vorher (7.19) und (7.21) erfüllt waren, gilt das auch hinterher.

Fall 12: $[\![ct]\!]_{q'} \in [\![\mathsf{BasicOps}]\!]_{q'} \cup [\![\mathsf{One}]\!]_{q'} \cup [\![\mathsf{All}]\!]_{q'}$
Da bei der Abarbeitung eines Kommandos aus den verbleibenden Kommandotypen der Speicher nicht modifiziert wird, sondern nur Werte verarbeitet werden und diese gemäß (7.21) gleich sind, hat die Speicherabbildung keine Auswirkungen auf die ursprüngliche Semantik. Wenn vorher die Relation auf den entsprechenden Zuständen galt, dann stehen auch die Folgezustände in Relation.

Falls kein Speicherüberlauf auftritt, dann definiert die Relation ρ, abgesehen von der Ausführung des Alloc-Kommandos, eine 1:1-Simulation von $\mathcal{A}_1$ durch $\mathcal{A}_2$. Da die Ausführung von Alloc nicht beobachtbar ist gilt: $\mathcal{A}_2 \prec_\rho \mathcal{A}_1 \Rightarrow \mathcal{A}_2 \prec_{\overline{\rho}} \mathcal{A}_1$. ♢

Weitere Schritte der Speicherabbildung

Die in den vorigen Abschnitten beschriebene Transformation der Semantik führt uns in den Speicherzustandsraum der Zielmaschine. Mit der Speicherabbildung ist auch eine Transformation von Programmen verbunden. In *MIS* gibt es keine expliziten Instruktionen für Zugriffe in den Speicher. Stattdessen hat man Adreßausdrücke und typabhängige Zuweisungen an Speicherzellen. Als nächste Transformation würden wir daher das Programm entsprechend abbilden. Wir übersetzen Load-Kommandos in Adreßausdrücke und Store-Kommandos in typabhängige *MIS*-Zuweisungen. Die Transformation von Store ist in Abschnitt 8.2.5 angegeben.

In einem zweiten Übersetzungsschritt sorgen wir für die explizite Speicherung von Zwischenergebnissen. In der beschriebenen Zwischensprache sind Zwischenergebnisse noch an Kommandos gebunden. Die Funktion *value* repräsentiert diese Zwischenergebnisse für unterschiedliche Instanzen einer Funktion. Zwischenergebnisse werden in der neuen Zwischensprache als temporäre Variablen explizit auf dem Laufzeitkeller abgelegt. Dieser Übersetzungsschritt hat keine Auswirkung auf schon existierende Relativadressen, sofern die temporären Variablen ans Ende einer Prozedurschachtel angehängt werden. Damit ändert sich zwar die Größe der Prozedurschachtel, aber schon zugeordnete Relativadressen müssen nicht verändert werden.

Zusätzlich eliminieren wir mit einer weiteren Transformation die Funktion *caller* und speichern Rücksprungadressen im Laufzeitkeller. Den Platz für die dafür benötigte Speicheradresse hatten wir bei der Adreßabbildung schon reserviert. Für die Korrektheit dieser Transformation müssen wir nachweisen, daß bei der Abarbeitung von Funktionen diese Speicherstellen nicht verändert werden. Ansonsten könnten wir nicht zeigen, daß sich die Rücksprungadressen bei der Ausführung von Return in der Quelle und dem Ziel nicht unterscheiden.

7.4.3 Programmtransformation: Beispiel While-Schleife

Am Beispiel der While-Schleife demonstrieren wir den zweiten Schritt unseres allgemeinen Beweisschemas. Hier kommen die Übersetzungstheoreme aus Kapitel 6 zur Anwendung. Die Montage in Abbildung 7.8 definiert die Semantik einer Schleife unter der Annahme, daß es

Break- und Continue-Anweisungen gibt. Die statische Semantik besagt, daß die Ausdrücke,

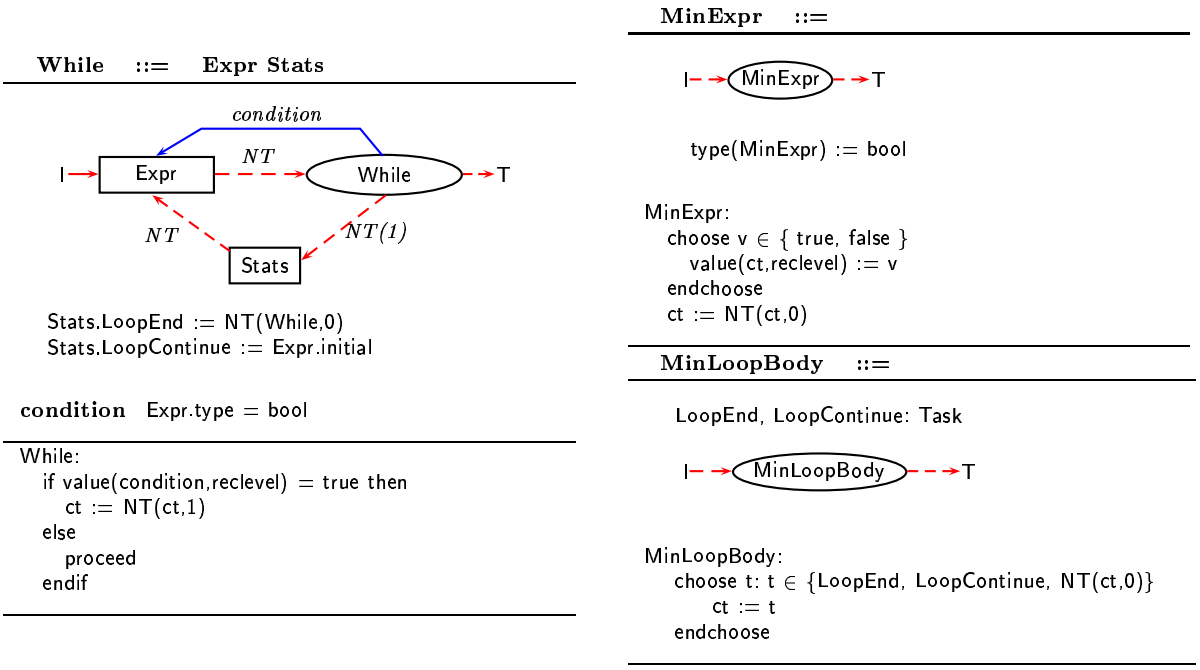


Abbildung 7.8: Semantik einer While-Schleife mit Minimalanforderungen MinExpr und MinLoopBody

die als Schleifenbedingung eingesetzt werden, vom Typ *bool* sind. Für eine lokale Semantik der Schleife benötigen wir eine Definition der minimalen Semantik von Expr und Stats. Die beiden Montages in Abbildung 7.8 definieren die entsprechenden ASMs $\mathcal{A}_{\mathsf{MinExpr}}$ und $\mathcal{A}_{\mathsf{MinLoopBody}}$. Wir gehen davon aus, daß die Quell- und die Zielsprache wohlgeformt sind und (6.5) bzw. (6.6) gelten. Weisen wir noch die Korrektheit der Übersetzung von While nach, dann können wir unter der Voraussetzung, daß die Teilkonzepte Stats und Expr ebenfalls korrekt übersetzt werden, Satz 6.19 anwenden. Damit folgt, daß jede wohlgeformte Instanz von While korrekt übersetzt wird.

Die Abbildung 7.9 zeigt die Übersetzung von While. Die Übersetzung führt ein zusätzliches statisches Attribut *key* für den Schleifenrumpf ein. Dieses Attribut ist eine zusätzliche Anforderung an den Schleifenrumpf. Für die minimale Semantik des Rumpfs der Zielprogrammschleife haben wir daher die Minimalanforderung MinLoopBody′, die sich nur in dem statischen Attribut *key* von MinLoopBody unterscheidet. *key* ist ein Element von $\Sigma_{[\![\mathsf{Stats}]\!]}$, wird aber nicht in $I_{[\![\mathsf{Stats}]\!]}$, sondern durch den Kontext der Verwendung von Stats definiert.

Wir zeigen zuerst, daß der durch die obige Transformation beschriebene AL-Graph eine korrekte strukturerhaltende Übersetzung von $\mathsf{While}(E < [\![\mathsf{MinExpr}]\!], S < [\![\mathsf{MinLoopBody}]\!])$ ist.

Satz 7.2 (Schleife)
$[\![\mathsf{AL\_While}(\mathcal{A}_{\mathsf{MinExpr}}, \mathcal{A}_{\mathsf{MinLoopBody'}})]\!] \prec_{\overline{\rho}} [\![\mathsf{While}(\mathcal{A}_{\mathsf{MinExpr}}, \mathcal{A}_{\mathsf{MinLoopBody}})]\!]$.

Beweis Die Transformation impliziert einen Signaturmorphismus $\sigma : \Sigma_{\mathsf{While}} \to \Sigma_{\mathsf{AL\_While}}$. Es gilt:

$$\forall f \in F_{\Sigma_{\mathsf{While}}} : \sigma(f) = f \quad \text{und} \quad \forall s \in S_{\Sigma_{\mathsf{While}}} : \sigma(s) = \begin{cases} \mathsf{One}, & \text{falls } s = \mathsf{While} \\ s, & \text{sonst} \end{cases}$$

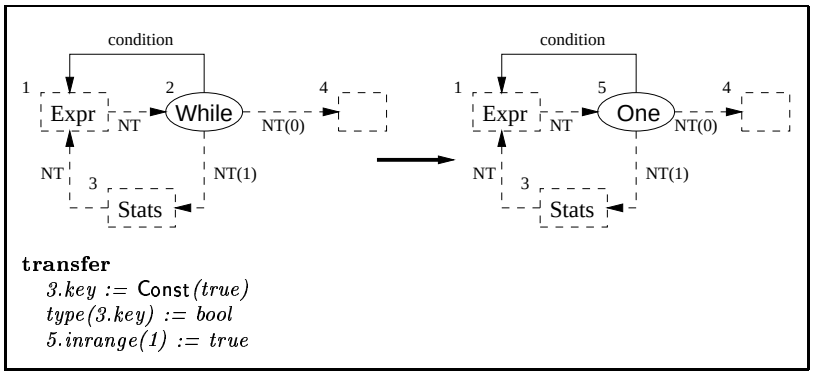


Abbildung 7.9: Transformation von While nach AL_While

und außerdem werden das Attribut *key* und die Funktion *inrange* eingeführt, die zur Interpretation von One benötigt werden. Die Interpretation dieser statischen Funktionen ist durch zusätzliche Gleichungen spezifiziert. Für alle Terme $x = c_{\mathsf{While}}(o_x, \mathsf{expr}, \mathsf{stats}, \mathsf{while})$ gilt

$$\begin{aligned} [\![\, \mathsf{stats}.\mathit{initial}.\mathit{key} \,]\!] &= [\![\, \mathsf{Const}(\mathit{true}) \,]\!] \\ [\![\, \mathit{type}(\mathsf{stats}.\mathit{initial}.\mathit{key}) \,]\!] &= [\![\, \mathsf{bool} \,]\!] \\ [\![\, \mathsf{while} \,]\!] &\in [\![\, \mathsf{One} \,]\!] \\ [\![\, \mathit{inrange} \,]\!]([\![\, \mathsf{while} \,]\!], [\![\, 1 \,]\!]) &= \mathit{true} \end{aligned}$$

Die Simulationsbeziehung wird durch Induktion über die Anzahl der Zustandsübergänge der Zielmaschine gezeigt. Zuerst definieren wir wieder die Relation ρ über den Zuständen von Quelle und Ziel. Im folgenden sprechen wir von $\mathcal{A}_{\mathsf{While}}$ als $\mathcal{A}_1$ und von $\mathcal{A}_{\mathsf{AL\_While}}$ als $\mathcal{A}_2$.

Ein Zustand $q' \in \mathit{Alg}(\Sigma_2)$ steht in Beziehung ρ zu einem Zustand $q \in \mathit{Alg}(\Sigma_1)$, wenn

$$\forall s \in S_{\Sigma_1} \forall t \in [\![\, s \,]\!]_q : t \in [\![\, \sigma(s) \,]\!]_{q'} \tag{7.24}$$

$$\forall f \in F_{\Sigma_1} : [\![\, f \,]\!]_q = [\![\, f \,]\!]_{q'} \tag{7.25}$$

Induktionsanfang: $n = 0$
Im Initialzustand i von $\mathcal{A}_1$ bzw. i' von $\mathcal{A}_2$ gilt $[\![\, ct \,]\!]_i = [\![\, ct \,]\!]_{i'} \in [\![\, \mathsf{MinExpr} \,]\!]$ und mit den Gleichungen der Transformation sind (7.24) und (7.25) erfüllt, daher gilt $i' \,\rho\, i$.

Induktionshypothese: Die Behauptung gilt für Zustandsfolgen der Länge n in $\mathcal{A}_2$.

Induktionsschluß: $n \curvearrowright n+1$
Sei $q \in \mathit{Alg}(\Sigma_1)$ und $q' \in \mathit{Alg}(\Sigma_2)$ mit $q' \;\rho\; q$. Wir betrachten wieder die Übergänge $q' \longrightarrow_{(2)} \overline{q'}$ und zeigen, daß es dann einen Zustand $\overline{q}$ gibt mit $q \longrightarrow_{(1)} \overline{q}$ und $\overline{q'} \;\rho\; \overline{q}$. Beim Beweis unterscheiden wir die Fälle anhand von $[\![\, ct \,]\!]_{q'}$.

Fall 1: $[\![\, ct \,]\!]_{q'} = [\![\, ct \,]\!]_q \in [\![\, \mathsf{MinExpr} \,]\!]_q$
Dieser Fall kann außer als Initialzustand auch während der Interpretation auftreten. Die Ausführung eines MinExpr-Kommandos berechnet für $[\![\, ct \,]\!]_{q'}$ indeterministisch einen Wert aus Bool. Nach der Ausführung gilt:

$[\![\, ct \,]\!]_{\overline{q'}} = [\![\, ct.NT(0) \,]\!]_{q'}$ und
$[\![\, \mathit{value} \,]\!]_{\overline{q'}}([\![\, ct \,]\!]_{q'}, [\![\, \mathit{reclevel} \,]\!]_{q'}) = \mathit{true}$ oder $[\![\, \mathit{value} \,]\!]_{\overline{q'}}([\![\, ct \,]\!]_{q'}, [\![\, \mathit{reclevel} \,]\!]_{q'}) = \mathit{false}$

Eine entsprechende Berechnung macht auch die Quellmaschine. Sie definiert zwei mögliche Ausführungsfolgen mit jeweils unterschiedlichen Werten *true* bzw. *false* für

$[\![value]\!]_{\overline{q}}([\![ct]\!]_q, [\![reclevel]\!]_q)$. Da ct auf der Quell- und der Zielmaschine gleich weitergeschaltet wir, ist (7.25) erfüllt und es gilt $\overline{q'}\,\rho\,\overline{q}$.

Fall 2: $[\![ct]\!]_{q'} \in [\![\mathsf{One}]\!]_{q'}$ und $[\![ct]\!]_q \in [\![\mathsf{While}]\!]_q$
Die Semantik des One-Kommandos unterscheidet zwei Möglichkeiten:

1. $[\![ct.condition]\!]_{q'} = true = [\![ct.NT(1).key]\!]_{q'} \Rightarrow [\![ct]\!]_{\overline{q'}} = [\![ct.NT(1)]\!]_{q'}$:
   In diesem Fall gilt aber auch $[\![ct.condition]\!]_q = true$ und damit $[\![ct]\!]_{\overline{q}} = [\![ct.NT(1)]\!]_q$. Da $[\![ct.NT(1)]\!]_{q'} = [\![ct.NT(1)]\!]_q$ laut Voraussetzung, gilt $\overline{q'}\,\rho\,\overline{q}$.
2. $[\![ct.condition]\!]_{q'} = false \Rightarrow [\![ct]\!]_{\overline{q'}} = [\![ct.NT(0)]\!]_{q'}$: Bei der Interpretation von $[\![ct]\!]_q$ wird der Befehlszähler auf $[\![ct.NT]\!]_q$ gesetzt, was laut Voraussetzung gleich $[\![ct.NT(0)]\!]_{q'}$ ist. Daher gilt $\overline{q'}\,\rho\,\overline{q}$.

Fall 3: $[\![ct]\!]_{q'} = [\![ct]\!]_q \in [\![\mathsf{MinLoopBody}]\!]_{q'}$
Die Argumentation ist analog zu Fall 1, nur können durch MinLoopBody zwei unterschiedliche Aussprungspunkte anstatt unterschiedlicher Ergebnisse berechnet werden. ♢

Wir haben die While-Schleife der Quellsprache durch eine AL-Schleife ersetzt und haben damit den zweiten Übersetzungsschritt ($\mathcal{C}_2$) aus Abschnitt 6.4.3 durchgeführt. Zur Vervollständigung der Übersetzung müssen wir noch den ersten Übersetzungsschritt ($\mathcal{C}_1$) verifizieren. Wir haben zu zeigen, daß die Teilkonzepte Expr und Stats ebenfalls korrekt übersetzt werden. Weisen wir für die Transformationen Expr → AL_Expr und Stats → AL_Stats

$$[\![\mathsf{AL\_Expr}]\!] \prec_{\overline{\rho}} [\![\mathsf{Expr}]\!] \text{ und } [\![\mathsf{AL\_Stats}]\!] \prec_{\overline{\rho}} [\![\mathsf{Stats}]\!],$$

nach, dann können wir mit Satz 6.19 folgern, daß die Schleife in Abb. 7.10 einen AL-Graphen definiert, der die Semantik der Originalschleife simuliert.

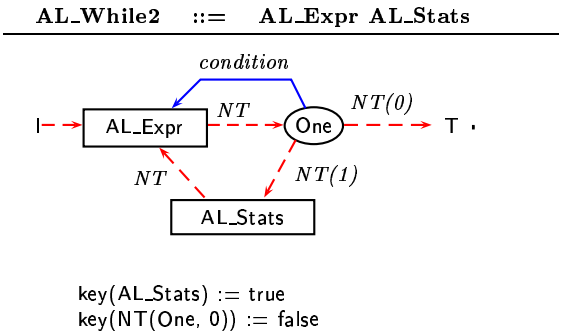


Abbildung 7.10: Semantik einer While-Schleife in AL

7.5 Zusammenfassung

Wir haben in diesem Kapitel die universelle Sprache AL in unseren Rahmen zur Konstruktion eines korrekten Übersetzers eingeordnet und haben dargelegt, wie mit Hilfe von AL die Semantik von Sprachen und korrekte Übersetzungen definiert werden können. Zusätzlich haben wir die zwei wesentlichen Techniken zur Verifikation vorgestellt und am Beispiel von zwei Transformationen illustriert.

AL bildet aus theoretischer Sicht die Syntax eines Transformationskalküls über Programmsemantiken. Semantik bekommt der Kalkül durch den ASM-Formalismus. Die Korrektheit des Kalküls wird durch den Nachweis garantiert, daß die Anwendung von Transformationsregeln eine Simulationsbeziehung zwischen altem und neuem AL-Term/-Graphen etabliert. Der Kalkül kann erweitert werden, indem neue Sprachkonzepte zu AL hinzugefügt werden, deren Semantik durch eine Abbildung nach AL definiert wird.

Aus Übersetzersicht ist AL eine Zwischensprache, die wir ausnutzen, um eine $n*m$-Beziehung bei der Übersetzung von n Quell- in m Zwischensprachen, in eine $n+m$-Beziehung umzuwandeln, indem wir Übersetzer $QS \rightarrow AL$ bzw. Übersetzer $AL \rightarrow QS$ wiederverwenden. Die Übersetzung $QS \rightarrow AL$ entspricht der formalen Semantikspezifikation der Quellsprache. Eine adäquate und einfache Formulierung dieser Spezifikation erreichen wir durch Verwendung einer Bibliothek von vordefinierten Sprachkonzepten, die die Semantik höherer imperativer Sprachkonzepte in Termen von AL beschreiben. Der AL-Kalkül repräsentiert die Spezifikationen der Bibliothek, ihre Implementierung wird in Kapitel 10 vorgestellt. Bei der Definition der Abbildung $AL \rightarrow ZS$ nutzen wir aus, daß unterschiedliche Zwischensprachen zwar unterschiedliche semantische Aspekte hervorheben, die Sprachen aber die gleichen Konzepte haben, wie sie in AL vorliegen. Daher ist die Abbildung in die Zwischensprache einfach zu definieren und zu verifizieren.

Simulationsbeweise im AL-Rahmen erfolgen in zwei Schritten. Im ersten Schritt geht man in einen gemeinsamen Speicherzustandsraum über. Sind wir mit Quell- und Zielsemantik im gleichen Zustandsraum, dann können wir leicht das Programm transformieren und die Simulationsbeziehung nachweisen. Bei der Speicherabbildung bleibt das Programm unverändert, nur der Speicherzustandsraum wird modifiziert. Wir definieren eine neue Interpretation des Programms und zeigen die Simulation von ursprünglicher und neuer Interpretation. Am Beispiel der Transformation einer generisch definierten While-Schleife haben wir die Verifikation einer Programmtransformation gezeigt. Um die Korrektheit der Übersetzung zu zeigen, haben wir die allgemeinen Übersetzungstheoreme aus dem vorigen Kapitel angewendet. Sowohl die Adreßabbildung aus Abschnitt 7.4.2 als auch die Transformation der Schleife in Abschnitt 7.4.3 stellen Konzepte der Bibliothek wiederverwendbarer Semantiken und Transformationen dar. Die Adreßabbildung ist sehr allgemein formuliert. Wir haben nur Eigenschaften definiert, die zum Nachweis der Korrektheit notwendig sind, haben die Adreßabbildungsfunktionen jedoch nicht explizit angegeben. Damit sind wir in der Lage beliebige Algorithmen zur Berechnung von Relativadressen zu verwenden, solange sie die geforderten Eigenschaften haben, und wir können durch Programmprüfung diese Eigenschaften sicherstellen. Dafür müssen wir nicht einmal die Implementierung der Algorithmen verifizieren.

8 Transformationsschemata

Innerhalb unseres Übersetzerrahmens verlangen wir, daß jede Transformation verifiziert ist, auch die Definition einer Quellsprache $[\![\,]\!] : QS \to AL$, falls eine formale Semantik mit ASMs gegeben ist. Transformationen $AL^+ \to AL$ innerhalb der Bibliothek müssen auf jeden Fall verifiziert sein. Zur Vereinfachung dieser Verifikationsaufgabe sind wir an der Wiederverwendung von Beweisen für Transformationen interessiert.

Bei genauer Betrachtung der Transformationen von Übersetzern stellt man fest, daß es bestimmte Transformationsschemata gibt, die immer wieder vorkommen. Das ist nicht weiter verwunderlich, da bei der Übersetzung von imperativen Quellsprachen in Maschinensprachen jedes Mal gleichartige Aufgaben anfallen. Komplexe Befehle müssen mit Hilfe von einfacheren Befehlen implementiert werden und Spezialbefehle der Zielmaschine implementieren bestimmte Befehlssequenzen der Quellsprache. Außerdem wird während der Übersetzung Indeterminismus eliminiert oder Berechnungen werden vertauscht. Diese Beobachtung haben wir ausgenutzt und haben bestimmte Klassen von Transformationen identifiziert und verifiziert.

Diese sprachunabhängigen Transformationen sind so allgemein formuliert, daß ihre Korrektheit nur mit den Eigenschaften des ASM-Formalismus bzw. mit den Eigenschaften der in Kapitel 5 eingeführten Klasse SASM gezeigt werden kann. Durch zusätzliche Einschränkungen der Semantikspezifikation können wir konkretere (mächtigere) Transformationsschemata definieren. Die Einschränkung der Spezifikationssprache stellt die Beziehung zu unserer konkreten Spezifikationssprache AL her. Da AL immer noch die Eigenschaften von SASM erfüllt, sind alle Sätze, die wir hier definieren, auch im Kontext von AL anwendbar. In unserem AL-Kalkül können syntaktisch mehr Transformationen abgeleitet werden, weil die Sprache syntaktische Muster in Form von Kommandos anbietet, die eine bestimmte Semantik ausdrücken.

Ein Transformationsschema muß nur ein einziges Mal verifiziert werden und kann im folgenden durch korrekte Instantiierung zur Konstruktion korrekter, konkreter Transformationen benutzt werden. Die vorgestellten generischen Transformationen sind: Dekomposition, Komposition, Vertauschung von Berechnungen bzw. Elimination von Indeterminismus und Spezialisierung.

Zu jeder sprachunabhängigen Transformation definieren und beweisen wir einen Satz, der die Korrektheit einer konkreten Instantiierung besagt und eine abstrakte Graphersetzungsregel definiert, die instantiiert werden kann. Die Verwendung eines Transformationsschemas demonstrieren wir anhand der Spezialisierung der überladen definierten Zuweisung. Wir wenden das entsprechende Transformationsschema an, um die Überladung aufzulösen.

8.1 Spezielle Eigenschaften von ASMs als Grundlage für Transformationen

In diesem Abschnitt beschreiben wir einige grundlegende Eigenschaften des ASM-Formalismus, die wir für Transformationen ausnutzen. Zusätzlich führen wir Begriffe ein, die uns die Beschreibung bestimmter Sachverhalte erleichtern.

Zustandsübergänge einer ASM $\mathcal{A} = (\Sigma, Alg(\Sigma), \rightarrow, I)$ werden durch das Ausführen von Regeln r_i einer Regelmenge $R = \{r_1, \ldots, r_n\}$ gesteuert, die $\rightarrow$ definiert, vgl. Abschnitt 4.1. Die Ausführung von Regeln in einer bestimmten Reihenfolge definiert Abhängigkeiten über den Aktualisierungen.

Beispiel 8.1 Seien f eine einstellige und x eine nullstellige Funktion (Variable). In einem Zustand q gelte $[\![f(2)]\!]_q = 5, [\![f(3)]\!]_q = 10, [\![x]\!]_q = 3$. Betrachten wir nun zwei Aktualisierungsregeln $x := 2$ und $f(x) := 2$, dann macht es einen Unterschied, ob diese Aktualisierungsregeln parallel ausgeführt werden oder sequentiell. Im parallelen Fall gilt in dem neuen Zustand q': $[\![x]\!]_{q'} = 2, [\![f(2)]\!]_{q'} = 5$ und $[\![f(3)]\!]_{q'} = 2$. Hintereinanderausführung liefert die neue Interpretation $[\![x]\!]_{q'} = 2, [\![f(2)]\!]_{q'} = 2$ und $[\![f(3)]\!]_{q'} = 10$. Das liegt daran, daß die Aktualisierung von f von der Aktualisierung von x abhängt.

Definition 8.1 (Semantische Abhängigkeit)
Eine Regel $r \equiv f(\overline{t}) := t$ hängt im Zustand q genau dann semantisch von einer Regel $r' \equiv g(\overline{u}) := u$ ab, wenn ein Teilterm s von $f(\overline{t})$ oder t existiert, der die Form $s \equiv g(\overline{v})$ hat und $[\![\overline{v}]\!]_q = [\![\overline{u}]\!]_q$ gilt. Wir schreiben $r \Leftarrow_q r'$ falls die Regel r semantisch von der Regel r' im Zustand q abhängt und $r \not\Leftarrow_q r'$, falls $(r, r') \notin \Leftarrow_q$. Wir lassen den Zustand q weg, falls die entsprechende Beziehung in allen Zuständen gilt.

Unser Ziel ist es, auf der syntaktischen Ebene Schlüsse über semantische Eigenschaften zu ziehen. Daher sind wir an einem syntaktischen Kriterium interessiert, anhand dessen semantische Abhängigkeiten erkannt werden können.

Definition 8.2 (Syntaktische Abhängigkeit von Regeln)
Eine Regel $r \equiv f(\overline{t}) := t$ hängt genau dann syntaktisch von einer Regel $r' \equiv g(\overline{u}) := u$ ab, wenn das Symbol g in $f(\overline{t})$ oder in t vorkommt.

Wir schreiben $r \leftarrow r'$, falls die Regel r von der Regel r' abhängt und $r \not\leftarrow r'$, falls $(r, r') \notin \leftarrow$.

Man sieht leicht, daß die syntaktische Unabhängigkeit $r \not\leftarrow r'$ ein hinreichendes Kriterium für die semantische Unabhängigkeit $r \not\Leftarrow r'$ ist ($r \not\leftarrow r' \Rightarrow r \not\Leftarrow r'$). Kommt das Symbol g weder auf der linken noch auf der rechten Seite von r vor, dann hat die Änderung der Interpretation von g natürlich keine Auswirkungen auf Aktualisierungen in r. Umgekehrt impliziert die semantische Abhängigkeit $r \Leftarrow_q r'$ in einem Zustand q die syntaktische Abhängigkeit $r \leftarrow r'$.

Bemerkung: Eine Alternative zu dem konservativen Kriterium für syntaktische Unabhängigkeit ist die Durchführung einer Datenflußanalyse, die semantische Abhängigkeiten durch eine Attributierung bzw. durch das Einziehen von Kanten im Programmgraphen explizit macht. So geht man üblicherweise bei der Optimierung vor. ♢

Unsere Definition von Regelabhängigkeit bzw. Unabhängigkeit läßt sich auf Regelmengen erweitern. Eine Regelmenge $R = \{r_1, \ldots, r_m\}$ hängt syntaktisch von einer Regelmenge $R' = \{r'_1, \ldots, r'_n\}$ ab, falls $r_i \leftarrow r'_j$, für ein $i \in \{1, \ldots, m\}, j \in \{1, \ldots, n\}$. Wir schreiben $R \leftarrow R'$, falls die Regelmenge R syntaktisch von R' abhängt. Semantische Abhängigkeit wird analog definiert.

Definition 8.3
Seien q und q' Zustände einer ASM und R eine Regelmenge. Wir schreiben

$$q \longrightarrow_R q'$$

wenn q' aus q durch Anwendung der Aktualisierungen von R entsteht.

Wir schreiben

$$q \longrightarrow_{R,q_1} q',$$

wenn sich q' durch Ausführung der Aktualisierungsmenge $Up(R, q_1)$ auf dem Zustand q ergibt.

Im ersten Fall findet ein Zustandsübergang nur dann statt, wenn die Aktualisierungsmenge $Up(R, q)$ konsistent ist. Mit der zweiten Notation können wir ausdrücken, daß eine Aktualisierungsmenge nicht oder nur teilweise von dem Zustand abhängt, in dem sie ausgeführt wird.

Im folgenden leiten wir einige Eigenschaften über der Ausführung von Regelmengen ab, die zum Beweisen von Transformationen nützlich sind.

Fakt 8.1 Betrachten wir die folgenden Übergänge

$$q \longrightarrow_{R \uplus R'} q_1$$
$$q \longrightarrow_{R'} q' \longrightarrow_{R,q} q_2,$$

dann gilt $q_1 = q_2$, falls $Up(R, q) \cup Up(R', q)$ konsistent ist.

Lemma 8.4
Seien R und R' zwei Regelmengen mit $R \not\leftarrow R'$ und der konsistenten Aktualisierungsmenge $Up(R, q) \cup Up(R', q)$. Betrachten wir den Übergang

$$q \longrightarrow_{R'} q', \tag{8.1}$$

dann gilt $Up(R, q) = Up(R, q')$.

Beweis Wir zeigen die Behauptung durch einen Widerspruchsbeweis. Angenommen es gilt $Up(R, q) \neq Up(R, q')$, dann bedeutet das, es gibt eine Regel $r \in R, r \equiv f(\overline{t}) := t$ mit $Up(r, q) \neq Up(r, q')$. Das kann zwei Gründe haben:

1. $((f, [\![\ \overline{t}\ ]\!]_q), [\![\, t\,]\!]_q) \in Up(r, q)$ und $((f, [\![\ \overline{t}\ ]\!]_q), [\![\, t\,]\!]_q) \notin Up(r, q')$ oder
2. $((f, [\![\ \overline{t}\ ]\!]_{q'}), [\![\, t\,]\!]_{q'}) \in Up(r, q')$ und $((f, [\![\ \overline{t}\ ]\!]_{q'}), [\![\, t\,]\!]_{q'}) \notin Up(r, q)$.

Wir betrachten nur den ersten Fall, den zweiten Fall zeigt man analog. Es gilt:

$$[\![\ \overline{t}\ ]\!]_q \neq [\![\ \overline{t}\ ]\!]_{q'} \text{ oder} \tag{8.2}$$
$$[\![\, t\,]\!]_q \neq [\![\, t\,]\!]_{q'} \tag{8.3}$$

Nach Voraussetzung gilt $r \not\leftarrow r'$ für alle $r \in R$ und $r' \in R'$. Es tritt also kein Funktionsname, dessen Interpretation durch $Up(R', q)$ aktualisiert wurde, in $f(\overline{t})$ oder in t auf. Das ist aber ein Widerspruch zu (8.2) und (8.3). ♢

Das versetzte Ausführen von zwei Regelmengen R und R' führt zum gleichen Folgezustand, wenn die Aktualisierungen jeweils auf dem gleichen Grundzustand durchgeführt werden. Die Konsistenz der Aktualisierungsmenge $Up(R, q) \cup Up(R', q)$ ist dafür ein hinreichendes Kriterium. Sind $Up(R, q)$ und $Up(R', q)$ beide inkonsistent, dann gilt trivialerweise $q_1 = q_2$.

Satz 8.5 (Faktorisierung)
Seien R und R' zwei Regelmengen mit $R \not\leftarrow R'$ und der konsistenten Aktualisierungsmenge $Up(R,q) \cup Up(R',q)$. Betrachten wir die Übergänge

$$q \longrightarrow_{R \uplus R'} q_1 \tag{8.4}$$
$$q \longrightarrow_{R'} q' \longrightarrow_{R} q_2, \tag{8.5}$$

dann gilt $q_1 = q_2$.

Beweis Wir haben zwei Fälle zu unterscheiden.

1. $Up(R,q) = Up(R,q')$: In diesem Fall können wir den Übergang (8.5) durch

   $$q \longrightarrow_{R'} q' \longrightarrow_{R,q} q_2$$

   ersetzen. Zusammen mit der Konsistenz von $Up(R,q) \cup Up(R',q)$ liefert Fakt 8.1 die Behauptung.

2. $Up(R,q) \neq Up(R,q')$: Dieser Fall kann wegen Lemma 8.4 nicht eintreten.

♢

Bemerkung: Im obigen Satz impliziert die Konsistenz von $Up(R,q)$ und $Up(R',q)$ wegen $R \not\leftarrow R'$ die Konsistenz von $Up(R,q) \cup Up(R',q)$. ♢

Ein hinreichendes Kriterium für die Konsistenz einer Regelmenge R ist, daß alle Regeln aus R paarweise von einander unabhängig sind. In diesem Fall kann keine Funktion in verschiedenen Regeln verändert werden, woraus die Konsistenz der Aktualisierungsmenge trivialerweise folgt.

Der Faktorisierungssatz kann noch verallgemeinert werden, indem lediglich semantische Unabhängigkeit in einem Zustand q gefordert wird, d. h. wir verlangen $R \not\Leftarrow_q R'$.

Satz 8.6 (Faktorisierung erweitert)
Seien R und R' zwei Regelmengen mit $R \not\Leftarrow_q R'$. Betrachten wir die Übergänge

$$q \longrightarrow_{R \uplus R'} q_1 \tag{8.6}$$
$$q \longrightarrow_{R'} q' \longrightarrow_{R} q_2 \tag{8.7}$$

und ist die Aktualisierungsmenge $Up(R,q) \cup Up(R',q)$ konsistent, dann gilt $q_1 = q_2$.

Beweis Wir gehen nach dem gleichen Schema vor, wie beim Beweis von Satz 8.5. Für $Up(R,q) = Up(R,q')$ liefert Fakt 8.1 die Behauptung. Gilt $Up(R,q) \neq Up(R,q')$, dann führen wir den Widerspruchsbeweis über die Formeln (8.2) und (8.3). Nach Voraussetzung gilt $r \not\Leftarrow_q r'$ für alle $r \in R$ und $r' \in R'$. Damit (8.2) oder (8.3) gelten, müßte ein Teilterm s von $\overline{t}$ oder t existieren, so daß $[\![s]\!]_q \neq [\![s]\!]_{q'}$ gilt. Das kann aber nur durch das Feuern von $Up(R',q)$ mit einer Regel $r' \equiv g(\overline{u}) := u$ eintreten, für die $s \equiv g(\overline{v})$ und $[\![\overline{v}]\!]_q = [\![\overline{u}]\!]_q$ gilt. Das steht aber im Widerspruch zur Voraussetzung. ♢

Bemerkung: Der Satz 8.5 ist ein Spezialfall von Satz 8.6. Wir haben Satz 8.5 deshalb extra aufgeführt, weil diese Variante für unsere Zwecke wichtiger ist. Im Normalfall wollen wir nämlich syntaktisch über die Anwendbarkeit bzw. die Korrektheit einer Transformation schließen. Der Nachweis semantischer Unabhängigkeit für beliebige Zustände q ist, wenn überhaupt, nur mit zusätzlichen Analysen möglich. Ein Mittelweg zu den Sätzen 8.5 und 8.6 wäre es,

gemischte Abhängigkeiten zu betrachten. Dabei würden syntaktische und durch Analysen erkannte semantische Abhängigkeiten berücksichtigt. ♢

Wollen wir Aussagen über Abhängigkeit oder Unabhängigkeit von ASM-Regeln machen, dann müssen wir jeder Regel, die der ASM-Formalismus definiert eine Regelmenge zuordnen, die Aktualisierungen beschreibt. Ist s eine ASM-Regel, dann schreiben wir $\mathcal{R}(s)$ für die zugehörige Menge von Aktualisierungen.

Dabei hat die Bedingung g einer bedingten ASM-Regel eine Sonderrolle. Für sie führen wir die spezielle Aktualisierungsregel $\_ := g$ ein. Der Unterschied zu einer Aktualisierungsregel ist, daß keine linke Seite existiert und damit keine andere Aktualisierungsregel von so einer Regel abhängen kann. Dagegen kann solche eine Regel sehr wohl von einer anderen Regel abhängen, wenn g in einer anderen Regel auftaucht. Mit dieser speziellen Regel wird der Tatsache Rechnung getragen, daß Bedingungen von Zustandsänderungen abhängen können, ohne selbst Zustandsänderungen zu verursachen.

Definition 8.7
Den einzelnen ASM-Regeln sind folgende Regelmengen zugeordnet:

$$\begin{aligned}
\mathcal{R}(\textbf{skip}) &\stackrel{\text{def}}{=} \{\} \\
\mathcal{R}(f(\bar{t}) := t) &\stackrel{\text{def}}{=} \{f(\bar{t}) := t\} \\
\mathcal{R}(\textbf{do in parallel } s_1, \ldots, s_n \textbf{ enddo}) &\stackrel{\text{def}}{=} \bigcup_{i=1}^{n} \mathcal{R}(s_i) \\
\mathcal{R}(\textbf{if } g \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ endif}) &\stackrel{\text{def}}{=} \mathcal{R}(s_1) \cup \mathcal{R}(s_2) \cup \{_ := g\} \\
\mathcal{R}(\textbf{do forall } v : g(v)\ s \textbf{ enddo}) &\stackrel{\text{def}}{=} \mathcal{R}(s) \cup \{_ := g(v)\} \\
\mathcal{R}(\textbf{choose } v : g(v)\ s \textbf{ endchoose}) &\stackrel{\text{def}}{=} \mathcal{R}(s) \cup \{_ := g(v)\}
\end{aligned}$$

Allgemein gilt:

$$\mathcal{R}(s_1, \ldots, s_n) \stackrel{\text{def}}{=} \bigcup_{i=1}^{n} \mathcal{R}(s_i). \tag{8.8}$$

Die Transformationen, die wir im folgenden beschreiben, sind theoretisch auf beliebige ASMs anwendbar, sofern sie die geforderten Eigenschaften erfüllen. In unserem Rahmen betrachten wir jedoch nur ASMs der Klasse SASM (siehe Abschnitt 5.1). Wenn wir im folgenden von ASMs reden, dann sind immer ASMs der Klasse SASM gemeint. Diese Annahme vereinfacht die Definition von Voraussetzungen für die Anwendbarkeit von Transformationsmustern und erspart uns viele Nebenbedingungen, die wir für allgemeine ASMs angeben müßten.

8.2 Transformationsschemata

Wir definieren Transformationsschemata auf ASMs und beweisen deren Korrektheit. Die Informationen, die zum Nachweis der Korrektheit einer Transformation benötigt werden, können teilweise auf syntaktischer Ebene gewonnen werden, teilweise müssen komplexere Analysen durchgeführt werden bzw. der Übersetzerbauer muß auf semantischer Ebene nachweisen, daß bestimmte Eigenschaften erfüllt sind. Jede der im folgenden eingeführten und verifizierten

Transformationen der Semantik hat ihre Entsprechung in realen Übersetzern, oder es gibt Übersetzungen, die eine Kombination dieser Transformationen implementieren.

8.2.1 Dekomposition

Die erste Transformation, die wir besprechen, kommt sehr häufig bei der Abbildung von Quell- in Zwischensprachen vor. In unserem Fall vor allem bei der Abbildung einer Quellsprache nach *AL*. Auf der Ebene der Quellsprache gibt es komplexe Sprachkonzepte, die bei der Übersetzung in weniger komplexe, maschinennahe Konzepte abgebildet werden müssen. In unseren ASMs entspricht dies einer Aufspaltung komplexer Kommandos, d. h. Kommandos bei deren Interpretation eine größere Menge von Aktualisierungen durchgeführt werden. Die genaue Ausprägung dieser Aufspaltung interessiert uns nur am Rande. Unser Dekompositionsschema definiert die Anforderungen an eine korrekte Aufspaltung.

In Abbildung 8.1 ist die Dekomposition als Graphersetzungsregel beschrieben. In der Abbildung entsprechen sich die mit den gleichen Zahlen benannten Kanten der linken und rechten Seite. Damit drücken wir aus, daß die Daten- bzw. Steuerflußinformation in diesem Falle gleich ist. Das heißt, der Anfang der Datenflußkante mit der Nummer eins ist auf beiden Seiten identisch, das Ziel der Datenflußkante und der Steuerflußkanten (1 und 2) ebenso.

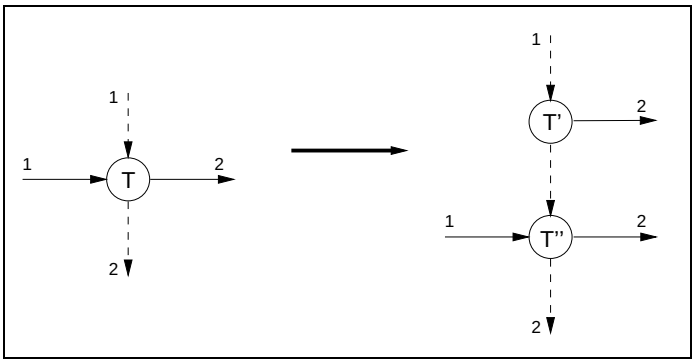


Abbildung 8.1: Graphersetzungsregel für Dekomposition

Betrachten wir nun eine ASM $\mathcal{A}$ mit einem Zustand q und der folgenden ASM-Regel für Kommandos aus T.

```
if T(ct) then
    do in-parallel
     rules
     eval
     proceed
    enddo
endif
```

Wir verwenden hier bestimmte Konventionen bzw. Makros zur Darstellung von ASM-Regeln.

- *rules* steht für beliebige Regeln, die in dem jeweiligen Kontext erlaubt sind. Wollen wir Regeln unterscheiden, dann schreiben wir $rules_i$.
- *eval* ist ein Makro und beschreibt eine Regel, in der lediglich die Funktion *value* aktualisiert wird. *value* wird nicht in einer mit *rules* bezeichneten Menge aktualisiert. Kommt

also *eval* in einer Regel nicht vor, dann bedeutet das, *value* wird nicht aktualisiert.

- Das Makro *proceed* beschreibt eine Aktualisierung von ct, also das Weiterschalten des abstrakten Befehlszählers. Tritt *proceed* nicht auf, dann wird ct nicht aktualisiert.

Als Voraussetzung für die Anwendung der Dekompositionstransformation verlangen wir, daß es keine Funktion f in *rules* bzw. *eval*, *proceed* gibt, die in einer der Regelmengen direkt von ct abhängt und gleichzeitig in der anderen Regelmenge in Abhängigkeit von ct aktualisiert wird. In diesem Fall sprechen wir von **Unabhängigkeit der Regelmengen in Bezug auf** ct.

Seien nun $R_1 \equiv \mathcal{R}(rules)$ und $R_2 \equiv \mathcal{R}(eval, proceed)$ und R_2 ist syntaktisch unabhängig von R_1, $R_2 \not\leftarrow R_1$. Dann besagt Satz (8.5), daß die Ausführung von R_1 und die anschließende Ausführung von R_2 die gleiche Zustandsänderung zur Folge hat, wie die Ausführung von $\mathcal{R}(rules, eval, proceed)$.

Betrachten wir die Dekomposition als Übersetzungsschritt von einer Sprache L_1 in eine andere Sprache L_2, deren Semantiken durch ASMs $\mathcal{A}_1$ bzw. $\mathcal{A}_2$ definiert sind, dann entspricht diese Übersetzung einer Transformation von $\mathcal{A}_1$. Wir definieren im folgenden diese Transformation und weisen die Korrektheit nach, indem wir zeigen, daß die resultierende ASM $\mathcal{A}_2$ die ASM $\mathcal{A}_1$ simuliert.

Transformationsschema 1 (Dekomposition) Wir konstruieren die ASM $\mathcal{A}_2$ aus $\mathcal{A}_1$. Am Anfang gilt $\mathcal{A}_2 = \mathcal{A}_1$, inklusive des Programms. Sei nun $\mathcal{A}_1$ in einem Zustand q, in dem gilt:

$$[\![ct]\!]_q = x, x \in [\![T]\!]_q$$

Der Zustand q' von $\mathcal{A}_2$ leitet sich ab, indem wir zur Trägermenge von $\mathcal{A}_2$ zwei neue Elemente y_1, y_2 hinzunehmen und gleichzeitig die Interpretationen von ct, *value* und *data* im Zustand q' anpassen:

$$[\![ct]\!]_{q'} \stackrel{\text{def}}{=} y_1 \tag{8.9}$$

$$[\![value]\!]_{q'}(y_2) \stackrel{\text{def}}{=} [\![value]\!]_q(x) \tag{8.10}$$

$$[\![nexttask]\!]_{q'}(t,i) \stackrel{\text{def}}{=} \begin{cases} [\![nexttask]\!]_q(t,i), & \text{falls } t \neq y_1, y_2 \ \wedge \ [\![nexttask]\!]_q(t,i) \neq x \\ y_1, & \text{falls } [\![nexttask]\!]_q(t,i) = x \\ y_2, & \text{falls } t = y_1 \ \wedge \ i = 0 \\ undef, & \text{falls } t = y_1 \ \wedge \ i > 0 \\ [\![nexttask]\!]_q(x,i), & \text{falls } t = y_2 \end{cases} \tag{8.11}$$

$$[\![data]\!]_{q'}(t,i) \stackrel{\text{def}}{=} \begin{cases} [\![data]\!]_q(t,i), & \text{falls } t \neq y_1, y_2 \ \wedge \ [\![data]\!]_q(t,i) \neq x \\ y_2, & \text{falls } [\![data]\!]_q(t,i) = x \\ [\![data]\!]_q(x,i), & \text{falls } t = y_1, y_2 \end{cases} \tag{8.12}$$

(8.13)

Gibt es noch eine Funktion f, die vom Wert von ct abhängt, dann gilt dafür

$$[\![f]\!]_{q'}(y_1) = [\![f]\!]_{q'}(y_2) \stackrel{\text{def}}{=} [\![f]\!]_q(x). \tag{8.14}$$

Wird die Interpretation einer Funktion f im Zustand q' für neue Elemente y_i nicht explizit definiert, dann gilt: $[\![f]\!]_{q'}(y_i) = undef$. ♢

Das Programm von $\mathcal{A}_2$ wird nun um zwei Regeln erweitert:

```
if T'(ct) then                          if T''(ct) then
  do in-parallel                          do in-parallel
    rules                       und         eval
    ct := nexttask(ct,0)                    proceed
  enddo                                   enddo
endif                                   endif
```

Dabei definieren T' und T'' neue Kommandosorten, die bestimmte Aufgaben modellieren:

$$[\![\, \mathsf{T}' \,]\!]_{q'} \stackrel{\text{def}}{=} \{y_1\} \text{ und } [\![\, \mathsf{T}'' \,]\!]_{q'} \stackrel{\text{def}}{=} \{y_2\}.$$

Wir können jetzt folgenden Satz formulieren.

Satz 8.8 (Dekomposition)
Die Anwendung des Transformationsschemas 1 auf eine ASM $\mathcal{A}_1$, liefert eine ASM $\mathcal{A}_2$ mit $\mathcal{A}_2 \prec_{\overline{\rho}} \mathcal{A}_1$.

Beweis Zum Nachweis der Simulation müssen wir die Basisfunktionen von $\mathcal{A}_1$ und $\mathcal{A}_2$ in Beziehung setzen. Beobachtbare Funktionen sind in unseren ASMs nur auf Elementen definiert, die nicht zu einem Kommandouniversum gehören. Daher können wir $\rho \subseteq Alg(\Sigma_2) \times Alg(\Sigma_1)$ wie folgt definieren:

$$\begin{aligned} q' \,\rho\, q \Leftrightarrow \;& [\![\, \mathsf{Task} \,]\!]_{q'} = [\![\, \mathsf{Task} \,]\!]_q \cup \{y_1, y_2\} \;\wedge\; (8.10)\text{–}(8.14) \;\wedge \\ & ([\![\, ct \,]\!]_q = x \Rightarrow [\![\, ct \,]\!]_{q'} = y_1) \;\wedge\; ([\![\, ct \,]\!]_q \neq x \Rightarrow [\![\, ct \,]\!]_{q'} = [\![\, ct \,]\!]_q) \end{aligned}$$

Die Anwendung von Satz 8.5 besagt die Korrektheit des getrennten Feuerns von R_1 und anschließend R_2. Die Kommandotypen T' und T'' modellieren genau die Aktualisierungen von R_1 bzw. R_2. Bleibt noch zu zeigen, daß Steuer- und Datenfluß richtig transformiert wurden.

Betrachten wir zuerst die Regeln für y_1. Die Regeln von *rules* und $ct := nexttask(y_1, 0)$ müssen konsistent sein. Das ist aber trivialerweise erfüllt, da ct laut Voraussetzung nicht in *rules* aktualisiert wird. Der Steuerfluß wird durch $ct := nexttask(y_1, 0)$ immer auf y_2 weitergeschaltet. Bei der Ausführung von y_2 wird der Wert von x berechnet, da dort die Aktualisierungen von *eval* ausgeführt werden. Das bedeutet, daß der Datenfluß von x nach y_2 umgeleitet werden muß. (8.12) im Dekompositionsschema beschreibt genau diese Umlenkung.

Problematisch kann jetzt noch sein, daß in T Funktionen in Abhängigkeit von ct aktualisiert werden können. In diesem Fall führt die Aufspaltung in zwei Kommandos unter Umständen zu Inkonsistenzen, da ct unterschiedliche Kommandos bezeichnet, je nachdem, ob die Aktualisierung in einem Kommando aus T' oder T'' durchgeführt wird. Diesen Fall haben wir jedoch schon ausgeschlossen, indem wir als Voraussetzung für die Anwendung der Dekomposition die Unabhängigkeit der Funktionen in *rules* und *eval* von ct gefordert haben. Für die Funktion *value* haben wir dieses Problem nicht, da die Definition für ASMs aus der Klasse SASM verlangt, daß *value* nur über die Datenflußfunktion *data* Werte von ct abfragt und *value* nur in *eval* verändert wird. ♢

Bemerkung: Das Dekompositionsschema beschreibt den grundlegenden Fall. Es läßt sich jedoch leicht verallgemeinern. Wir können erlauben, daß *eval* auch in der ersten Regel ausgeführt wird, wenn die Datenflußkanten entsprechend umgehängt werden. Außerdem können wir *rules* in $rules_1$ und $rules_2$ auftrennen, die in T' bzw. T'' ausgeführt werden. Wir müssen

nur darauf achten, daß für die entsprechenden Regelmengen R_1 und R_2 gilt $R_2 \not\leftarrow R_1$ und die Einschränkungen bzgl. *ct* erhalten bleiben. Der Beweis bleibt auch gültig, wenn wir Aktualisierungen beobachtbarer Funktionen berücksichtigen und diese *rules* bzw. komplett entweder $rules_1$ oder $rules_2$ zuordnen. ♢

8.2.2 Komposition

Die Komposition von Kommandos ist das Gegenstück zu der im vorigen Abschnitt eingeführten Dekomposition. Wir benötigen sie für Transformationen, die mehrere Grundoperationen zu einer komplexen Operation zusammenfassen. Dazu gehören beispielsweise bestimmte Speicherzugriffe, die von der Maschine effizient implementiert werden. Abbildung 8.2 beschreibt die entsprechende Graphersetzungsregel und gibt eine Intuition für die Transformation.

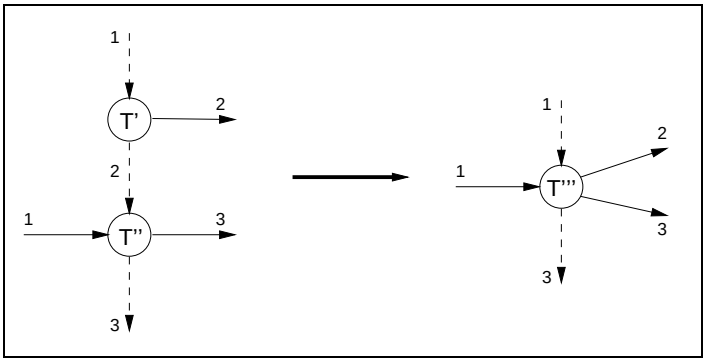


Abbildung 8.2: Graphersetzungsregel für die Komposition

Wir betrachten also zwei Kommandos, deren Semantik durch

```
if T'(ct) then
  do in-parallel
    rules
    ct := nexttask(ct,0)
  enddo
endif
```

und

```
if T''(ct) then
  do in-parallel
    eval
    proceed
  enddo
endif
```

beschrieben ist. Als Voraussetzung für die Anwendung der Kompositionstransformation verlangen wir, daß $R_2 \not\leftarrow R_1$ für $R_1 \equiv \mathcal{R}(rules)$ und $R_2 \equiv \mathcal{R}(eval, proceed)$. q sei ein Zustand der ASM $\mathcal{A}_1$ der folgende Voraussetzungen erfüllt:

1. $[\![ct]\!]_q = x_1, x_1 \in [\![\mathsf{T}']\!]_q$.
2. $[\![nexttask]\!]_q(x_1, i) = x_2, x_2 \in [\![\mathsf{T}'']\!]_q$.
3. $\forall x \in [\![\mathsf{T}]\!]_q, \forall i \in \mathrm{N} : [\![data]\!]_q(x, i) \neq x_1$, d. h. x_1 ist nicht über eine Datenflußkante erreichbar.
4. $\forall x \in [\![\mathsf{T}]\!]_q \setminus \{x_1\}, \forall i \in \mathrm{N} : [\![nexttask]\!]_q(x, i) \neq x_2$, d. h. x_2 ist nur von x_1 über eine Steuerflußkante erreichbar.
5. Die Regeln *rules*, *eval* und *proceed* sind konsistent im Zustand q.
6. Keine Funktion f, mit *ct* als Parameter, tritt gleichzeitig in *rules* und *eval* auf. Die Funktion *nexttask* wird dabei nicht berücksichtigt.

Dann können wir aus $\mathcal{A}_1$ eine neue ASM $\mathcal{A}_2$ mit Anfangszustand q' und den neuen Kommandos T''' konstruieren mit der $\mathcal{A}_2 \prec_{\overline{p}} \mathcal{A}_1$ gilt. $\mathcal{A}_2$ ist eine Erweiterung von $\mathcal{A}_1$ und es gilt für T''':

```
if T'''(ct) then
  do in-parallel
    rules
    eval
    proceed
  enddo
endif
```

Wir leiten $\mathcal{A}_2$ nach folgendem Schema aus $\mathcal{A}_1$ ab.

Transformationsschema 2 (Komposition) Wir fügen ein neues Element y zur Trägermenge von $\mathcal{A}_1$ hinzu. Dann definieren wir die Funktionen *ct*, *value* und *nexttask* und definieren damit den Zustand q' der ASM $\mathcal{A}_2$.

$$[\![ct]\!]_{q'} \stackrel{\text{def}}{=} y \tag{8.15}$$

$$[\![value]\!]_{q'}(y) \stackrel{\text{def}}{=} [\![value]\!]_q(x_2) \tag{8.16}$$

$$[\![nexttask]\!]_{q'}(t,i) \stackrel{\text{def}}{=} \begin{cases} [\![nexttask]\!]_q(t,i), & \text{falls } t \neq x_1, x_2, y \ \wedge \ [\![nexttask]\!]_q(t,i) \neq x_1 \\ y, & \text{falls } [\![nexttask]\!]_q(t,i) = x_1 \\ [\![nexttask]\!]_q(x_2,i), & \text{falls } t = y \end{cases} \tag{8.17}$$

$$[\![data]\!]_{q'}(t,i) \stackrel{\text{def}}{=} \begin{cases} [\![data]\!]_q(t,i), & \text{falls } t \neq x_1, x_2, y \ \wedge \ [\![data]\!]_q(t,i) \neq x_2 \\ y, & \text{falls } [\![data]\!]_q(t,i) = x_2 \\ [\![data]\!]_q(x_{1,i}), & \text{falls } t = y \ \wedge \ i < |[\![data]\!]_q(x_1)| \\ [\![data]\!]_q(x_2,j), & \text{falls } t = y \ \wedge i = |[\![data]\!]_q(x_1)| + j \end{cases} \tag{8.18}$$

$$[\![f]\!]_{q'}(y) \stackrel{\text{def}}{=} [\![f]\!]_q(t), \text{ falls } t = x_1, x_2 \tag{8.19}$$

Satz 8.9 (Komposition)
Die Anwendung des Transformationsschemas 2 auf eine ASM $\mathcal{A}_1$, liefert eine ASM $\mathcal{A}_2$ mit $\mathcal{A}_2 \prec_{\overline{p}} \mathcal{A}_1$, falls die Voraussetzungen 1–6 erfüllt sind.

Beweis Wegen $R_2 \not\leftarrow R_1$ und Konsistenz der Regeln *rules*, *eval* und *proceed* können wir Satz 8.5 anwenden und zeigen, daß alle Regeln, gleichzeitig ausgeführt werden können. Wir lassen dabei $ct := nexttask(ct, i)$ in T' aus, da der einzige Zweck dieser Regel, das Weiterschalten zum Kommando T'' war.

Wegen Voraussetzung 6 ist das Zusammenlegen von x_1 und x_2 wohldefiniert. Datenabhängigkeiten werden auf y umgeleitet. Das ist erlaubt, weil es keine Datenabhängigkeit von x_1 gibt und *value* nur in T'' gesetzt wird. Voraussetzung 4 sichert zu, daß es nur Steuerfluß über x_1 zum Kommando x_2 gibt. Das heißt, eine Ausführung von x_2 ohne vorherige Ausführung von x_1 ist nicht möglich. Steuerflußkanten hin zu x_1 werden durch Kanten auf y ersetzt, Kanten die von x_2 ausgehen werden zu ausgehenden Kanten von y mit dem entsprechenden Ziel, vgl. Änderung 8.17. Damit ist sicher, daß Steuer- und Datenfluß äquivalent ist und die gleichen Aktualisierungen ausgeführt werden. ♢

Bemerkung: Werden entweder in x_1 oder in x_2 beobachtbare Funktionen aktualisiert, dann bleibt die Aussage über die Komposition trotzdem gültig. Der Fall, daß beobachtbare Funk-

tionen durch beide Kommandos aktualisiert werden, ist durch die Definition von *eval* und *proceed* ausgeschlossen. ♢

8.2.3 Vertauschung von Berechnungen und Elimination von Indeterminismus

Wir können Transformationen auf ASMs der Klasse SASM auch nutzen, um Optimierungen eines Übersetzers als korrekt nachzuweisen. Im folgenden untersuchen wir die Vertauschung von Einzeloperationen, was die Grundlage für die Verschiebung von Code zur Elimination redundanter Berechnungen ist. Wir bleiben bei dieser Untersuchung absichtlich auf einer sehr allgemeinen Ebene und sprechen daher auch nicht von Vertauschung von Instruktionen innerhalb eines Grundblocks. Wir beginnen mit einer konservativen Version der Verschiebung von Einzeloperationen bzw. Kommandos und setzen die syntaktische Unabhängigkeit von zwei Regelmengen voraus, um zwei Kommandos auf einem Berechnungspfad zu vertauschen. Die Abbildung 8.3 zeigt die entsprechende Graphtransformation. Dann definieren wir die aggressivere Variante, die Daten- und Steuerflußabhängigkeiten betrachtet und daher auf eine größere Klasse von Kommandos anwendbar ist.

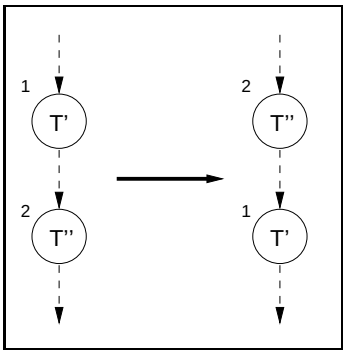


Abbildung 8.3: Graphersetzungsregel für die Vertauschung

Wir gehen von zwei Kommandos $\mathsf{T}', \mathsf{T}''$ aus

```
if T'(ct) then
   do in-parallel
    eval1
    ct:= nexttask(ct,0)
   enddo
endif
```

```
if  T''(ct) then
  do in-parallel
    obs2
    eval2
    ct:= nexttask(ct,0)
  enddo
endif
```

Für unseren ersten, konservativen Ansatz nehmen wir an, daß weder die Regeln in T' von den Regeln in T'' noch umgekehrt die Regeln in T'' von den Regeln in T' syntaktisch abhängen, d.h. für $R \equiv \mathcal{R}(eval_1)$ und $S \equiv \mathcal{R}(obs_2\ eval_2)$ gilt

$$S \not\leftarrow R \qquad \text{und} \qquad R \not\leftarrow S. \tag{8.20}$$

Wegen Satz 8.5 gilt, daß in diesem Fall die Reihenfolge der Ausführung der Regelmengen beliebig ist.

Transformationsschema 3 (Vertauschung) Wir konstruieren $\mathcal{A}_2$ aus $\mathcal{A}_1$. Am Anfang gilt $\mathcal{A}_2 = \mathcal{A}_1$ inklusive des Programms.

Sei nun q ein Zustand von $\mathcal{A}_1$ in dem folgende Bedingungen gelten:

1. $[\![ct]\!]_q = x_1$ und $x_1 \in [\![\mathsf{T}']\!]_q$.
2. $[\![nexttask]\!]_q(x_1, 0) = x_2$ und $x_2 \in [\![\mathsf{T}'']\!]_q$.
3. $\forall x \in [\![\mathsf{T}]\!]_q \setminus \{x_1\}$, $\forall i \in \mathrm{N}$ gilt: $[\![nexttask]\!]_q(x, i) \neq x_2$, d.h. x_2 ist nur von x_1 aus über eine Steuerflußkante erreichbar (vgl. Bedingung 2).
4. Die Regel $eval_1$ ist konsistent im Zustand q.
5. Die Regeln $obs_2, eval_2$ sind konsistent im Zustand q.

$\mathcal{A}_2$ im Zustand q' wird nun aus $\mathcal{A}_1$ folgendermaßen konstruiert:

$$[\![ct]\!]_{q'} \stackrel{\text{def}}{=} x_2$$

$$[\![nexttask]\!]_{q'}(t,i)) \stackrel{\text{def}}{=} \begin{cases} [\![nexttask]\!]_q(t,i), & \text{falls } t \neq x_1, x_2 \wedge \\ & \quad [\![nexttask]\!]_q(t,i) \neq x_1, x_2 \\ x_1, & \text{falls } t = x_2 \ \wedge \ i = 0 \\ [\![nexttask]\!]_q(x_2,i), & \text{falls } t = x_1 \\ x_2, & \text{falls } [\![nexttask]\!]_q(t,i) = x_1 \end{cases}$$

Satz 8.10 (Vertauschung)
Die Anwendung des Transformationsschemas 3 auf eine ASM $\mathcal{A}_1$ liefert eine ASM $\mathcal{A}_2$ mit $\mathcal{A}_2 \prec_{\overline{p}} \mathcal{A}_1$.

Beweis Betrachten wir die Änderungen an $\mathcal{A}_1$, dann ist es offensichtlich, daß $\mathcal{A}_1$ von $\mathcal{A}_2$ simuliert wird. Wir haben lediglich die Reihenfolge der Ausführung der beiden Kommandos verändert.

1. Wegen der gegenseitigen Unabhängigkeit der Regelmengen ist Satz 8.5 anwendbar und die Vertauschung hat keine Auswirkungen auf die durchgeführten Aktualisierungen.
2. Bedingung 3 schließt aus, daß es eine Möglichkeit gibt, das Kommando T'' zu betreten, ohne davor das Kommando T' auszuführen.
3. Beobachtbare Funktionen werden höchstens in einem Kommando, in unserem Fall in T'' (alternativ kann das auch in T' geschehen), aktualisiert.
4. Die Bedingungen 4 und 5 garantieren, daß sowohl die Abarbeitung von Kommando T' als auch die Abarbeitung von T'' in Zustand q konsistente Aktualisierungen bedeutet und daher die Ausführung nicht steckenbleibt.

Mit 1–4 sind alle Bedingungen für die Simulation erfüllt. ♢

Allerdings sind die Bedingungen $S \not\leftarrow R$ und $R \not\leftarrow S$ so konservativ, daß wenn wir in einem Kommando die Funktion *value* aktualisieren, diese in dem anderen Kommando nicht mehr benutzen dürfen, auch wenn es keine Datenabhängigkeit zwischen diesen beiden Kommandos gibt.

Für unsere aggressive Variante zur Vertauschung von Kommandos müssen wir mehr über die beteiligte ASM $\mathcal{A}_1$ wissen. Als erstes gehen wir dazu über, Abhängigkeiten bezüglich der Funktion ct gesondert zu betrachten. Für eine Regelmenge heißt das, daß wir nur Regeln betrachten, die ct nicht enthalten. Wir führen dazu die folgende Notation ein.

Definition 8.11
Sei $R = \{r_1, \ldots, r_n\}$ eine Menge atomarer Regeln. Wir bezeichnen mit $\overline{R}$ die Teilmenge von R, deren Regeln den Funktionsnamen ct nicht enthalten.

Im vorigen Abschnitt hatten wir $S \not\leftarrow R$ und $R \not\leftarrow S$ gefordert. Diese Forderung schwächen wir ab und verlangen

$$\overline{S} \not\leftarrow \overline{R} \qquad \text{und} \qquad \overline{R} \not\leftarrow \overline{S}.$$

Wenn wir jetzt alle Datenabhängigkeiten zwischen den beiden Kommandos ausschließen können, dann sind alle Voraussetzungen für eine korrekte Vertauschung erfüllt. In der Situation aus dem vorigen Abschnitt bedeutet dies, daß wir

$$[\![\, data \,]\!]_q(x_1, i) \neq [\![\, data \,]\!]_q(x_2, j)$$

für alle $i, j \in \mathrm{N}$ fordern. Da *data* eine statische Funktion ist, ist diese Bedingung entweder in allen Zuständen erfüllt oder in keinem.

Das weitere Vorgehen, um eine simulierende ASM $\mathcal{A}_2$ zu erhalten, ist identisch mit dem aus dem vorigen Abschnitt.

Das Transformationsschema 3 gibt uns die Möglichkeit, einzelne Kommandos in korrekter Art und Weise durch einen Kommandographen zu schieben. Nehmen wir zusätzlich an, daß ein Schema zur Elimination von redundanten Berechnungen existiert, die direkt hintereinander ausgeführt werden, dann können wir durch gezielte Anwendung dieser beiden Transformationsschemata die Elimination redundanter Berechnungen auf der Semantikebene von ASMs ausdrücken und durch die Anwendung der entsprechenden Transformationen korrekt durchführen. Ist die Vertauschung von zwei Berechnungen t_1, t_2 erlaubt, dann ist es unter Optimierungsgesichtspunkten nützlich, das auch im Programm explizit auszudrücken. Mit dem All-Kommando aus AL können wir ausdrücken, daß die Argumente in beliebiger Reihenfolge berechnet werden (siehe Abschnitt B.4) und damit mehrere Berechnungsfolgen möglich und erlaubt sind.

Bemerkung: Häufig muß nicht einmal analysiert werden, ob bestimmte Berechnungen vertauscht werden dürfen. Schon die Semantik imperativer Programmiersprachen definiert bestimmte indeterministische Berechnungen, z. B. bei der Auswertung von Parametern oder der Berechnung von Teilausdrücken. ♢

Für den Programmgraphen auf der linken Seite von Abb. 8.4 sind somit beide Ausführungen auf der rechten Seite möglich.

Transformationsschema 4 (Elimination von Indeterminismus) Wir konstruieren eine deterministische ASM $\mathcal{A}_2$ aus einer indeterministischen ASM $\mathcal{A}_1$. Am Anfang gelte $\mathcal{A}_1 = \mathcal{A}_2$. Sei nun q ein Zustand von $\mathcal{A}_1$ in dem folgende Bedingungen gelten:

1. $[\![\, ct \,]\!]_q = x, x \in \mathsf{All}$
2. $[\![\, numofall \,]\!]_q(x) = k$
3. $[\![\, nexttask \,]\!]_q(x, 1) = y_1, \ldots [\![\, nexttask \,]\!](x, k) = y_k$
4. $[\![\, nexttask \,]\!]_q(y_i, 0) = x$

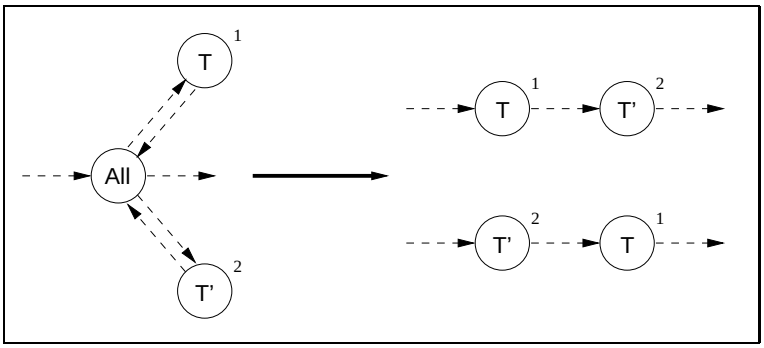


Abbildung 8.4: Mögliche Ausführungsfolgen für All

Für die ASM $\mathcal{A}_2$ im Zustand q' gilt dann:

$$\begin{aligned}
[\![\, ct \,]\!]_{q'} &\stackrel{\text{def}}{=} y_1 \\
[\![\, nexttask \,]\!]_{q'}(y_1, 0) &\stackrel{\text{def}}{=} y_2 \\
&\cdots \\
[\![\, nexttask \,]\!]_{q'}(y_{k-1}, 0) &\stackrel{\text{def}}{=} y_k \\
[\![\, nexttask \,]\!]_{q'}(y_k, 0) &\stackrel{\text{def}}{=} [\![\, nexttask \,]\!]_q(x, 0)
\end{aligned}$$

Satz 8.12 (Elimination von Indeterminismus)
Die Anwendung des Transformationsschemas 4 auf eine ASM $\mathcal{A}_1$ liefert eine ASM $\mathcal{A}_2$ mit $\mathcal{A}_2 \prec_{\overline{p}} \mathcal{A}_1$.

Beweis Unsere Korrektheitsbegriff für Übersetzungen ist so gewählt, daß für ein nichtdeterministisches Quellprogramm das Zielprogramm nicht exakt das Verhalten des Quellprogramms nachbilden muß. Vielmehr ist ein Zielprogramm eine korrekte Übersetzung, wenn es einen der möglichen Ausführungspfade des Quellprogramms implementiert.

Die Semantik von All definiert alle möglichen Permutation über der Folge $y_1, \ldots, y_k$ als erlaubte Berechnungsfolgen. Die Transformation wählt davon eine aus und erzeugt die sequentielle Ausführung von $y_1, \ldots, y_k$. ♢

Im allgemeinen Fall sind die Alternativen eines All-Kommandos durch Kommandographen beschrieben. In diesem Fall muß man für jede Alternative ein Attribut *final* berechnen, welches das letzte auszuführende Kommando des entsprechenden Teilgraphen beschreibt. Mit Hilfe dieser *final*-Attribute können wir das Schema zur Auflösung von Indeterminismus verallgemeinern.

Transformationsschema 5 (Elimination von Indeterminismus allgemein) Am Anfang gelte $\mathcal{A}_1 = \mathcal{A}_2$. Sei nun q ein Zustand von $\mathcal{A}_1$ in dem folgende Bedingungen gelten:

1. $[\![\, ct \,]\!]_q = x, x \in \mathsf{All}$
2. $[\![\, numofall \,]\!]_q(x) = k$
3. $[\![\, nexttask \,]\!]_q(x, 1) = y_1, \ldots [\![\, nexttask \,]\!](x, k) = y_k$
4. $[\![\, final \,]\!]_q(x, i) = s_i, 1 \leq i \leq k$

5. Für ein terminales Kommando t der Alternative i gilt $[\![nexttask]\!]_q(t,0) = x$

Die ASM $\mathcal{A}_2$ im Zustand q' ergibt sich nun aus $\mathcal{A}_1$ folgendermaßen.

$$\begin{aligned}
[\![ct]\!]_{q'} &\stackrel{\text{def}}{=} y_1 \\
\forall t \in [\![s_1]\!]_q : [\![nexttask]\!]_{q'}(t,0) &\stackrel{\text{def}}{=} y_2 \\
&\cdots \\
\forall t \in [\![s_{k-1}]\!]_q : [\![nexttask]\!]_{q'}(t,0) &\stackrel{\text{def}}{=} y_k \\
\forall t \in [\![s_k]\!]_q : [\![nexttask]\!]_{q'}(t,0) &\stackrel{\text{def}}{=} [\![nexttask]\!]_q(x,0)
\end{aligned}$$

8.2.4 Spezialisierung

Programmiersprachen definieren aus Gründen der Komfortabilität bzw. der Verständlichkeit überladene Operatoren. Diese Operatoren werden bei der Übersetzung in eine Zwischensprache unter Verwendung von Typinformation spezialisiert. Beispiele für überladene Operationen sind die Zuweisung, die Addition und die Subtraktion. Überladung drückt sich auch in der Definition der Semantik aus. Ist die Information, anhand der die Unterscheidung getroffen wird, statisch berechenbar, dann kann schon bei der Übersetzung die konkrete Variante in das Programm eingetragen werden. Dieses Prinzip funktioniert nicht, wenn die Unterscheidung von dynamischer Information abhängt. In diesem Abschnitt führen wir ein Schema für Spezialisierungen ein, mit dem man auf der Semantikebene die Überladungen auflösen und die Korrektheit der entsprechenden Übersetzung nachweisen kann. Eine Spezialisierung definiert im Prinzip eine partielle Evaluation einer ASM.

Wir gehen wieder von einem Kommando aus, das folgendermaßen definiert ist:

```
if T(ct) then
    do in-parallel
        rules(R)
    enddo
endif
```

Dabei sind in *rules* auch Aktualisierungen von ct, *value* oder von beobachtbaren Funktionen zugelassen. $rules(R)$ zeigt an, daß *rules* eine Regel R der folgenden Form enthält:

```
if g(ct) then
    R1
else
    R2
endif
```

Die Funktion g ist im Sinne der ASM-Semantik statisch. Das bedeutet, ihre Interpretation ändert sich bei der Abarbeitung nicht. Sie wird durch den Initialzustand der ASM definiert. Wenn die Funktion jedoch statisch ist, dann ist klar, daß sie vom Übersetzer berechnet und die Information zur Übersetzung verwendet werden kann. Die Transformationsregel in Abb. 8.5 veranschaulicht die entsprechende Transformation.

Transformationsschema 6 (Spezialisierung) Gilt in einem Zustand q der ASM $\mathcal{A}_1$

$[\![ct]\!]_q = x, x \in [\![\mathsf{T}]\!]_q$ und $[\![g]\!]_q(x) = true$,

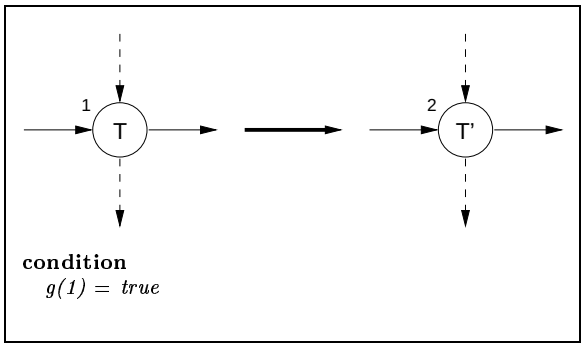


Abbildung 8.5: Graphersetzungsregel für die Spezialisierung

dann können wir eine ASM $\mathcal{A}_2$ mit Anfangszustand q' konstruieren, die $\mathcal{A}_1$ simuliert. Die neue ASM $\mathcal{A}_2$ ist eine Erweiterung der existierenden ASM $\mathcal{A}_1$, die ein zusätzliches Kommando definiert:

```
if T'(ct) then
    do in-parallel
        rules(R)  \ { R } ∪ { R1 }
    enddo
endif
```

q' unterscheidet sich von q nur an einer Stelle. Das Kommando x hat einen spezielleren Typ bekommen und ist nun Teil des neu eingeführten Universum T'.

$x \in [\![\mathsf{T}]\!]_q$ und $x \in [\![\mathsf{T}']\!]_{q'}$.

Satz 8.13 (Spezialisierung)
Die Anwendung des Transformationsschemas 6 auf eine ASM $\mathcal{A}_1$, liefert eine ASM $\mathcal{A}_2$ mit $\mathcal{A}_2 \prec_{\overline{p}} \mathcal{A}_1$.

Beweis Zum Nachweis der Korrektheit der Transformation müssen wir zeigen, daß bei Abarbeitung der neuen Regel in $\mathcal{A}_2$ genau die gleichen Aktualisierungen durchgeführt werden, wie in der ursprünglichen ASM $\mathcal{A}_1$. Das ist genau dann der Fall, wenn

$$[\![g]\!]_q(x) \stackrel{\text{def}}{=} [\![g]\!]_{q'}(x) = \mathit{true},$$

was die Voraussetzung der Transformation war. ♢

Der zweite Fall, in dem $[\![g]\!]_q(x) = \mathit{false}$ gilt und T' durch

```
if T'(ct) then
    do in-parallel
        rules(R)  \ { R } ∪ { R2 }
    enddo
endif
```

definiert ist, ergibt sich analog zum oben beschriebenen Fall.

8.2.5 Ein Beispiel für die Anwendung eines Transformationsschemas

Die Zuweisungsoperation ist in Quellsprachen normalerweise überladen definiert. Auf der Ebene der Zwischensprache, z. B. in *MIS*, will man keine überladenen Operatoren mehr haben. Das

Programm soll die Semantik eindeutig definieren, ohne Information, die zusätzlich berechnet werden muß. Daher ist der Zuweisungsoperator für jeden Typ unterschiedlich.

Wir zeigen hier die Übersetzung des Store-Kommandos aus *AL* in die entsprechenden Operationen der Zwischensprache *MIS*. Die Auflösung der Überladung mit Hilfe von, zur Übersetzungszeit berechenbarer, Typinformation ist eine Anwendung des Spezialisierungsschemas 6 aus dem vorigen Abschnitt.

Die Transformation impliziert die Einführung neuer Kommandosorten auf der ASM-Semantik. Alle Store-Kommandos im Programm werden durch neue speziellere Kommandos ersetzt. Sind in einem Programm alle Store-Kommandos ersetzt, dann kann man die entsprechende Regel und die Sorte aus der Semantik streichen. Wir erweitern also zuerst den Zustandsraum der ASM und schrumpfen ihn anschließend wieder. Die Erweiterung zusammen mit der Programmtransformation muß als korrekt bewiesen werden. Das Schrumpfen des Zustandsraums hat keine semantischen Auswirkungen, da nur die Regel zur Interpretation von *Store* eliminiert wird, für die es nach der kompletten Transformation sowieso keine Anwendung mehr gibt, da die Sorte Store keine Elemente mehr hat.

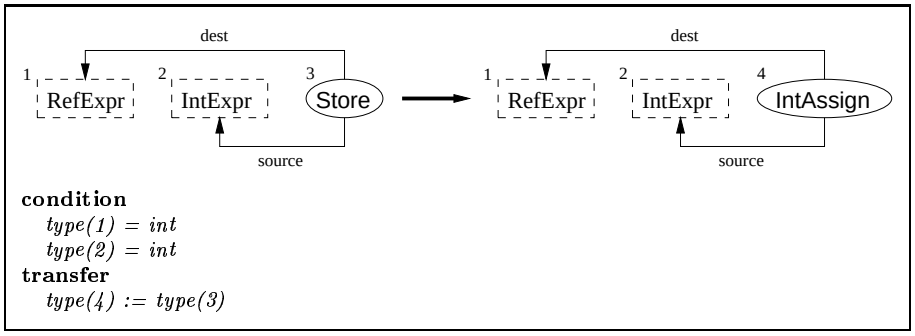
Wir gehen davon aus, daß die Speicherabbildung schon durchgeführt wurde und wir uns im Speicherzustandsraum der Zwischensprache bewegen. Desweiteren nehmen wir an, daß der Übersetzung der Zuweisung ein Transformationsschritt vorgeschaltet ist, um Zwischenergebnisse explizit auf dem Keller zu speichern. In *AL* werden Zwischenergebnisse einzelner Funktionsinstanzen durch *value(task,reclevel)* modelliert. Damit ist sichergestellt, daß bei einem rekursivem Aufruf einer Funktion die Zwischenergebnisse einer Berechnung erhalten bleiben, weil *reclevel* unterschiedlich ist. Ist eine Funktion abgearbeitet, dann wird *reclevel* dekrementiert und die Zwischenergebnisse dieser Aufruftiefe werden vergessen. Die Semantik des Store-Kommandos ist wie folgt definiert:

```
if Store(ct) then
  cont(value(ct.dest), typesize(type(ct))-1) := value(ct.source)
  proceed
endif
```

Dabei haben wir schon die Definition $\overline{content}(a,t) := e \triangleq \overline{cont}(a, typesize(t)-1) := e$ (7.2) eingesetzt. Eine äquivalente Definition der Semantikregel ist

```
if Store(ct) then
  if type(ct) = int then
    cont(value(ct.dest), typesize(int)-1) := value(ct.source)
  elsif type(ct) = float then
    cont(value(ct.dest), typesize(float)-1) := value(ct.source)
  elsif type(ct) = bool then
    cont(value(ct.dest), typesize(bool)-1) := value(ct.source)
  elsif type(ct) = addr then
    cont(value(ct.dest), typesize(addr)-1) := value(ct.source)
  endif
  proceed
endif
```

Die Äquivalenz ist leicht einzusehen. In der ersten Regel wird in einem Zustand q das Makro $\overline{content}$ jeweils mit der Größe von $[\![type(ct)]\!]_q$ als Argument interpretiert. Wir unterscheiden die Fälle anhand des Typs des Kommandos und rufen dann $\overline{cont}$ mit einer konkreten Größe auf. Damit haben wir die konkrete Semantik jedoch noch nicht im Programm explizit gemacht. Dies erreichen wir durch eine Menge von Transformationsregeln, von denen wir nur die Transformation der Zuweisung für Integer-Werte angeben.



Das Store-Kommando für die Typen *addr, float* und *bool* wird entsprechend in AddrAssign, FloatAssign und BoolAssign transformiert.

Satz 8.14 (Spezialisierung von Store)
Die Spezialisierung des Store*-Kommandos ist eine korrekte Übersetzung.*

Beweis Die Korrektheit ergibt sich durch rekursive Anwendung des Spezialisierungsschema 6 auf jedes Vorkommen eines Store-Kommandos, bis die entsprechende Semantikregel nur noch eine unbedingte Zuweisung enthält. Dies führt uns zu vier spezialisierten Regeln, die jeweils für genau einen Basistyp die dynamische Semantik beschreiben.

Die verwendeten Prädikate sind

$g_1 : type(ct) = int$
$g_2 : type(ct) = float$
$g_3 : type(ct) = bool$
$g_4 : type(ct) = addr$

Jede spezialisierte Regel der dynamischen Semantik definiert eine neue Kommandosorte IntAssign, FloatAssign, BoolAssign bzw. AddrAssign. Das sind aber genau die *MIS*-Zuweisungen. Wir geben nur die ausgerechnete Semantik von IntAssign an. Die Zuweisungen für Gleitkommazahlen, boolesche Werte und Adressen sind analog definiert.

```
if IntAssign(ct) then
  cont(value(ct.dest),typesize(int)-1) :=  value(ct.source)
  proceed
endif
```

♢

8.3 Zusammenfassung

Wir haben in diesem Kapitel Schemata für Transformationen – Komposition, Dekomposition, Vertauschung von Berechnungen bzw. Elimination von Indeterminismus und Spezialisierung – vorgestellt, die in jedem Übersetzer auftreten. Diese Transformationen haben wir auf der Basis einer sehr allgemeinen ASM mit Daten- und Steuerflußabhängigkeiten und unter Ausnutzung der Semantik des ASM-Formalismus als korrekt nachgewiesen. Wird für eine konkrete Transformation nachgewiesen, daß sie eine korrekte Instantiierung eines Transformationsschemas ist, dann folgt daraus die Korrektheit der Transformation. Das erspart den Aufwand für den häufig mühsamen Simulationsbeweis. Das Vertauschungsschema bildet die Grundlage für unterschiedliche Optimierungen, da damit auf syntaktischer Ebene bzw. in der aggressiven Version unter Berücksichtigung von Analyseinformation die Korrektheit der Verschiebung von

Code gezeigt werden kann. Ist Indeterminismus explizit im Programm ausgedrückt, dann kann unter Ausnutzung unseres Korrektheitsbegriffs ein Transformationsschema zur Elimination indeterministischer Berechnungsfolgen angewendet werden. Dieses Schema findet in Übersetzern sehr häufig Anwendung, da Zielprogramme deterministischer sind als Quellprogramme und daher bei der Übersetzung Indeterminismus aufgelöst werden muß.

Den praktischen Nutzen von Transformationsschemata haben wir am Beispiel der Übersetzung des überladenen Zuweisungsoperators beschrieben. Überladene Operatoren sind eine typische Anwendung für das Spezialisierungsschema. Die Zuweisung wurde anhand ihres Typs, der während der semantischen Analyse berechnet werden kann, in die jeweilige spezielle Store-Anweisung der Zwischensprache transformiert. Im nächsten Kapitel werden wir weitere Instanzen der hier vorgestellten Transformationsschemata kennenlernen.

9 Wiederverwendung von Sprachkonzepten und Übersetzungen

Die formale Spezifikation der Semantik realer imperativer Programmiersprachen und ihre verifizierte Übersetzung sind sehr aufwendig. Denkt man zum Beispiel an den Umfang der informellen Beschreibung von JAVA oder auch an die Verifikation der Speicherabbildung in Kapitel 7, dann ist klar, daß die intensive Wiederverwendung sowohl von Semantikspezifikationen als auch von korrekten Transformationen von großer Bedeutung ist.

Die Semantik von Programmiersprachen ist kompositionell definiert und die Konzepte, die verwendet werden, sind oft gleich oder zumindest ähnlich. Aus diesem Grund können wir die Spezifikation der Semantik einer Sprache unter Verwendung vordefinierter Konzepte zusammenstecken. Da auch die Übersetzung einer Quellsprache kompositionell über den Transformationen der Sprachkonzepte aufgebaut ist, können wir mit der Wiederverwendung von Sprachkonzepten auch deren Transformationen wiederverwenden. Die Wiederverwendung von Implementierungen geht Hand in Hand mit der Wiederverwendung der Spezifikationen, da jeder Transformationsregel eine Implementierung in einer höheren Implementierungssprache zugeordnet ist. Wir müssen nur sicherstellen, daß die Sprachkonzepte in der Implementierungssprache nach einem einheitlichen Schema repräsentiert sind. Sonst können zwar die Tranformationen auf der Spezifikationsebene zusammengesteckt werden, aber es ist nicht sicher, daß sie auch auf der Implementierungsebene zusammenpassen.

In diesem Kapitel beschreiben wir die Voraussetzungen für Wiederverwendung, führen eine Bibliothek zur Spezifikation und Transformation imperativer Programmiersprachen ein und stellen unterschiedliche Techniken zur Wiederverwendung von Sprachkonzepten vor.

Eine spezielle Art der Wiederverwendung die wir jedoch schon ausgiebig in den vorigen Kapiteln dargelegt haben, erhalten wir durch die Sprache *AL*. Wir verwenden *AL* als feste Zwischensprache und bilden alle Quellsprachen darauf ab. Damit können wir komplette Übersetzer wiederverwenden. Für m Quell- und n Zwischensprachen müssen anstatt $m*n$ nur noch $m+n$ Übersetzer gebaut werden.

9.1 Voraussetzungen für Wiederverwendung

Wir wollen einen Baukasten vordefinierter Konzepte und Transformationen zur Definition der Semantik von Programmiersprachen einsetzen und die dort definierten Konzepte zur Spezifikation von Erweiterungen benutzen. In diesem Abschnitt beschreiben wir die Voraussetzungen, die dafür nötig sind.

Die dynamische Semantik einer Sprache ist kompositionell entsprechend der abstrakten Syntax definiert. Kontextabhängige semantische Informationen werden in der semantischen Analyse eines Übersetzers als Teil der statischen Semantik berechnet und können zur Definition der dynamischen Semantik ausgenutzt werden. Die Semantik der Bibliothekskonzepte ist ebenfalls

über der abstrakten Syntax spezifiziert. Allerdings ist diese quellsprachunabhängig. Zur Spezifikation der Quellsprachsemantik wird eine Abbildung der Quellsprachsyntax auf die sprachunabhängige Syntax der Bibliothek angegeben. Diese Abbildung definiert formal die statische und dynamische Semantik, da *AL*-Elementen neben der festen Syntax auch eine feste Semantik mit ASMs zugeordnet ist. Da die Bibliothekssyntax bekannt und fix ist, können wir Transformationen vordefinieren und auch vorab verifizieren. Ist ein Quellsprachkonzept in bekannter Bibliothekssyntax beschrieben, dann können wir alle Transformationen, die auf dieser Syntax definiert sind, automatisch wiederverwenden. Die Spezifikation einer Quellsprache wird aus einer informell vorliegenden Sprachdefinition abgeleitet und hat daher definitorischen Charakter. Die Korrektheit dieser Abbildung kann nicht verifiziert, sollte aber zumindest validiert werden.

Bauen wir Sprachkonzepte durch Komposition gegebener Konzepte oder durch Instantiierung generischer Konzepte auf, dann brauchen wir konkrete Sprachkonzepte als Basisfälle der Spezifikation. Diese Basisfälle sind durch die Sprache *AL* gegeben, die wir in Kapitel 5 eingeführt haben. Die konkreten Konzepte von *AL* sind daher in der Bibliothek vorhanden. Ausgehend von *AL* haben wir zwei Richtungen für Übersetzungen. Zum einen werden imperative Quellsprachkonzepte auf *AL* abgebildet (ihre Semantik wird spezifiziert) und zum anderen wird *AL* in unterschiedliche Zwischensprachen übersetzt. In der Bibliothek tauchen also neben Quellsprachkonzepten auch Zwischen- oder Zielsprachkonzepte auf.

Wir haben folgende Anforderung an die Konzepte einer Bibliothek.

Anforderung 9.1 (Sprachkonzepte einer Bibliothek)
Ist S ein Sprachkonzept der Bibliothek, dann gibt es entweder eine Abbildung $[\![\,]\!]$, die S auf einen *AL*-Graphen abbildet oder es gibt eine Übersetzung $\mathcal{C}$, deren Ziel S ist. $[\![\,]\!]$ und $\mathcal{C}$ sind entweder per Definition korrekt oder müssen gegenüber einer existierenden formalen Semantik verifiziert sein.

Im ersten Fall kann das Sprachkonzept S zur Spezifikation der Semantik einer Sprache benutzt werden, im zweiten Fall ist S ein Zwischensprachkonzept und stellt einen Ausgang aus der Bibliothek dar. Selbstverständlich macht es auch Sinn, für ein Sprachkonzept Abbildungen in beide Richtungen zu definieren. In diesem Fall kann es als Quell- und Zwischensprachkonzept verwendet werden.

Anforderung 9.2 (Transformationen in der Bibliothek)
Alle zusätzlichen Transformationen, die in der Bibliothek vorkommen und die nicht die Semantik eines Sprachkonzepts definieren, müssen verifiziert sein. Dazu gehören z. B. optimierende Transformationen zwischen *AL*-Graphen.

Die statische Semantik eines Sprachkonzepts ist durch eine Attributierung definiert, die teils graphisch (Steuerfluß und Datenabhängigkeiten), teils textuell (z. B. der statische Typ von Kommandos) in den Montages-Spezifikationen definiert wird. Für unsere Bibliothek ist vor allem die dynamische Semantik der Sprachkonzepte von Interesse. Die meisten Attribute und Attributierungsregeln, die wir zur Spezifikation der dynamischen Semantik und zur Definition von Transformationen benötigen, sind Standard im Übersetzerbau und werden z. B. in (Waite und Goos, 1984) ausführlich beschrieben. In unseren Spezifikationen haben wir daher auf die Angabe von Attributierungsregeln verzichtet, die ausschließlich zur Weitergabe von Information dienen. Zum Beispiel braucht man für die Namens- und Typanalyse eine Art

Definitionstabelle, die als Attribut in allen Knoten des AST auftaucht. Dort werden Informationen über Namen und Typen eingetragen und bei Bedarf herausgelesen. Wir geben nur die Attributierungen an, die zur Definition der dynamischen Semantik eines Sprachkonzept notwendig sind.

Wir setzen voraus, daß die statische Semantik korrekt spezifiziert wurde und die semantische Analyse ebenfalls korrekt ist. Nach der Analysephase sind dann die Sorte Task und die folgenden statischen Funktionen definiert:

$$\begin{aligned} \mathit{nexttask}&: \mathsf{Task} \times \mathrm{N} \to \mathsf{Task} \\ \mathit{data}&: \mathsf{Task} \times \mathrm{N} \to \mathsf{Task} \end{aligned}$$

Wir verwenden in der Spezifikation $NT \mathrel{\hat{=}} \lambda\, x\ \mathit{nexttask}(x, 0)$ und benennen *data(x,i)* mit Namen wie z. B. *source* und *dest*. Zusätzlich ist eine Typisierung des Kommandographen definiert und zu jeder Benutzung eines Bezeichners ist seine Definition bekannt.

In der dynamischen Semantik verwenden wir die folgenden Funktionen, die statische semantische Informationen beschreiben:

$$\begin{aligned} \mathit{type}&: \mathsf{Task} \to \mathsf{Type} \\ \mathit{lval}&: \mathsf{Task} \to \mathsf{Bool} \\ \mathit{isglobal}, \mathit{islocal}, \mathit{isparam}&: \mathsf{Task} \to \mathsf{Bool} \end{aligned}$$

type beschreibt den statischen Typ eines Kommandos. *lval* wird für den Zugriff auf Objekte benötigt und zeigt an, ob die Referenz oder der Inhalt eines Objekts zugegriffen werden soll. Die Prädikate *isglobal, islocal* und *isparam* zeigen an, ob ein Objekt lokal, global oder ein Parameter ist, sie sind auch für Schachteln definiert. Kommandos können weitere Attribute haben, die zur statischen Semantik gehören, die hier aber nicht aufgeführt sind. Ihr Zweck wird aber durch die Benennung klar und im Einzelfall geben wir eine Attributierungsregel an.

Wir haben zwar die statische Information durch ASM-Funktionen modelliert. Aber letztendlich können wir alle diese Funktionen als Anfragen an eine Datenbank ansehen, die das *Environment* implementiert und während der semantischen Analyse aufgebaut wird. Diese Datenbank liefert dann zu einer Task die gewünschte Information. Die Art der Information, die von der Datenbank zur Verfügung gestellt wird, unterscheidet sich natürlich für unterschiedliche Sprachen. Zum Beispiel hat man für eine Sprache wie PASCAL, die geschachtelte Funktionen erlaubt, andere Scoping-Regeln als in C.

9.2 Die Bibliothek von Sprachkonzepten

Für die Spezifikation einer Quellsprache verwenden wir die abstrakte Syntax der Konzepte aus der Bibliothek. Die abstrakte Syntax, die statische und die dynamische Semantik dieser Konzepte ist jeweils durch eine Montages-Spezifikation erklärt. Die meisten Sprachkonzepte der Bibliothek sind generisch definiert.

Im folgenden stellen wir generische Montages-Spezifikationen für die üblichen Klassen imperativer Sprachkonzepte vor. Bestimmte Klassen von Konzepten, wie z. B. die Definition strukturierter Datentypen mit den entsprechenden New−, Load− und Store-Operationen, geben wir nur exemplarisch an. Eine ausführlichere Beschreibung der Bibliothekskonzepte in Zusammenhang mit der Spezifikation und Übersetzung der imperativen Sprache *IS* (DOLD ET AL.,

1996a; HEUZEROTH und HEBERLE, 1998) findet sich in (HEUZEROTH, 1998). In unseren Spezifikationen verwenden wir alle Funktionen und Kommandos, die von *AL* definiert werden. Wir dürfen sie ohne explizite Angabe einer Semantik verwenden, da diese vordefiniert ist und die *AL*-Kommandos ein Teil der Bibliothek sind. Einzelne Montages führen neue Kommandosorten ein. In diesem Fall werden auch die entsprechenden ASM-Regeln zur Spezifikation der dynamischen Semantik angegeben.

9.2.1 Funktionen

Zur Definition der Semantik von Funktionen und Prozeduren sind die Elemente ausreichend, die *AL* für die Spezifikation zur Verfügung stellt. Wir benutzen die dynamischen Funktionen:

globalframe: CreateFrame
localframe: N → CreateFrame
parframe: N → CreateFrame
caller: N → Task
reclevel: → N.

globalframe enthält die Objekte zu globalen Variablen. *localframe* und *parframe* modellieren zusammen mit der Aufruftiefe *reclevel* den Laufzeitkeller. Das Universum Reference modelliert den Speicher. Es ist wie in *AL* (Anhang B) partitioniert in Keller- und Haldenreferenzen. Obwohl wir hier nur Kellerobjekte beschreiben, können Haldenobjekte mit dem Alloc-Kommando von *AL* ebenfalls erzeugt werden.

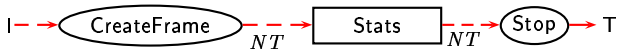
Ein imperatives Programm besteht im allgemeinen Fall aus einer Menge von Typdefinitionen, den globalen Variablen, Funktions- und Prozedurdefinitionen und dem Hauptprogramm. Erlaubt eine Quellsprache z. B. nicht die Definition zusätzlicher Typen, dann würde man diesen Teil bei der Abbildung der Quellsprache auf die abstrakte Syntax der Bibliothek einfach leerlassen. Die Montage für Programme ist in Abb. 9.1 zu sehen.

TypeDecls definiert eine Zuordnung von Namen zu Typen und hat nur Auswirkungen in der statischen Semantik. Wir gehen daher nicht näher darauf ein. *VarDecls* definiert eine Menge von Paaren $\langle Typ, Name \rangle$ und legt damit den Typ des CreateFrame-Kommandos fest, der die Erzeugung der Kellerschachteln für globale bzw. lokale Variablen beschreibt. Ein Programm definiert Funktionen und Prozeduren, $PFDecls ::= (Function|Procedure)^*$. Funktionen und Prozeduren unterscheiden sich nur darin, daß bei einer Prozedur kein Resultat existiert. Abgesehen davon, daß das Hauptprogramm genau ein einziges Mal aufgerufen wird, unterscheidet sich ein Programm in der dynamischen Semantik nicht von einer Funktion bzw. Prozedur. Bei der Ausführung eines Programms werden erst die globalen Variablen erzeugt, dann werden die Anweisungen ausgeführt, wobei eine ausgezeichnete Stop-Anweisung das Programmende markiert. Am Ende einer Funktion oder Prozedur wird stattdessen zum Aufrufer zurückgesprungen. Wir sprechen im folgenden nur noch von Funktionen, meinen damit aber immer auch eine entsprechende Prozedur.

Unser Bibliothekskonzept zur Modellierung von Funktionen definiert zusätzlich Parameter und ein Resultat (*Type*). Damit kann es auch zur Modellierung der Semantik von Programmen eingesetzt werden, denen beim Aufruf Parameter übergeben werden und die mit einem Ergebnis terminieren.

Die Definition einer Funktion führt ein neues Paar *(Funktionstyp, Name)* ein. Die Umgebung verwaltet die entsprechende Typinformation. Wir modellieren Anfragen zum Typ einer Funktion mit *proctable* : String → Type Ein Funktionstyp besteht aus einem Typ für die Parameter

Program ::= TypeDecls VarDecls PFDecls Stats

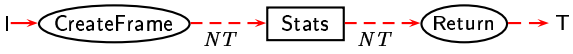



```
proctable := PFDecls.type

type(CreateFrame) := type(VarDecls)
pfname(CreateFrame) := 'main'
globalframe := CreateFrame

CreateFrame:
    if not allocated(ct) then
      extend Reference with r
        type(r) := type(ct)
        content(r) := r
        value(ct,reclevel) := r
      endextend
      allocated(ct) := true
    elsif |{ c | c ∈ components(type(ct)) ∧ notexecuted(c) }| = 0 then
      do forall c : (c ∈ components(type(ct)))
        notexecuted(c) := true
        value(ct,reclevel).snd(c) := value(fst(c).New,reclevel)
      enddo
      allocated(ct) := false
      proceed
    else
      choose c : (c ∈ { co | co ∈ components(type(ct)) ∧ notexecuted(co) })
        notexecuted(c) := false
        ct := fst(c).New
      endchoose
    endif
```

Function ::= Identifier Params Type VarDecls Stats




```
type(Function) := pair(fun(type(Params),type(Type),CreateFrame),
                       Identifier.string)

type(CreateFrame) := type(VarDecls)
pfname(CreateFrame) := Identifier.string
localframe := CreateFrame
firsttask(Identifier.string) := CreateFrame
```

Abbildung 9.1: Die Semantik eines Programms und einer Funktion

und das Resultat sowie einem Verweis auf das erste Kommando der Funktion. *firsttask* liefert zu einem Funktionsbezeichner das entsprechende Anfangskommando. Zusätzlich verwenden wir die Projektionsfunktionen *argument* und *result*.

$$\begin{aligned} \mathit{fun} &: \mathsf{Type} \times \mathsf{Type} \times \mathsf{Task} \to \mathsf{Type} \\ \mathit{firsttask} &: \mathsf{String} \to \mathsf{Task} \\ \mathit{argument}, \mathit{result} &: \mathsf{Type} \to \mathsf{Type} \end{aligned}$$

Falls eine Programmiersprache überladene oder polymorphe Funktionen erlaubt, dann würde man *firsttask* anstatt als Funktion als Relation definieren. Allerdings muß man dann bei einem Funktionsaufruf, wo *firsttask* zur Identifikation des Funktionsanfangs verwendet wird, eine entsprechende Auflösungsstrategie angeben.

Eine Funktion kann unterschiedliche Parameterarten definieren, die sich in unterschiedlichen statischen Typen spezieller Variablen ausdrücken (Abb. 9.2). Der Typ wird durch ein Tripel beschrieben.

$$\begin{aligned} \mathit{ptupel} &: \mathsf{Type} \times \mathsf{String} \times \mathit{PARAMTYPE} \to \mathsf{Type} \\ \mathit{PARAMTYPE} &\triangleq \{VAR, VAL, OUT, INOUT\} \end{aligned}$$

Der dritte Parameter beschreibt die Art des Parameters. In der Modellierung der dynamischen Semantik benötigen wir nur die Unterscheidung zwischen Wert- und Referenzsemantik, da dies für den Zugriff auf Parameter wichtig ist. Wir benötigen in der Bibliothek keine spezielle Se-

ValParDecl ::= Identifier Type

```
if type(Type) ∈ RefType then
   type(ValParDecl) := ptupel(type(Type),Identifier.string,VAR)
else
   type(ValParDecl) := ptupel(type(Type),Identifier.string,VAL)
endif
```

VarParDecl ::= Identifier Type

```
type(VarPArDecl) :=
            ptupel(type(Type),Identifier.string,VAR)
```

Abbildung 9.2: Die Semantik von Parameterdeklarationen

mantik für *Out*- und *In-/Out*-Parameter, da diese durch Komposition existierender Konzepte modelliert werden können. Einen *In-/Out*-Parameters würde man als Referenzparameter modellieren, der wie ein Werteparameter verwendet wird und an den vor einem Rücksprung aus der Funktion noch ein Wert zugewiesen wird. Ein *Out*-Parameter würde man als zusätzlichen Return-Wert modellieren. Programmiersprachen definieren Typen mit Werte- und Referenzsemantik. Ein Werteparameter, dessen Typ Referenzsemantik hat, wird als Referenz-Parameter modelliert.

Eine Anweisungsliste (Abb. 9.3) besteht aus einzelnen Anweisungen, die im Steuerfluß sequentiell verkettet sind. Die leere Anweisung (EmptyStats) haben wir in unserer Montage nicht aufgeführt, weil sie in der Konstruktion nur eine Steuerflußkante definiert. Die minimalen Anforderungen an die Semantik von Anweisungen und Anweisungslisten unterscheiden sich nicht und sind durch die gleiche Minimalanforderung *BoundStats* beschrieben. Sie besagt, daß eine Anweisung normal oder mit einem Fehler terminieren kann. Im Fehlerfall kann auf ein beliebiges Kommando verzweigt werden, ansonsten wird mit dem nächsten Kommando im Steuerfluß fortgefahren.

Bemerkung: Die Minimalanforderungen sagen nichts über die Art der Berechnungen und ihre Details aus. Das ist auch nicht erwünscht, da sonst die Minimalanforderung mehr Eigenschaften verlangen würde als nötig. ♢

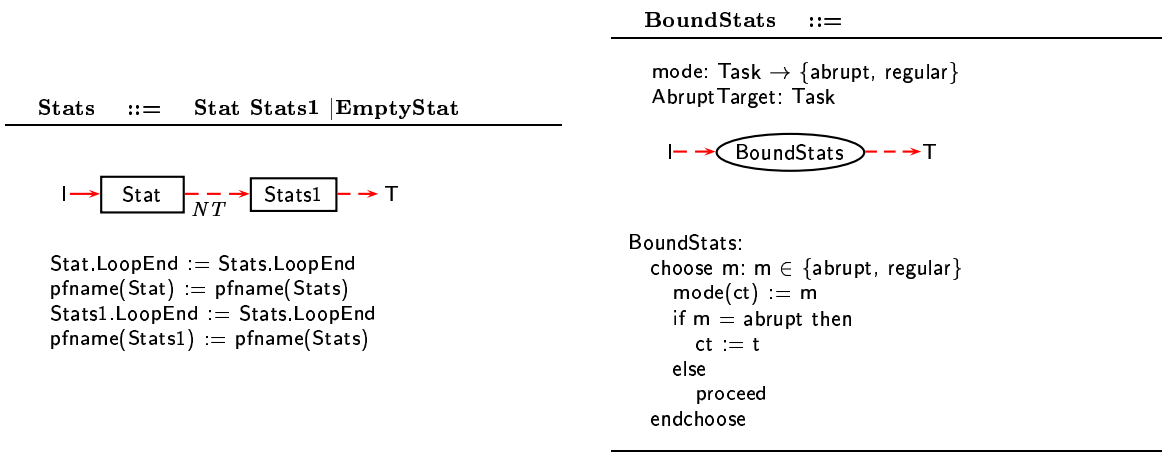


Abbildung 9.3: Die Semantik von Anweisunglisten

9.2.2 Datentypen

Die Semantik der *AL*-Basistypen ist parametrisiert. Da wir auch einen Parameter haben, der einen Datentyp als Maschinendatentyp definiert, können wir die Semantik aller üblichen Basisdatentypen durch *AL*-Basistypen spezifizieren. Die Spezifikation legt dann nur noch die Parameter fest. New und Copy beschreiben die Semantik der Erzeugung und des Kopierens eines Basisobjekts (Abb. 9.4). Die Semantik entspricht der *AL*-Semantik.

```
BasicType ::= Int | Float | Bool | Char
New(BasicType):
  extend Reference with r
    type(r) := type(ct)
    content(r) := ⊥
    value(ct,reclevel) := r
  endextend
  proceed

Copy(BasicType):
  content(value(ct.dest,reclevel)) :=
            content(value(ct.source,reclevel))
  ct := rettask
```

Abbildung 9.4: Die Definition von Basistypen

Imperative Sprachen haben Reihungen und Verbunde mit Werte- und Referenzsemantik. Wir stellen hier nur statische Reihungen mit Wertesemantik vor (Abb. 9.5). Die anderen Typen werden entsprechend modelliert. In unterschiedlichen Programmiersprachen ist die Indizierung unterschiedlich, z. B. beginnt sie in PASCAL-Reihungen bei 1 bzw. für Ausschnittstypen bei einer beliebigen Untergrenze, während sie in C oder SATHER bei 0 beginnt. Daher hat unser Reihungstyp eine Untergrenze und eine Obergrenze der Indizierung und einen Elementtyp.

Diese Verallgemeinerung erlaubt die Definition von Reihungen beliebiger Programmiersprachen.

$$row: \ \mathsf{Int} \times \mathsf{Int} \times \mathsf{Type} \rightarrow \mathsf{Type}$$

lower, upper und *element_type* sind die entsprechenden Projektionsfunktionen. *size* gibt die Größe einer Reihung an und berechnet sich aus der Differenz von Unter- und Obergrenze.

```
RowType   ::=   NumConst1 NumConst2 Type

type(RowType) := row(Numconst1.ConstValue, NumConst2.ConstValue, type(Type))

New(RowType):
  if counter(ct) = ⊥ then
    extend Reference with r
      type(r) := type(ct)
      content(r) := r
      value(ct,reclevel) := r
    endextend
    counter(ct) := type(ct).lower
    proceed   /* entspricht: ct.element_type.New */
  else
    value(ct,reclevel).(counter(ct)) := value(ct.NT,reclevel)
    if counter(ct) < type(ct).upper then
      counter(ct) := counter(ct) + 1
      proceed
    else
      counter(ct) := ⊥
      ct := ct.rettask
    endif
  endif

Copy(RowType):
  if counter(ct) = ⊥ then
    counter(ct) := type(ct).lower
  else
    if counter(ct) ≤ type(ct).upper then
      counter(ct) := counter(ct) + 1
      ct := type(ct).element_type.Copy
      type(ct).element_type.Copy.source := value(ct.source),reclevel).(counter(ct))
      type(ct).element_type.Copy.dest := value(ct.dest,reclevel).(counter(ct))
      type(ct).element_type.Copy.rettask := ct
    else
      counter(ct) := ⊥
      ct := rettask
    endif
  endif
```

Abbildung 9.5: Die Definition der statischen Reihung

Die Erzeugung von Objekten eines Typs ist durch New-Operationen über der induktiven Struktur des Typs definiert. Zu jedem im Typterm auftretenden Typ gibt es ein New-Kommando. Der Steuerfluß zwischen den Kommandos ist durch die statische Struktur des Terms festgelegt und wird während der semantischen Analyse berechnet. Bei strukturierten Typen stellt die statische Semantik sicher, daß das New-Kommando einer Komponente nach ihrer Ausführung die Kontrolle wieder an das übergeordnete New-Kommando des strukturierten Typs zurückgibt. Die Zuweisung von strukturierten Objekten wird durch eine Copy-Operation erklärt, die rekursiv die Werte der Teilobjekte kopiert. Für Basisdatentypen wird einfach der Wert des Objekts zugewiesen. Für die Reihung beschreiben New*(RowType)* und Copy*(RowType)* das Erzeugen bzw. die Kopiersemantik.

Bemerkung: Bestimmte Programmiersprachen definieren, daß die Elemente strukturierter Typen in beliebiger Reihenfolge im Speicher angeordnet werden. Wir würden diese Eigenschaft modellieren, indem wir anstatt einer festen Erzeugungsstrategie mit dem All-Kommandos eine indeterministische Reihenfolge für die Erzeugung der Objekte einer Struktur angeben. ♢

In der Bibliothek gibt es zusätzlich eine Operation Alloc, mit der Objekte dynamisch erzeugt werden können. Alloc ist ein Kommando aus *AL* (siehe S. 219). Die Objekte werden auf der Halde angelegt und können über eine Referenz zugegriffen werden. Mit Alloc und den entspre-

chenden Kommandos zur Erzeugung strukturierter Objekte, können beliebige Strukturen auf der Halde angelegt werden.

9.2.3 Variablen

Der Zugriff auf Objekte ist in allen Sprachen durch einen Zugriffspfad beschrieben. In der Bibliothek gibt es den einfachen Zugriff über eine Variable, den Zugriff auf ein Element einer Reihung und den Zugriff auf eine Komponente eines Verbunds.

Designator ::= *Simple* | *Indexed* | *Qualified*

Kennt eine Quellsprache einen der Zugriffspfade nicht, dann wird das entsprechende Konzept einfach als leer definiert.

Ein Designator bezeichnet immer ein Objekt. Wird der Designator in einem Ausdruck verwendet, dann drücken wir das mit *Primary* aus. Die Semantik von *Primary* hängt vom Kontext der Verwendung ab. Handelt es sich um einen Ausdruck, der als formaler Parameter für einen Referenzparameter verwendet wird, dann berechnet *Primary* das durch *Designator* bezeichnete Objekt. Ansonsten wird der Inhalt dieses Objekts berechnet. Die Unterscheidung wird mit Hilfe des Attributs *lval* getroffen. Die Abbildung 9.6 beschreibt die Montage für *Primary* und zeigt eine Minimalanforderung für *Designator*.

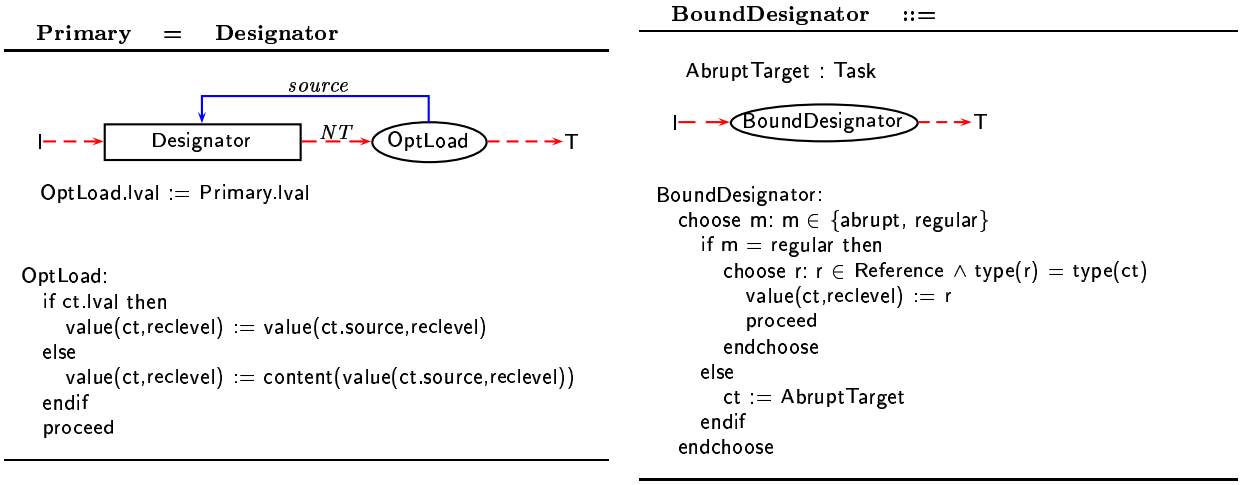


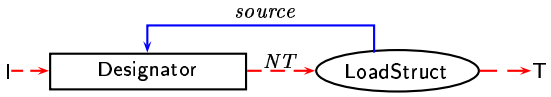
Abbildung 9.6: Die Semantik des Objektzugriffs

Lokale, globale oder Parameterobjekte sind durch einfache Bezeichner (*Simple*) beschrieben. Die Beziehung zwischen Namen und Objekten stellt das SimpleDesignator-Kommando her, das abhängig vom Bezeichner auf die aktuelle Parameterschachtel bzw. auf die Schachteln für lokale oder globale Variablen zugreift. Der Zugriff auf strukturierte Objekte wird allgemein durch *Qualified* und *Indexed* angegeben. Auf Elemente von Strukturen wird mit dem LoadStruct-Kommando über den Bezeichner für das Kopfobjekt und einen Bezeichner für das Element (Selektor) zugegriffen. Bei Reihungen ist der Selektor ein Index, der beim Zugriff auf Einhaltung von Grenzen überprüft wird, bei Verbunden ist der Selektor ein Name. Der Aufbau des Kommandographen stellt sicher, daß jeder Zugriffspfad in einem SimpleDesignator-Kommando endet. Die Minimalanforderung an den Index ist eine Spezialisierung der Minimalanforderung für allgemeine Ausdrücke. Wir fordern für den Typ des Ausdrucks $[\![type(Expr)]\!] = int$. Die Semantik der unterschiedlichen Zugriffspfade ist in Abb. 9.7 beschrieben.

Simple = Identifier


```
SimpleDesignator:
  if isparam(ct) then
    if kind(ct) = VAR then
      value(ct.reclevel) :=
              content(value(parframe,reclevel).(ct.string))
    else
      value(ct,reclevel) := value(parframe,reclevel).(ct.string)
    endif
  elsif islocal(ct) then
    value(ct,reclevel) := value(localframe,reclevel).(ct.string)
  else
    value(ct,reclevel) := globalframe.(ct.string)
  endif
  proceed
```

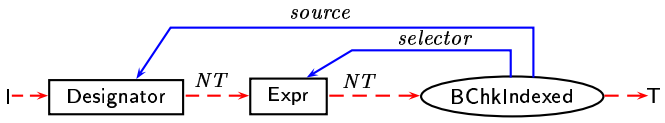
Qualified ::= Designator Identifier




```
type(Qualified) := type(Designator).(Identifier.string)
kind(Qualified) := kind(Designator)
LoadStruct.selector := Identifier.string
type(LoadStruct) := type(Designator).(Identifier.string)

LoadStruct:
  value(ct,reclevel) := value(ct.source,reclevel).(ct.selector)
  proceed
```

Indexed ::= Designator Expr




```
type(Indexed) := type(Designator).element_type
Expr.lval := false
kind(Indexed) := kind(Designator)
type(BChkIndexed) := type(Designator).element_type

BChkIndexed:
if (value(ct.selector,reclevel)<lower(type(value(ct.source,reclevel)))) or
   (value(ct.selector,reclevel)>upper(type(value(ct.source.value(reclevel)))))
then
   ct := BoundsCheckError      /* RunTimeError ! */
else
   value(ct,reclevel):=
         value(ct.source,(reclevel)).(value(ct.selector,reclevel))
   proceed
endif
```

BoundExpr ::=

```
mode: Task → { abrupt, regular }
AbruptTask : Task
```


```
Expr:
choose m: m ∈ {abrupt, regular}
   mode(ct) := m
   if m = regular then
    choose v: v ∈ Value ∧
                type(v)=type(BoundExpr)
          value(ct,reclevel) := v
    endchoose
    proceed
   else
    ct := AbruptTask
   endif
endchoose
```

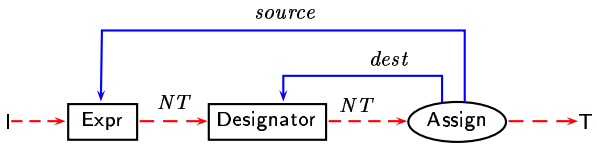
Abbildung 9.7: Die Semantik von Zugriffspfaden

9.2.4 Anweisungen

Die Bibliothek enthält die üblichen Anweisungen imperativer Sprachen: die Zuweisung, die While- und die For-Schleife mit Break und Continue, die bedingte Anweisung sowie Ein- und Ausgabeoperationen. Die Semantik der Anweisungen ist so definiert, daß möglichst viele Konstrukte existierender Sprachen dadurch beschrieben werden. Zum Beispiel kann die Schleife auch zur Modellierung einer Schleife benutzt werden, die keine Break-Anweisung kennt.

Die Zuweisung in imperativen Sprachen ist überladen definiert (Abb. 9.8). Betrachtet man := als Operator, dann ist dieser auf beliebige Argumente anwendbar, solange der Typ des rechten Operanden anpaßbar an den Typ des linken Operanden ist. Was in diesem Zusammenhang Anpaßbarkeit heißt, ist abhängig von den Konventionen der jeweiligen Sprache. Die Semantik der Zuweisung ist mit Hilfe der Kopiersemantik definiert, die es zu jedem Typ in Form eines entsprechenden Copy-Kommandos gibt. Für einen Referenztyp würde man nur den entsprechenden Zeiger kopieren, während man für einen Wertetyp, alle Elemente dieses Typs kopieren würde. Die Zuweisung ist generisch in den rechten und linken Seiten. Die Minimalanforderun-

Assignment ::= Designator Expr



Expr.lval := false

```
Assign:
  if type(ct.dest) ∈ BasicType then
    if (type(ct.source) ≠ type(ct.dest)) then
      content(value(ct.dest,reclevel) := coerce(value(ct.source,reclevel),type(ct.dest))
      proceed
    else
      content(value(ct.dest,reclevel) := value(ct.source,reclevel)
      proceed
    endif
  else
    type(ct.dest).Copy.rettask := ct.NT
    type(ct.dest).Copy.source := ct.source
    type(ct.dest).Copy.dest := ct.dest
    ct := type(ct.dest).Copy
  endif
```

Abbildung 9.8: Die Semantik der Zuweisung

gen für Designator und Expr haben wir schon angegeben. In der Semantik der Zuweisung ist auch eine Typanpassung vorgesehen, falls die Typen unterschiedlich sind. Die erlaubten Anpassungen gibt die statische Semantik der Quellsprache vor. Die semantische Analyse stellt sicher, daß für alle Zuweisungen eines Programms die Operanden anpaßbar sind.

Die Semantik einer While-Schleife haben wir, inklusive der Minimalanforderungen für die Schleifenbedingung und den Schleifenrumpf, schon in Abschnitt 7.4.3 angegeben. Das vorgestellte While-Konzept beschreibt Schleifen aus PASCAL oder MODULA. Eine Schleife in C erfüllt nicht direkt die Minimalanforderungen an die Schleifenbedingung. Wir werden aber in Abschnitt 9.5.1 sehen, wie Bedingungen in C mit Hilfe der Bibliothekskonzepte beschrieben werden können und wie damit auch das Konzept While zur Modellierung der C-While-Schleife verwendet werden kann.

Eine Schleife kann mit einer Break- oder einer Continue-Anweisung verlassen werden. Das Aussprungziel einer Break-Anweisung ist das nachfolgende Kommando der nächsten umschließenden Schleife und ist durch das Attribut *LoopEnd* beschrieben. Das Sprungziel für die Continue-Anweisung wird aus der Menge der ersten Kommandos der Schleifenbedingung (*LoopContinue*) ausgewählt (Abb. 9.9). Die Aussprungziele gehören zur statischen Semantik einer Sprache.

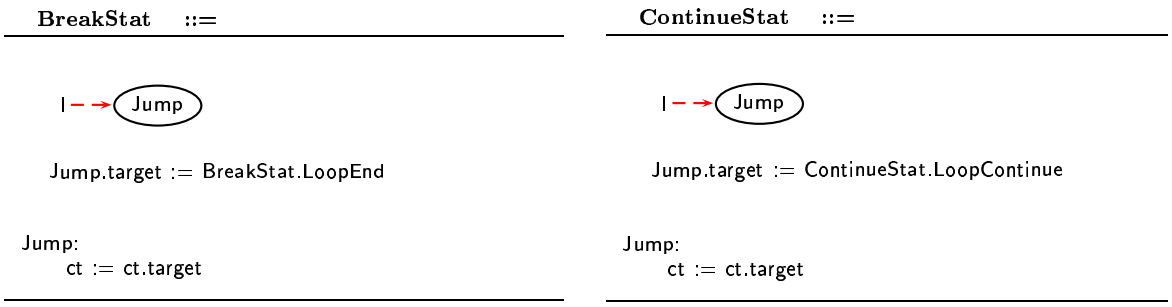


Abbildung 9.9: Die Semantik von Break und Continue

Die For-Schleife besteht im allgemeinsten Fall aus einer Initialisierung *ForInit*, einer Bedingung *Expr*, einem Schleifenrumpf *Stats* und einem Aktualisierungsteil *ForUpdate* (Abb. 9.10). Sie entspricht damit der For-Schleife von JAVA (GOSLING ET AL., 1996) und der For-Schleife in C (KERNIGHAN und RITCHIE, 1978). Die Minimalanforderungen für die Schleifenbedingung und den Schleifenrumpf sind die gleichen, wie bei der While-Schleife. Der Rumpf einer For-Schleife kann ebenfalls mit Continue oder Break verlassen werden. Allerdings unterscheidet sich die Definition des Attributs *LoopContinue*. Die Berechnung wird an der ersten Anweisung von *ForUpdate* fortgesetzt, wenn der Schleifenrumpf über eine Continue-Anweisung verlassen wird. Nach einer Break-Anweisung wird die umschließende Schleife abgebrochen und die auf die Schleife folgende Anweisung ausgeführt. Die Minimalanforderungen für *ForInit* und *ForUpdate* entsprechen der Minimalanforderung für die allgemeine Anweisungsliste. Daher gilt $BoundForInit = BoundForUpdate = BoundStats$.

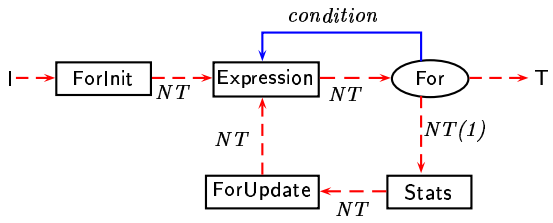
Bemerkung: Die For-Schleife ist auch zur Beschreibung der For-Schleife in PASCAL geeignet. Allerdings müßte man da sicherstellen, daß auch die Semantik bei Grenzen *MinInt* und *MaxInt* korrekt definiert ist. Man würde die kritischen Fälle separat behandeln und die Semantik der Schleife so beschreiben, wie sie auch in einem Übersetzer implementiert wird (WAITE und GOOS, 1984). ♢

Die bedingte Anweisung ist wie üblich definiert. Wir erlauben einen Then- und einen Else-Teil und können damit sowohl If-Then- als auch If-Then-Else-Anweisungen beschreiben (Abb. 9.11). Eine If-Then-Anweisung wird als If-Then-Else-Anweisung mit einem leeren Else-Teil modelliert. Für die Bedingung gelten die gleichen Bemerkungen wie bei der Schleife.

Die Case-Anweisung würde man analog zur bedingten Anweisung definieren, nur würde man für die Bedingung unterschiedliche Typen und beliebig viele Alternativen erlauben. Das One-Kommando aus *AL* beschreibt genau die Semantik, die mit Hilfe von Case ausgedrückt wird.

Ein- und Ausgabeströme sind in *AL* durch *input*, *output* :→ Value* modelliert. Read und Write-Kommandos definieren Ein- und Ausgabe-Operationen, wie sie z. B. in PASACL existieren. In den meisten Programmiersprachen werden die komplexen Ein-/Ausgabeoperationen durch Bibliotheksfunktionen implementiert. Daher ist es ausreichend, in der Bibliothek nur die beiden Konzepte in Abb. 9.12 anzubieten.

ForStat ::= ForInit Expr ForUpdate Stats

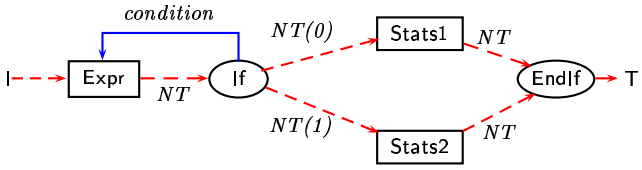


Stats.LoopContinue := choose(ForUpdate.initial)
Stats.LoopEnd := ForStat.NT(0)

For:
  if value(ct.condition,reclevel) = true then
    ct := ct.NT(1)
  else
    proceed
  endif

Abbildung 9.10: Die Semantik der For-Schleife

IfStat ::= Expr Stats1 Stats2



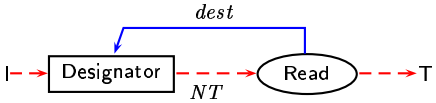
Expr.lval := false

If:
  if value(ct.condition,reclevel) = true then
    ct := ct.NT(0)
  else
    ct := ct.NT(1)
  endif

EndIf:
  proceed

Abbildung 9.11: Die Semantik der bedingten Anweisung

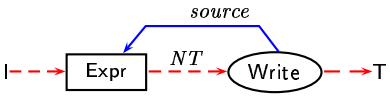
ReadStat ::= Designator



type(Read) := char

Read:
  content(value(ct.dest,reclevel) := head(input)
  input := tail(input)
  proceed

WriteStat ::= Expr



Expr.lval := false
type(Write) := char

Write:
  output := append(output,[value(ct.source,reclevel)])
  proceed

Abbildung 9.12: Die Semantik der Ein- und Ausgabeoperationen

Die dynamische Semantik eines Funktionsaufrufs (Abb. 9.13) unterscheidet sich von der Semantik eines Prozeduraufrufs nur dadurch, daß eine Schachtel für die Parameterobjekte und zusätzlichem Resultat erzeugt wird (CreateFrame), während der Resultatswert bei einem Prozeduraufruf undefiniert ist. Das Parameterobjekt wird auf der nächsthöheren Schachtelungstiefe zur Verfügung gestellt (SetParFrame), damit die aufgerufene Funktion auf die Parameter zugreifen kann. Anschließend werden die Argumente ausgewertet und an die Parameterobjekte zugewiesen. Dann erst erfolgt der eigentliche Funktionsaufruf durch einen Sprung zum ersten Kommando, der durch den Bezeichner identifizierten Funktion. Die Parameterobjekte werden zur Laufzeit durch die Funktion *parframe* verwaltet. Nach Abarbeitung der Funktion wird das Resultat aus dem Parameterobjekt gelesen.

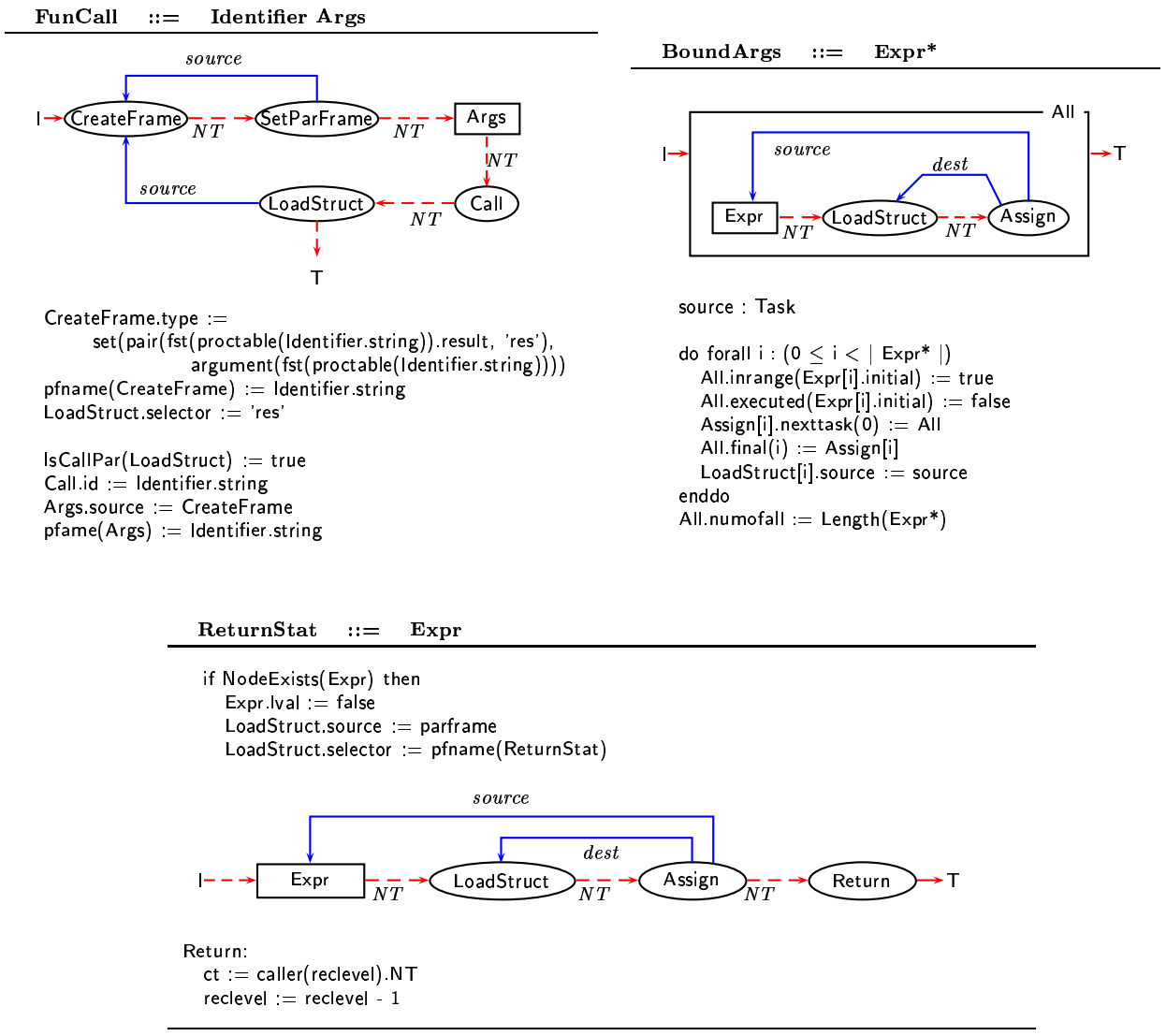


Abbildung 9.13: Der Funktionsaufruf mit Parametern und die Returnanweisung

Der Funktionsaufruf selbst ist durch das Kommando Call aus *AL* spezifiziert. Die Auswertung der Argumente haben wir als eigenständiges Sprachkonzept modelliert, da wir damit mehr Freiheiten haben. Nehmen wir als Minimalanforderung für die Argumentauswertung eine beliebige Auswertungsreihenfolge an, dann können wir sowohl Instanzen mit fester, als auch mit beliebiger Auswertungsreihenfolge modellieren. Spezifizieren wir eine Quellsprache die Argumente in beliebiger Reihenfolge auswertet, dann beschreibt die Minimalanforderung schon die Semantik dieses Quellsprachkonzepts.

Mit der Return-Anweisung kann eine Prozedur oder Funktion verlassen werden, wobei im Falle einer Funktion ein Resultatswert zurückgegeben werden kann.

9.2.5 Ausdrücke

Neben den Konstanten und dem Zugriff auf den Inhalt von Objekten kennt unsere Bibliothek noch unäre und binäre Ausdrücke. Wir beschreiben hier nur binäre Ausdrücke. ComposedExpr definiert mit Hilfe des All Kommandos eine beliebige Reihenfolge bei der Berechnung der Teilausdrücke eines Ausdrucks (Abb. 9.14 links). Das entspricht z. B. der Semantik der Ausdrucksberechnung von C. Die logischen Operationen LogAnd und LogOr werden mit Kurzauswertung berechnet. Dementsprechend wird der rechte Teilausdruck nur in Abhängigkeit des linken Teilausdrucks ausgewertet. In Abb. 9.14 haben wir die Semantik der Kurzauswertung eines Ausdrucks mit LogAnd beschrieben, die Semantik von LogOr ist analog. Die semantischen Details der einzelnen arithmetischen Operationen sind der Beschreibung in (Heuzeroth und Heberle, 1998) zu entnehmen.

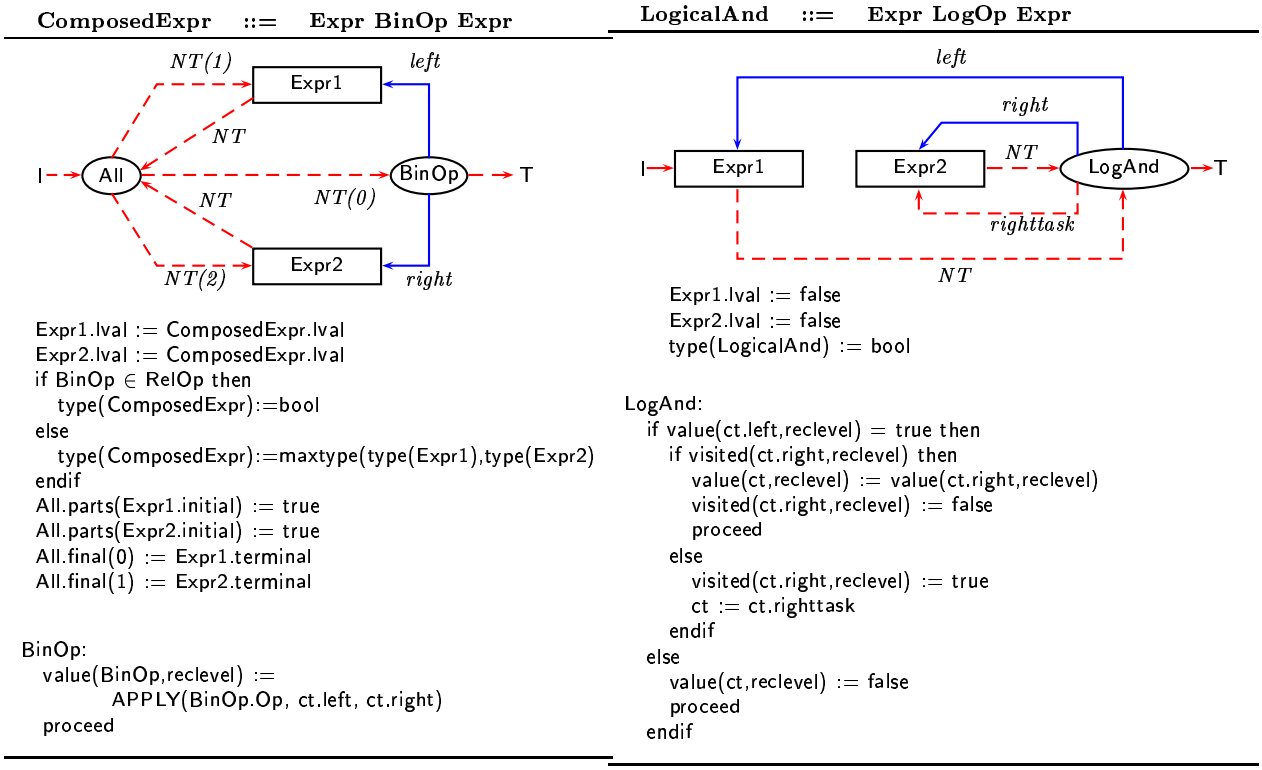


Abbildung 9.14: Die Semantik der Ausdrucksberechnung

Eine alternative Definition der Ausdrucksberechnung ergibt sich, wenn wir den Steuerfluß zwischen den Ausdrücken und der Operation nicht mehr explizit machen (Abb. 9.15 links). In diesem Fall würde man indeterministisch über *proceed* aus der Menge der Nachfolger einen auswählen, der mit den Datenabhängigkeiten konform ist. Diese Semantik erfordert allerdings einen zusätzlichen Berechnungsschritt in dem eine Ordnung über Kommandos gemäß den Datenabhängigkeiten festgelegt wird. Gibt eine Sprache eine feste Auswertungsreihenfolge über Ausdrücken vor, wie das z. B. Java tut, dann würde man für die Ausdrucksauswertung den Steuerfluß wie in Abb. 9.15, rechts, festlegen.

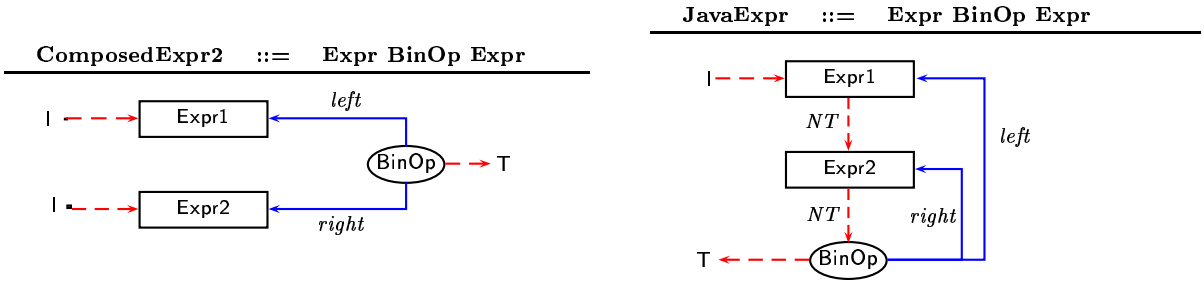


Abbildung 9.15: Alternative Semantiken der Ausdrucksberechnung

Zusammen mit der Semantik der Operationen über den Basisdatentypen, die indirekt durch die Parametrisierung der *AL*-Basisdatentypen festgelegt werden, sind die vorgestellten Konzepte damit ausreichend, um beliebige imperative Quellsprachen zu spezifizieren.

9.3 Übersetzung der Bibliothekskonzepte nach *AL*

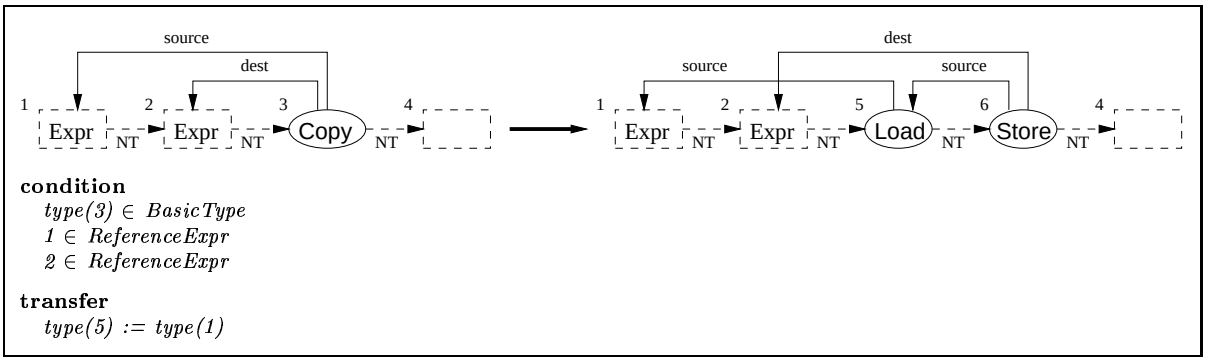
Die Semantik der Konzepte, die wir im letzten Abschnitt eingeführt haben, ist teilweise unter Verwendung von *AL*-Kommandos spezifiziert, teilweise werden höhere Sprachkonzepte verwendet, die noch nach *AL* abgebildet werden müssen. Im folgenden spezifizieren wir die Abbildung von *IS* nach *AL* durch Graphersetzungsregeln, deren Bedeutung in Anhang A.3 angegeben ist.

Jede Abbildung eines höheren Sprachkonzepts stellt eine Regel des Transformationskalküls dar und definiert eine Erweiterung der Syntax des Kalküls.

9.3.1 Datentypen

Ein Typ abstrahiert die Eigenschaften einer Menge von Werten. Die Typisierung gehört zur statischen Semantik. Die dynamische Semantik beschreibt die Erzeugung und das Kopieren von Objekten.

Die Erzeugung von Objekten eines Basistyps ist schon durch die entsprechenden AL-Kommandos (New) spezifiziert worden. Daher ist keine Übersetzung notwendig. Die Kopierfunktion Copy existiert in *AL* jedoch nicht. Sie läßt sich aber durch ein *AL*-Programm nachimplementieren.



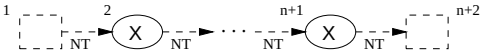
ReferenceExpr beschreibt die Unterklasse von Ausdrücken, die eine Referenz berechnen.

Die Abbildung von strukturierten Typen beschreiben wir am Beispiel der statischen Reihung mit Wertesemantik. *AL* kennt nur den strukturierten Typ Menge (Set). Der Funktor Φ bildet die Typen aus AL^+ auf die Typen aus *AL* ab. Aus algebraischer Sicht beschreibt Φ eine Uminterpretation der Typterme aus AL^+. Für den Reihungstyp ist diese Uminterpretation wie folgt definiert. Sei $q \in Alg(\Sigma_{AL^+})$ der ursprüngliche und $q' \in Alg(\Sigma_{AL})$ der neue Zustand, dann gilt:

$$\forall t \in [\![\, Type \,]\!]_q : [\![\, t \,]\!]_q = row(l,u,et) \Rightarrow$$
$$[\![\, t \,]\!]_{q'} = \begin{cases} set(pair([\![\, et \,]\!]_{q'}, [\![\, l \,]\!]_q), [\![\, row(l+1, u, et) \,]\!]_{q'}), & \text{falls } [\![\, l \,]\!]_q < [\![\, u \,]\!]_q \\ set(pair([\![\, et \,]\!]_{q'}, [\![\, l \,]\!]_q), \bot), & \text{falls } [\![\, l \,]\!]_q = [\![\, u \,]\!]_q \end{cases} \quad (9.1)$$

$\Phi(t)$ berechnet für einen Typterm t der Quellsprache den entsprechenden Typterm aus *AL*.

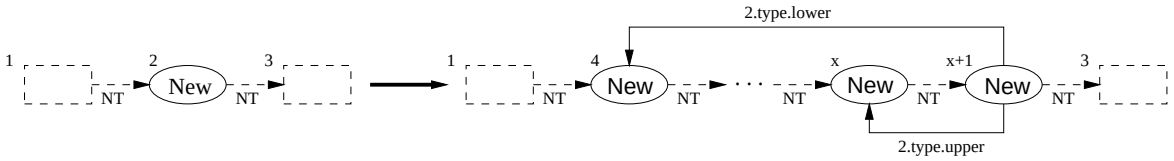
Zur Spezifikation von Transformationen, die eine beliebige, aber feste Anzahl gleicher Kommandos einführen, benutzen wir eine abkürzende Schreibweise. Der Graph



beschreibt das n-fache Auftreten des Kommandos X, wobei $1 \leq n$ gilt. Die Kommandos sind durchlaufend numeriert. Verwenden wir diese spezielle graphische Notation auf der linken und rechten Seite einer Graphersetzungsregel, dann beschreibt das die Menge der Ersetzungsregeln, für beliebige n. In den Transferregeln werden die entsprechenden Attribute dann exakt durch eine prädikatenlogische Formel angegeben.

Die New-Operation für eine statische Reihung wird aufgespalten in die Erzeugung der Einzelelemente und die Erzeugung eines Kopfobjekts. In der folgenden Transformationsregel ist *kind* ein Platzhalter für die Prädikate *isglobal, islocal* und *isparam* über Schachteln.

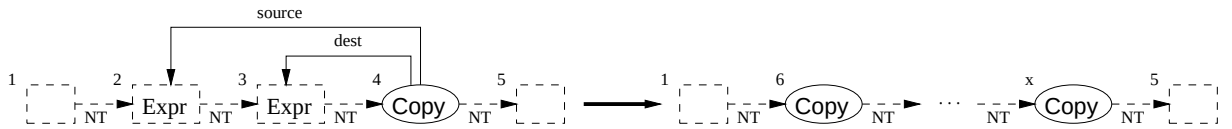
In dieser Transformation wird nur der Typ des Kopfobjekts $(x+1)$ verändert. Die Typen der New-Kommandos für die Elemente werden durch die Anwendung anderer Transformationsregeln abgebildet.



condition
type(2) $\in$ *RowType*

transfer
($\forall i, 0 \leq i \leq (type(2).size - 1)$:
type(4+i) := type(2).element_type
kind(4+i) := kind(2).element_type
(4+i).NT(0) := $4+i+1$)
type(4+(type(2).size)) := Φ*(type(2))*
kind(4+(type(2).size)) := kind(2)
where *x=4+(type(2).size))*

Die Kopieroperation für statische Reihungen wird durch die sequentielle Ausführung der Kopieroperationen der einzelnen Elemente beschrieben.

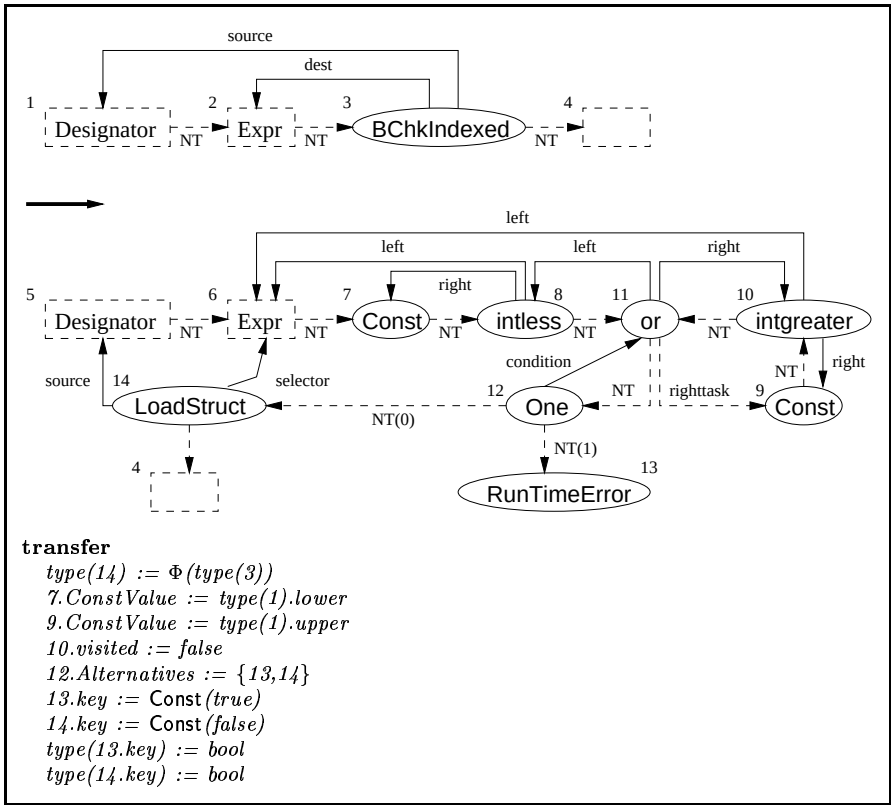


condition
type(4) $\in$ *RowType*
2 $\in$ *ReferenceExpr*
3 $\in$ *ReferenceExpr*

transfer
($\forall i, 0 \leq i \leq deref(type(2)).size - 1$:
(6+i).source := 4.source.((deref(type(2))).lower)+i)
(6+i).dest := 4.dest.((deref(type(3))).lower)+i)
($\forall i, 0 \leq i \leq deref(type(4.source)).size - 2$:
$(6 + i).NT(0) := 6 + i + 1$)
$(6 + deref(type(4.source)).size - 1).NT(0) = 5$
where $x = 6 + deref(type(4.source)).size - 1$

Die Quelle und die Senke der Kopieroperation berechnen Referenzen auf Reihungen. Die Funktion $deref : \uparrow T \rightarrow T$ dereferenziert einen Referenztyp.

Der Zugriff auf ein Element einer statischen Reihung wird in AL^+ durch das Kommando BChkIndexed beschrieben. Dabei wird überprüft, ob der Index innerhalb der Grenzen der Reihung ist. Durch die Abbildung nach AL wird diese Überprüfung explizit im Programm ausgedrückt. Wird ein Zugriff außerhalb der Grenzen versucht, dann tritt ein Laufzeitfehler auf. Selbstverständlich könnte es in einer Sprache unterschiedliche Arten von Laufzeitfehlern geben. In unserer Bibliothek abstrahieren wir davon und modellieren Laufzeitfehler durch einen Sprung auf ein spezielles Kommando RunTimeError.

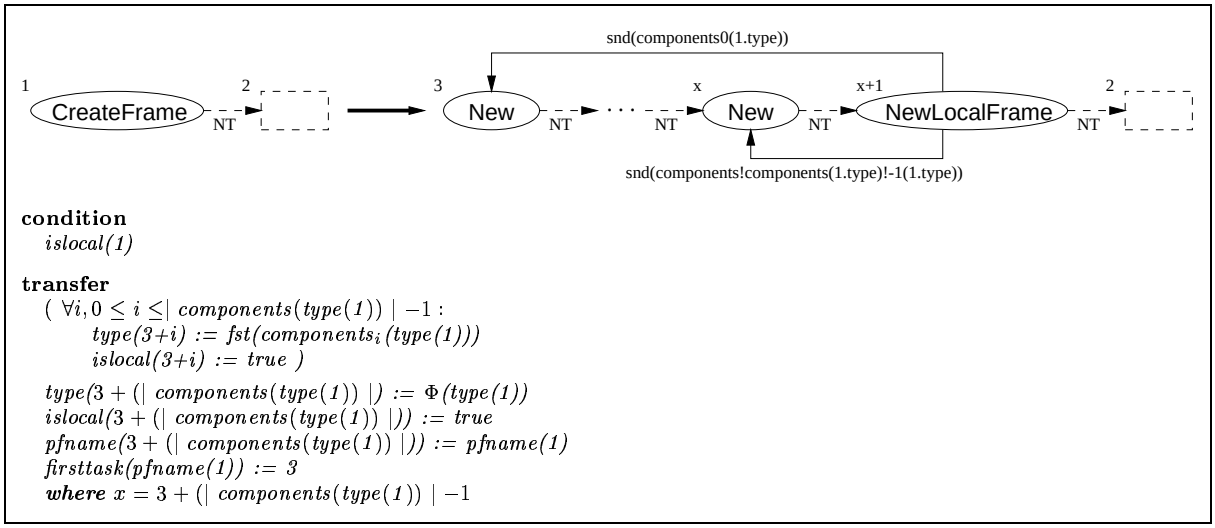


transfer
type(14) := Φ(type(3))
7.ConstValue := type(1).lower
9.ConstValue := type(1).upper
10.visited := false
12.Alternatives := {13,14}
13.key := Const*(true)*
14.key := Const*(false)*
type(13.key) := bool
type(14.key) := bool

9.3.2 Schachteln und Bezeichner

Die Erzeugung einer Schachtel für lokale, globale oder Parameterobjekte ist rekursiv über der Erzeugung der einzelnen Objekte definiert. Eine Schachtel ist ein strukturiertes Objekt, der Typterm beschreibt den Aufbau dieses Objekts. Bisher ist die Erzeugung einer Prozedurschachtel durch die dynamische Semantik des CreateFrame-Kommandos beschrieben. Die folgende Transformation macht die Erzeugung explizit, indem eine Struktur über den entsprechenden New-Kommandos der einzelnen Objekte aufgebaut wird.

Die Erzeugung von Schachteln für Funktionen und das Hauptprogramm sind prinzipiell gleich. Die erste Schachtel, die bei der Ausführung eines Programms erzeugt wird, ist die Schachtel für die globalen Variablen. Sie wird genau einmal erzeugt und kann daher über eine feste Adresse zugegriffen werden. Lokale Variablen werden über das oberste Element des Laufzeitkellers zugegriffen. Diese unterschiedliche Semantik wird in *AL* auch syntaktisch durch NewGlobalFrame- bzw. NewLocalFrame-Kommandos unterschieden. In der folgenden Abbildung beschreiben wir die Transformation für lokale Schachteln. Die Transformation für die globale Schachtel ist analog. Die Transformation bildet noch nicht die Typen ab, die die Erzeugung der einzelnen Objekte bestimmen. Das muß so sein, weil die einzelnen New-Kommandos noch rekursiv weiter transformiert werden müssen, falls es sich um strukturierte Objekte handelt. Nur das Kopfobjekt der Schachtel wird durch ein *AL*-Kommando erzeugt.



condition
islocal(1)

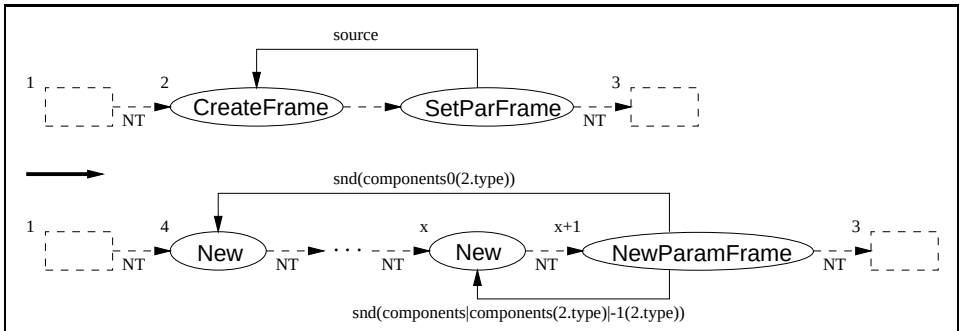
transfer
($\forall i, 0 \leq i \leq |\ components(type(1))\ | -1$:
type(3+i) := fst(components$_i$(type(1)))
islocal(3+i) := true)
type$(3 + (|\ components(type(1))\ |)) := \Phi(type(1))$
islocal$(3 + (|\ components(type(1))\ |)) := true$
pfname$(3 + (|\ components(type(1))\ |)) := pfname(1)$
firsttask(pfname(1)) := 3
where $x = 3 + (|\ components(type(1))\ | -1$

Die Erzeugung von Parameterschachteln besteht aus zwei Aktionen, die in *AL* zusammengefaßt sind. Zuerst wird die Schachtel erzeugt, dann wird die Schachtel auf der nächsthöheren Aufruftiefe bekannt gemacht. Die Transformation berücksichtigt, daß die einzelnen Elemente einer Parameterschachtel Werte- oder Referenzparameter darstellen, die unterschiedlich behandelt werden müssen. Für einen Werteparameter wird ein New-Kommando mit dem Typ des Parameters eingeführt, für einen Referenzparameter ist der Typ von New eine Referenz auf den Parametertyp. Die Abbildung Φ der Quellsprachtypen berücksichtigt dies. Sei wieder $q \in Alg(\Sigma_{AL^+})$ und $q' \in Alg(\Sigma_{AL})$, dann gilt:

$$\forall n \in [\![\ New\ ]\!]_q : [\![\ type\ ]\!]_q(n) = ptupel([\![\ t\ ]\!]_q, name, VAL) \Rightarrow [\![\ type\ ]\!]_{q'}(n) = [\![\ t\ ]\!]_q \tag{9.2}$$

$$\forall n \in [\![\ New\ ]\!]_q : [\![\ type\ ]\!]_q(n) = ptupel([\![\ t\ ]\!]_q, name, VAR) \Rightarrow [\![\ type\ ]\!]_{q'}(n) = \uparrow [\![\ t\ ]\!]_q \tag{9.3}$$

Der Typ t des Schachtelelements selbst wird in dieser Transformation noch nicht verändert. Das passiert bei der Abbildung der Komponenten der Parameterschachtel.

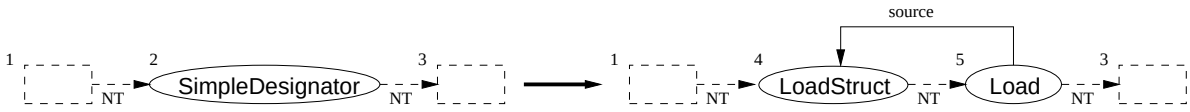


condition
isparam(2)

transfer
($\forall i, 0 \leq i \leq |\ components(type(1))\ | -1$:
type(4+i) := fst(components$_i$ (type(2)))
isparam(4+i) := true)
$type(4 + (|\ components(type(2))\ |)) := \Phi(type(2))$
$isparam(3 + (|\ components(type(2))\ |)) := true$
$pfname(3 + (|\ components(type(2))\ |)) := pfname(2)$
where $3 + (|\ components(type(1))\ |$

Die dynamische Semantik von einfachen Bezeichnern ist durch das Kommando SimpleDesignator gegeben. Die Abbildung auf *AL* hängt vom Typ des Bezeichners ab. Wir müssen Werte- und Referenzparameter von lokalen und globalen Variablen unterscheiden.

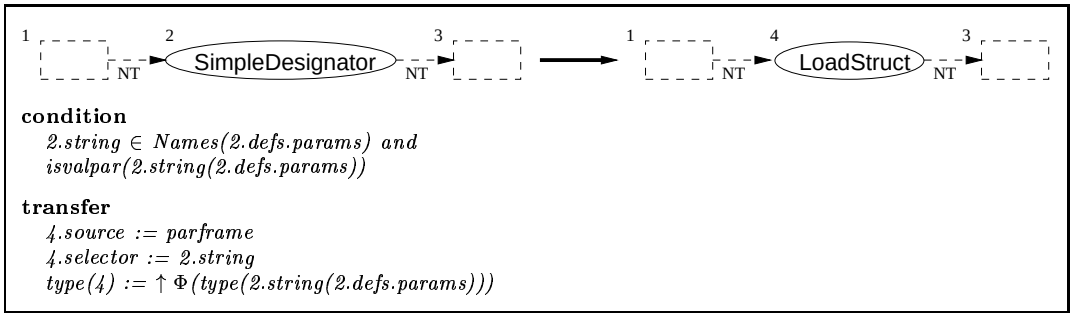
Entsprechend der Modellierung von Referenzparametern muß bei der Verwendung eines Referenzparameters eine zusätzliche Dereferenzierung durchgeführt werden. Die ASM-Regel zur Interpretation eines SimpleDesignator-Kommandos beschreibt diese Dereferenzierung. Bei der Abbildung nach *AL* wird diese Dereferenzierung durch ein Load-Kommando im Programm explizit gemacht.



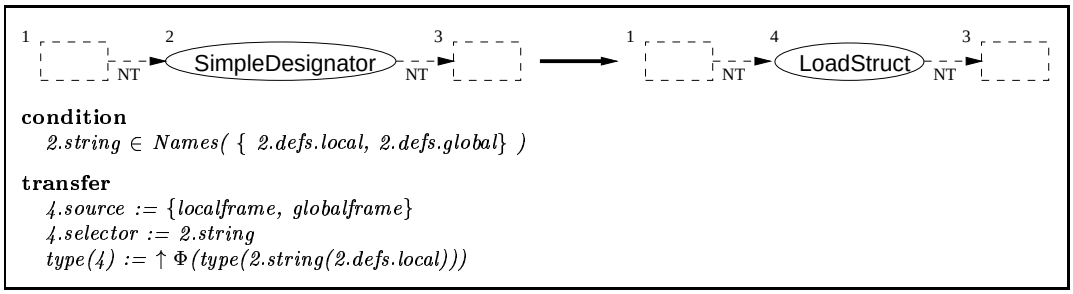
condition
2.string ∈ Names(2.defs.params) and
isvarpar(2.string(2.defs.params))

transfer
4.source := parframe
4.selector := 2.string
$type(4) := \uparrow \Phi(type(2.string(2.defs.params)))$
$type(5) := \Phi(type(2.string(2.defs.params)))$

isvarpar (*isvalpar*) ist erfüllt, wenn der Name *2.string* einen Referenzparameter (Werteparameter) bezeichnet. Der Zugriff auf Werteparameter und lokale bzw. globale Variablen unterscheidet sich nur durch die Schachtel auf die mit LoadStruct zugegriffen wird. Die entsprechende Schachtel ist durch *parframe*, *localframe* oder *globalframe* beschrieben, die einen Verweis auf die entsprechende Schachtel enthalten. In der zweiten Ersetzungsregel sind die Transformationen für lokale und globale Variablen zusammen beschrieben.



bzw.

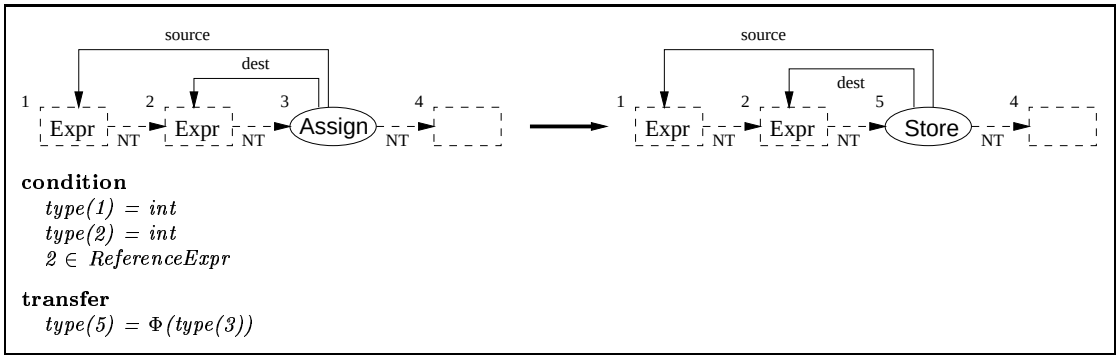


In Zusammenhang mit Bezeichnern haben wir das Kommando OptLoad definiert, das zu einem Bezeichner entweder das entsprechende Objekt oder den Wert des Objekts liefert. Die Unterscheidung wird mit Hilfe des Attributs *lval* getroffen. Wir übersetzen OptLoad in ein Load-Kommando, wenn $[\![lval]\!] = false$ und eliminieren OptLoad im Falle von $[\![lval]\!] = true$. Die Graphersetzungsregel ist klar und wird daher nicht extra aufgeführt.

9.3.3 Anweisungen

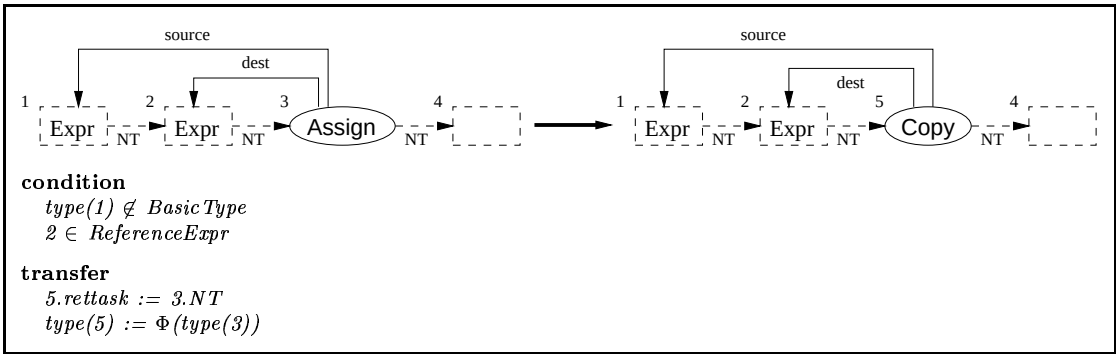
Die Abbildung von Anweisungen betrifft nur noch die Zuweisung, die bedingte Anweisung, die Schleife und die Break- bzw. Continue-Anweisung. Alle anderen Anweisungen sind schon durch *AL*-Terme beschrieben.

Bei der Zuweisung unterscheiden wir die Zuweisung von Basiswerten und von strukturierten Objekten. Für die Zuweisung eines Basiswertes stellt *AL* die Store-Kommandos entsprechenden Typs zur Verfügung.



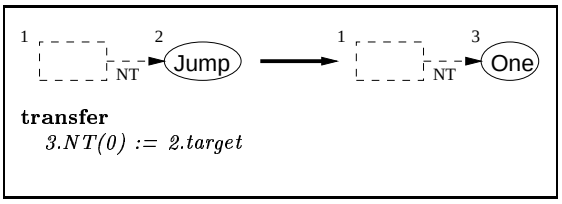
Die Zuweisung definiert eine automatische Typanpassung, falls Integer-Werte an ein Float-Objekt zugewiesen werden. In diesem Fall würde dem Store-Kommando eine IntToFloat-Operation vorgeschaltet. Die Semantik von IntToFloat ist maschinenabhängig.

Die Zuweisung von strukturierten Objekten geschieht mit Hilfe entsprechender Copy-Kommandos. In Abschnitt 9.3.1 haben wir die Transformation des Copy-Kommando für statische Reihungen beschrieben.

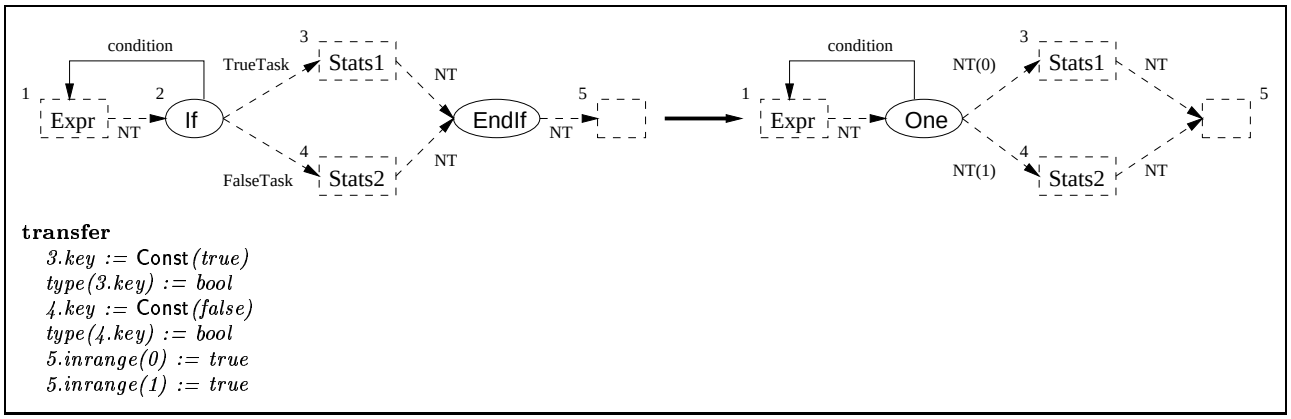


Die Transformation der While-Schleife wurde schon in Abschnitt 7.4.3 spezifiziert und verifiziert.

Die Semantik der Break- bzw. Continue-Anweisung ist durch einen unbedingten Sprung beschrieben. Wir bilden den unbedingten Sprung durch einen bedingten Sprung mit One nach. Das Sprungziel ist durch das Attribut *target* des Jump-Kommandos definiert. Aus diesem Attribut berechnet sich der Nachfolger des One-Kommandos im Steuerfluß, der angesprungen wird, falls es keine Alternativen gibt, die die Bedingung von One erfüllen. Da wir keine Bedingung definieren, tritt dieser Fall immer ein.



Die Semantik des If-Kommandos kann analog zum While-Kommando 1:1 in eine *AL*-Struktur abgebildet werden. Wir definieren für die Verzweigung die Attribute *key* und *inrange*. Das EndIf-Kommando schaltet nur den Steuerfluß weiter und wird einfach durch eine Steuerflußkante ersetzt.



9.3.4 Ausdrücke

Die Abbildung von Ausdrücken betrifft nur noch die Operatoren, die zwar in der Quellsprache, aber nicht in *AL* vorhanden sind. Sie müssen durch entsprechende *AL*-Operationen nachimplementiert werden. Zum Beispiel unterscheidet *AL* den Negationsoperator vom Komplementoperator während Quellsprachen den Negationsoperator häufig überladen definieren. Die Transformation überladener Operatoren ist jedoch nur die Anwendung des Spezialisierungssatzes

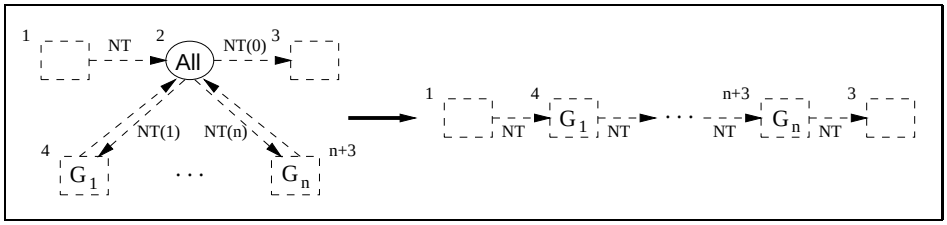
aus Abschnitt 8.2.4. Komplexe Operationen können wir entsprechend dem Dekompositionsschema 1 durch weniger komplexe Operationen implementieren.

9.3.5 Zusätzliche Transformationen

AL hat ein abstraktes Speichermodell in dem Objekte über symbolische Referenzen angesprochen werden. Auf der Zielmaschine greift man Speicherstellen über konkrete Adressen zu, die sich aus einer Basis und einer Relativadresse berechnen. Wir haben in Kapitel 7 die notwendigen Eigenschaften der Speicherabbildung formuliert, aus denen die Korrektheit der Abbildung folgt. Damit haben wir die Möglichkeit beliebige Algorithmen für die Speicherabbildung zu verwenden. Die Korrektheit dieser Algorithmen kann durch Überprüfung der geforderten Eigenschaften sichergestellt werden.

Die Spezialisierung überladener Operatoren wird mit Hilfe des Spezialisierungsschemas aufgelöst. Im vorigen Kapitel haben wir das am Beispiel der Zuweisung schon demonstriert.

Das Ziel von *AL* ist, möglichst alle Freiheiten der Semantik eines Programms zu repräsentieren. Zwischensprachprogramme sind üblicherweise deterministischer als Quellprogramme. Daher ist die folgende Transformation für *AL*-Graphen zweckmäßig. Das All-Kommando definiert eine indeterministische Berechnungreihenfolge über seinen Argumenten. Eine korrekte Transformation ist in diesem Fall, wenn eine bestimmte Folge aus der Menge der erlaubten Berechnungsfolgen ausgewählt wird. Graphisch stellt sich die Transformation wie folgt dar.



Die Argumente von All können Teilgraphen sein. Deswegen ist es unter Umständen schwierig, die letzten Kommandos dieser Teilgraphen zu identifizieren. Wir nutzen dazu das Attribut *final*, welches wir extra für diese Transformation von der Quellsprachebene her mitgeführt haben. Es beschreibt das letzte Kommando der Argumente eines All-Kommandos und erleichtert damit die Identifikation der G_i. In der Quellsprache läßt sich dieses Attribut leicht aus der Struktur ablesen, aber auf der Ebene eines *AL*-Graphen muß man eine aufwendige Analyse betreiben, um diese Information zu berechnen.

9.4 Korrektheit der Transformationen

Bei der Abbildung der Quellsprachkonzepte nach *AL* haben wir uns immer im Speicherzustandsraum von *AL* bewegt. Daher war keine Speicherabbildung notwendig. Erst bei der Abbildung in eine Zwischensprache wird diese Transformation benötigt. Die Verifikation der Speicherabbildung haben wir schon in Abschnitt 7.4.2 am Beispiel vorgestellt. Für die anderen Transformationen können wir jeweils die Relation ρ aufstellen und zum Nachweis der Simulation über die Aktualisierungen argumentieren, die durch die neuen und alten Programmteile beschrieben sind.

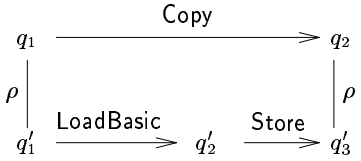
Die meisten der vorgestellten Transformationen entsprechen allgemeinen Transformationsschemata, die wir im vorigen Kapitel eingeführt haben. Für die Verifikation einer konkreten Trans-

formation zeigt man nur, daß die Transformation auch tatsächlich eine Instanz eines verifizierten Transformationsschemas ist.

Für jede Transformation des vorigen Abschnitts beschreiben wir hier die Beweisidee und geben entweder das allgemeine Transformationsschema oder die Relation ρ an. Die Details der Beweise können in (HEUZEROTH, 1998) nachgelesen werden.

Kopieren von Grundtypen
Das Copy-Kommando über Grundtypen wird durch ein Load-Kommando und die anschließende Ausführung eines Store-Kommandos implementiert.

Wir haben

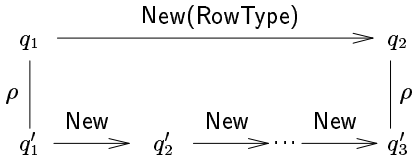


zu zeigen.

Die Transformation entspricht einer Dekomposition der Aktualisierungen von Copy und ist eine Instanz des Transformationsschemas Dekomposition aus Abschnitt 8.2.1.

Erzeugung von Reihungen
Die Ausführung von New(*RowType*) erzeugt zunächst ein Kopfobjekt (eine Referenz) für die Reihung. Anschließend wird in einer Schleife jedes der Reihungselemente erzeugt. Die Elemente können selbst wieder komplexe Objekte sein, so daß weitere Erzeugungsaktionen ausgeführt werden. Nachdem ein Reihungselement erzeugt wurde, wird es mit dem aktuellen Index als Selektor im Reihungsobjekt eingehängt. Die *AL*-Implementierung erzeugt zunächst die Objekte der Elemente und am Ende das Reihungsobjekt selbst. Die Elementobjekte werden unter demselben Index wie in der Originalreihung abgelegt, weil in beiden Fällen die Abarbeitungsreihenfolge durch den Typterm definiert ist. Die Erzeugung von komplexen Reihungselementen ist korrekt, wenn das zugehörige New-Kommando korrekt ist. Durch die Abbildung wird die Information über die Generierungsreihenfolge, die ursprünglich aus dem Typterm zu ersehen war, nun explizit durch Steuerflußkanten dargestellt.

Für den Beweis zeigen wir, daß das folgende Diagramm kommutiert.



Dabei spielt das Universum Reference die Hauptrolle. Nach der Abarbeitung von New(RowType) muß Reference genauso viele und genauso strukturierte Referenzen enthalten wie das nach der Abarbeitung der transformierten Version des New-Kommandos der Fall ist.

Ein Zustand $q' \in Alg(\Sigma_2)$ steht in Relation ρ zu einem Zustand $q \in Alg(\Sigma_1)$, wenn

$$\forall r_1 \in [\![\mathit{Reference}]\!]_q \, \exists_1 r_2 \in [\![\mathit{Reference}]\!]_{q'} :$$
$$([\![\mathit{type}]\!]_q(r_2) = \Phi([\![\mathit{type}]\!]_{q'}(r_1)) \ \wedge \ ([\![\mathit{type}]\!]_q(r_1) \notin [\![\mathit{StructuredType}]\!]_q) \Rightarrow$$
$$([\![\mathit{content}]\!]_q(r_1) = [\![\mathit{content}]\!]_{q'}(r_2))) \quad (9.4)$$

$$\forall r_2 \in [\![\mathit{Reference}]\!]_{q'} \exists_1 r_1 \in [\![\mathit{Reference}]\!]_q : ([\![\mathit{type}]\!]_{q'}(r_2) = \Phi([\![\mathit{type}]\!]_q(r_1)) \wedge ([\![\mathit{type}]\!]_{q'}(r_2) \notin [\![\mathit{StructuredType}]\!]_{q'}) \Rightarrow ([\![\mathit{content}]\!]_q(r_1) = [\![\mathit{content}]\!]_{q'}(r_2))) \quad (9.5)$$

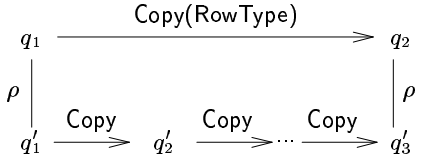
$$[\![ct]\!]_q \in [\![\mathsf{New}(\mathit{RowType})]\!]_q \Rightarrow [\![ct]\!]_{q'} \in [\![\mathsf{New}]\!]_{q'} \wedge [\![\mathit{type}(ct)]\!]_{q'} = [\![\mathit{element\_type}(\mathit{type}(ct))]\!]_q \quad (9.6)$$

$$[\![ct]\!]_q \notin [\![\mathsf{New}(\mathit{RowType})]\!]_q \Rightarrow [\![ct]\!]_{q'} = [\![ct]\!]_q \quad (9.7)$$

Die Interpretation der anderen Funktionen ist abgesehen von der Änderung des Programms gleich.

Kopieren einer statischen Reihung

Die Schleife zum Kopieren der einzelnen Elemente der Reihung wird ausgerollt. Wir haben



zu zeigen. Der Beweis funktioniert analog zum Nachweis der korrekten Erzeugung der Reihung. Die Simulation erreichen wir durch Aufspaltung der Aktualisierungsmenge, die durch $\mathsf{Copy}(\mathit{RowType})$ definiert wird. Nur argumentieren wir dieses Mal nicht über die Anzahl der Referenzen, sondern nur über den Inhalt der Referenzen.

Ein Zustand $q' \in \mathit{Alg}(\Sigma_2)$ steht in Beziehung ρ zu einem Zustand $q \in \mathit{Alg}(\Sigma_1)$, wenn

$$[\![\mathit{Reference}]\!]_q = [\![\mathit{Reference}]\!]_{q'} \quad (9.8)$$

$$\forall r_1 \in [\![\mathit{Reference}]\!]_{q_1} : [\![\mathit{content}]\!]_q(r) = [\![\mathit{content}]\!]_{q'}(r) \quad (9.9)$$

$$[\![ct]\!]_q \in [\![\mathsf{Copy}(\mathit{RowType})]\!]_q \Rightarrow [\![ct]\!]_{q'} \in [\![\mathsf{Copy}]\!]_{q'} \wedge [\![\mathit{type}(ct)]\!]_{q'} = [\![\mathit{element\_type}(\mathit{type}(ct))]\!]_q \quad (9.10)$$

$$[\![ct]\!]_q \notin [\![\mathsf{Copy}(\mathit{RowType})]\!]_q \Rightarrow [\![ct]\!]_{q'} = [\![ct]\!]_q \quad (9.11)$$

Die Interpretation der anderen Funktionen ist abgesehen von der Änderung des Programms gleich.

Erzeugung von Kellerschachteln

Kellerschachteln sind spezielle Verbunde und werden entsprechend transformiert. Die Erzeugung der Teilelemente, die im CreateFrame-Kommando zusammengefaßt war, wird in AL durch eine explizite Struktur von New-Kommandos implementiert. Die Verifikation funktioniert analog zur Erzeugung von Reihungen. Wir argumentieren dabei wieder über die Anzahl der Referenzen.

Bezeichner

Die Übersetzung macht die unterschiedliche Semantik des SimpleDesignator-Kommandos explizit. Wir wenden zuerst das Spezialisierungsschema 6 aus Abschnitt 8.2.4 an und zerlegen das Kommando, in Abhängigkeit davon, ob es sich um einen Werte- oder Referenzparameter

bzw. eine lokale oder globale Variable handelt. Im zweiten Schritt implementieren wir den Zugriff auf die entsprechende Schachtel mit *AL*-Kommandos.

Wir geben die Relation ρ nur für den Fall an, daß es sich bei dem Bezeichner um eine lokale Variable handelt, die anderen Fälle sind analog.

Ein Zustand $q' \in Alg(\Sigma_2)$ steht in Beziehung ρ zu einem Zustand $q \in Alg(\Sigma_1)$, wenn

$$\begin{array}{l} [\![\, ct \,]\!]_q \in [\![\, \mathsf{SimpleDesignator} \,]\!]_q \Rightarrow \\ [\![\, ct \,]\!]_{q'} \in [\![\, \mathsf{LoadStruct} \,]\!]_{q'} \wedge \uparrow \Phi([\![\, type(ct) \,]\!]_q) = [\![\, type(ct) \,]\!]_{q'} \end{array} \tag{9.12}$$

$$[\![\, ct \,]\!]_q \notin [\![\, \mathsf{SimpleDesignator} \,]\!]_q \Rightarrow [\![\, ct \,]\!]_q = [\![\, ct \,]\!]_{q'} \tag{9.13}$$

$$\forall i \in \mathrm{N} : [\![\, nexttask \,]\!]_q([\![\, ct \,]\!]_q, i) = [\![\, nexttask \,]\!]_{q'}([\![\, ct \,]\!]_{q'}, i) \tag{9.14}$$

$$\forall i \in \mathrm{N} : [\![\, data \,]\!]_q([\![\, ct \,]\!]_q, i) = [\![\, data \,]\!]_q([\![\, ct \,]\!]_{q'}, i) \tag{9.15}$$

Die Interpretation der anderen Funktionen ist abgesehen von der Änderung des Programms gleich.

Zuweisung
Auch die Semantik der Zuweisung wird durch die Transformation spezialisiert. Es wird zwischen Basis- und strukturierten Objekten unterschieden und eventuell wird auch noch eine Typanpassung vorgenommen. Die Verifikation der Transformation funktioniert analog zur Verifikation bei Copy.

Schleifen
Die Verifikation der Abbildung von While und For ist analog und wurde schon in Abschnitt 7.4.3 angegeben.

Break und Continue
Eine Schleife kann mit Break oder Continue verlassen werden. Beide entsprechen einem unbedingten Sprung, den wir durch einen bedingten Sprung implementieren, dessen Bedingung immer erfüllt ist. Das Ziel des Sprungs und des bedingten Sprungs ist dabei dasselbe. Wir haben zu zeigen:

$$\begin{array}{ccc} q_1 & \xrightarrow{\mathsf{Jump}} & q_2 \\ \rho \big| & & \big| \rho \\ q_1' & \xrightarrow{\mathsf{One}} & q_2' \end{array}$$

Für die Verifikation argumentieren wir vor allem über *ct*. Ein Zustand $q' \in Alg(\Sigma_2)$ steht in Beziehung ρ zu einem Zustand $q \in Alg(\Sigma_1)$, wenn

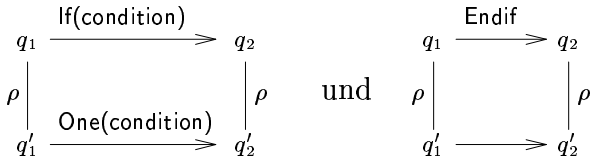
$$[\![\, ct \,]\!]_q \in [\![\, \mathsf{Jump} \,]\!]_q \Rightarrow [\![\, ct \,]\!]_{q'} \in [\![\, \mathsf{One} \,]\!]_{q'} \tag{9.16}$$

$$[\![\, ct \,]\!]_q \notin [\![\, \mathsf{Jump} \,]\!]_q \Rightarrow [\![\, ct \,]\!]_q = [\![\, ct \,]\!]_{q'} \tag{9.17}$$

$$[\![\, target \,]\!]_q([\![\, ct \,]\!]_q) = [\![\, nexttask \,]\!]_{q'}([\![\, ct \,]\!]_{q'}, 0) \tag{9.18}$$

Die Interpretation der anderen Funktionen ist abgesehen von der Änderung des Programms gleich.

Bedingte Anweisung
Bei der bedingten Anweisung haben wir zwei Dinge zu zeigen:



Für die Verifikation des linken Teils argumentieren wir vor allem über ct. Ein Zustand $q' \in Alg(\Sigma_2)$ steht in Beziehung ρ zu einem Zustand $q \in Alg(\Sigma_1)$, wenn

$$[\![ct]\!]_q \in [\![\mathsf{If}]\!]_q \Rightarrow [\![ct]\!]_{q'} \in [\![\mathsf{One}]\!]_{q'} \tag{9.19}$$

$$[\![ct]\!]_q \notin [\![\mathsf{If}]\!]_q \Rightarrow [\![ct]\!]_q = [\![ct]\!]_{q'} \tag{9.20}$$

Die Interpretation der anderen Funktionen ist abgesehen von der Änderung des Programms gleich.

Die Kommutativität des zweiten Diagramms ist sofort einsichtig, wenn man die Semantik des Endif-Kommandos betrachtet. Dabei wird nur ct weitergeschaltet. Bei der Transformation haben wir den Steuerfluß entsprechend umgesetzt. Ist t ein Vorgänger eines Endif-Kommandos t' und t'' der Nachfolger von t', dann wurde durch die Transformation $[\![t.NT(0)]\!] = t''$ gesetzt.

Operationen
Die Abbildung von komplexen Operationen ist entweder eine Spezialisierung oder bedeutet eine Dekomposition der Aktualisierungsmengen der entsprechenden ASM-Regel. In beiden Fällen können wir ein vordefiniertes Transformationsschema anwenden.

Abbildung von Store
Die Abbildung von Store ist eine Instanz des allgemeinen Transformationsschemas Spezialisierung und wurde schon in Abschnitt 8.2.5 verifiziert.

Elimination von Indeterminismus
Das All-Kommando modelliert eine indeterministische Auswertungsreihenfolge über seinen Elementen. Unser Korrektheitsbegriff für Übersetzungen erlaubt, daß das Zielprogramm deterministischer ist als das Quellprogramm. Das Zielprogramm muß nur eine der erlaubten Zustandsübergangsfolgen des Quellprogramms implementieren. Daher ist die Festlegung genau einer erlaubten Ausführungsreihenfolge eine korrekte Übersetzung indeterministischer Ausführungsfolgen. Diese Transformation ist auch ein allgemeines Transformationsschema, siehe Abschnitt 8.2.3.

9.5 Techniken zur Wiederverwendung

Spezifizieren wir eine Quellsprache, dann müssen wir eine Abbildung der Quellsprachkonzepte in die Sprache AL definieren. Kommt ein Quellsprachkonzept auch in der Bibliothek vor, dann ist diese Abbildung die Identität. Wir werden jedoch nicht jedes gewünschte Quellsprachkonzept in der Bibliothek finden. Allerdings können die Konzepte der Bibliothek zur Definition neuer Konzepte wiederverwendet werden. Die unterschiedlichen Techniken hierfür stellen wir in den nächsten Abschnitten vor.

9.5.1 Erweiterung und Anpassung von Sprachkonzepten

Oft unterscheidet sich das zu modellierende Konzept nur geringfügig von einem Konzept, das schon in der Bibliothek vorhanden ist. Durch Angabe zusätzlicher Funktionalität kann dann die gewünschte Semantik beschrieben werden. Einschränkungen der statischen Semantik sind ein Spezialfall hiervon. Alternativ kann die Semantik eines neuen Konzepts auch durch Komposition existierender Konzepte spezifiziert werden.

Beispiel 9.1 (Interpretation von Integer-Werten als boolesche Werte) In C können Integer-Werte als boolesche Werte interpretiert werden. Die Vorschrift hierfür ist einfach. Ist der Wert gleich Null, dann wird er als *false* interpretiert, sonst entspricht der Wert *true*. Angenommen, diese Semantik von Integer existiert noch nicht in der Bibliothek, dann kann sie einfach unter Benutzung von Vergleichsoperationen über Integer nachgebildet werden. Abbildung 9.16 zeigt den entsprechenden *AL*-Graphen. Zur Verifikation muß gezeigt werden, daß es eine Simulationsbeziehung zwischen den entsprechend interpretierten C-Integern und den booleschen Werten aus der Bibliothek gibt. Der Datentyp *Bool* ist in der Bibliothek enthalten, weil er Teil von *AL* ist. ♢

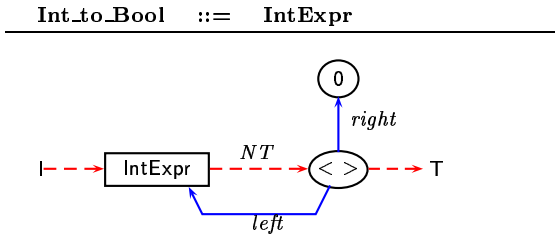


Abbildung 9.16: Interpretation von Integer-Werten als boolesche Werte

Die Komposition von Konzepten S und S′, wird über Beziehungen zwischen Kommandos aus S und S′ spezifiziert, die Steuerfluß oder Datenabhängigkeiten angeben. Diese Beziehungen drücken sich durch die statischen Funktionen *nexttask* und *data* aus. Wir können die Semantik beliebiger konkreter Quellsprachkonzepte durch Zusammenstecken existierender Konzepte modellieren. Das ist vom theoretischen Standpunkt gegeben, weil *AL* turingmächtig. Aus der Sicht des Praktikers garantieren das die Konzepte von *AL*, die unterschiedlichste Zwischensprachen subsummieren.

9.5.2 Generische Sprachkonzepte

Konzepte können allgemeiner wiederverwendet werden, wenn sie möglichst abstrakt definiert sind und wir nur unterschiedliche Instanzen davon bilden müssen. Das erreichen wir mit Hilfe generischer Sprachkonzepte, die wir schon in Kapitel 6 kennengelernt haben. Während wir bisher generische Sprachkonzepte zur Modularisierung der Verifikation genutzt haben und vor allem daran interessiert waren, die Semantik eines Konzepts so zu spezifizieren, daß die Semantik der Teilkonzepte generisch ist, kommt nun ein weiteres Kriterium hinzu. Als Konzept einer Bibliothek sollte ein generisches Sprachkonzept so definiert sein, daß es von den unterschiedlichen Nuancen in unterschiedlichen Sprachen abstrahiert und durch Instantiierung diese leicht unterschiedlichen Konzepte abgeleitet werden können. Die Entscheidung, was bei der

Definition eines Konzepts generisch und was daran konkret ist, ist eine Entwurfsentscheidung und hängt von der Weitsicht und der Erfahrung des Entwerfers ab.

Beispiel 9.2 (Schleife) Unser generisches Konzept While beschreibt bisher eine Pascal-Schleife. Angenommen das Konzept While ist zusammen mit booleschen Ausdrücken und dem Datentyp Integer Teil einer Bibliothek von Sprachkonzepten. Die Minimalanforderung für die Schleifenbedingung von While legt fest, daß die Schleifenbedingung eine binäre Entscheidung berechnen muß.

Betrachten wir nun eine neue Sprache, die neben booleschen Werten auch Integerwerte als Schleifenbedingung erlaubt. Integer-Werte werden entsprechend der Semantik in Abb. 9.16 als boolesche Werte interpretiert. In diesem Fall können wir die Spezifikation, sowie die Transformation von While in AL einfach wiederverwenden, indem wir Integerwerte mit Int_to_Bool als boolesche Werte uminterpretieren. Abbildung 9.17 veranschaulicht die Situation. Wir haben dann nur noch die Wohlgeformtheit der Spezifikation zu zeigen, also $[\![\,\mathsf{Int\_to\_Bool(IntExpr)}\,]\!] \prec_{\overline{p}} [\![\,\mathsf{BoolExpr}\,]\!]$. ♢

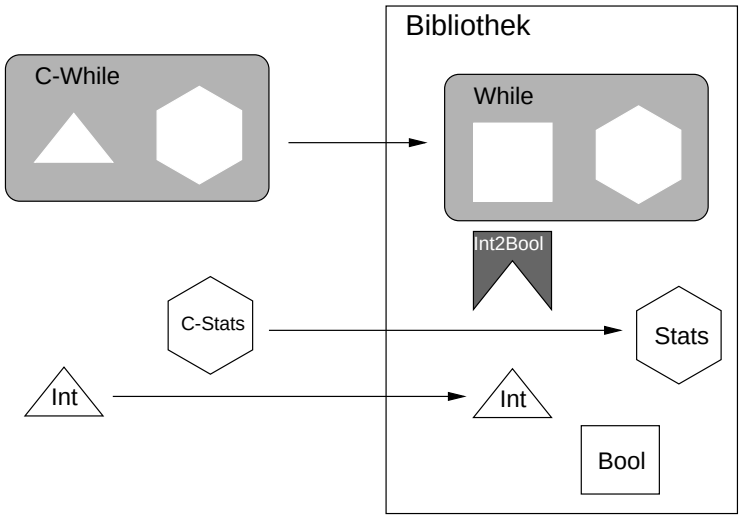


Abbildung 9.17: Wiederverwendung eines Schleifen-Konzepts

Die Verwendung von generischen Sprachkonzepten mit Minimalanforderungen an ihre Teilkonzepte hat neben der besseren Strukturierung von Spezifikationen den zusätzlichen Vorteil, daß über die Minimalanforderungen das Zusammenstecken von Sprachspezifikationen kontrolliert werden kann. Bei der Spezifikation muß nachgewiesen bzw. zugesichert werden, daß Minimalanforderungen bei der Instantiierung eingehalten werden, vgl. Kapitel 6.

9.5.3 Parametrisierte Sprachkonzepte

Parametrisierte Sprachkonzepte sind eine spezielle (eingeschränkte) Variante generischer Sprachkonzepte. Der Unterschied besteht darin, daß die Parameter hier nur noch eine bestimmte Ausprägung eines Sprachkonzepts festlegen. Die Wertebereiche der Parameter sind fest vorgegeben. Die Definition parametrisierter Konzepte hat den Vorteil, daß eine Transformation in Abhängigkeit dieser Parameter definiert und symbolisch verifiziert werden kann. Bei der Instantiierung wird überprüft, ob die Parameter innerhalb des erlaubten Wertebereichs liegen und damit die Instantiierung wohldefiniert ist. Simulationsbeweise zum Nachweis der Wohlgeformtheit sind überflüssig.

Beispiel 9.3 (Speicher von Maschinensprachen) Der sequentielle und über Adressen zugreifbare Speicher der Zielmaschine kann parametrisiert spezifiziert werden. Dazu werden für jeden Basistyp die Größe und die erforderliche Ausrichtung von Werten als Parameter definiert. Zugriff auf Inhalte von Speicherzellen erfolgt in Abhängigkeit des Typs. Die Speicherabbildung von *AL* in den Speicher der Zielmaschine kann ebenfalls in Abhängigkeit dieser Parameter angegeben und verifiziert werden (Abschnitt 7.4.2). ♢

Beispiel 9.4 (Integer- und Gleitkommadatentypen) *AL* definiert unterschiedliche Datentypen für *Int* und *Float*. Der Integer-Datentyp ist durch die Parameter *minint, maxint* und *arithmetic* beschrieben. *Float* ist entsprechend dem IEEE-Standard in Abhängigkeit der Anzahl der Bits definiert, die zur Darstellung verwendet werden. In beiden Fällen können wir eine Abbildung zwischen unterschiedlichen Ausprägungen des jeweiligen Datentyps definieren.

♢

Beispiel 9.5 (Funktionsaufruf) Unter Wiederverwendungsgesichtspunkten ist die ursprüngliche Definition des Funktionsaufrufs nicht allgemein genug. Eine Spezifikation, die auch noch generisch in der Auswertungsreihenfolge der Parameter ist, wäre sinnvoller. Wir können die Reihenfolge als Relation über den Argumenten definieren und haben damit von unterschiedlichen Auswertungsreihenfolgen abstrahiert. In diesem Fall stellt die Auswertungsreihenfolge einen Parameter dar, der eine Eigenschaft über Listen beschreibt. ♢

9.6 Fehlerhafte Verwendung von Konzepten

Für das Zusammenstecken einer Sprache besagt die Erfahrung des Übersetzerbauers bzw. Sprachdesigners, daß die unvorsichtige Kombination von bestimmten Sprachkonzepten zu unerwünschten Effekten führen kann. Die Spezifikation einer Sprache unter Verwendung von Sprachkonzepten mit Minimalanforderungen an ihre Teilkonzepte beugt dem auf zwei Arten vor. Der Entwerfer der Sprache ist einerseits gezwungen, die Semantik der Sprachkonzepte genau anzugeben und muß zusätzlich die Einhaltung der Minimalanforderungen nachweisen.

Wir haben drei Arten von Fehlern identifiziert.

1. Inkonsistenzen, die sich in der statischen Semantik ausdrücken,
2. Mehrdeutigkeiten in der dynamischen Semantik und
3. Inkonsistenzen in der dynamischen Semantik durch falsche Instantiierung von Teilkonzepten.

Für die Spezifikation der dynamischen Semantik setzen wir voraus, daß bestimmte statische Informationen existieren, die durch statische Funktionen der ASM spezifiziert sind. Definieren zwei Sprachkonzepte inkonsistente Signaturen für den gleichen Funktionsnamen, dann ist die kombinierte ASM nicht konsistent. In unseren Fallbeispielen fallen die meisten problematischen Kombinationen von Sprachkonzepten in diese erste Problemklasse. Mehrdeutigkeiten in der dynamischen Semantik können nicht als Fehler erkannt werden, da nicht ausgeschlossen werden kann, daß sie so intendiert waren. Die dritte Art von Problemen kann mit unserem Ansatz vermieden werden. Wir stellen die Konsistenz der Sprachspezifikation sicher, indem nachgewiesen werden muß, daß die Teilkonzepte, die verwendet werden, die minimalen Eigenschaften haben, die für sie angenommen wurden.

Am Beispiel der Kombination von Überladung und Generizität stellen wir ein Problem der ersten Klasse vor. Mit der inkorrekten Instantiierung eines generischen Funktionsaufrufs zeigen wir ein Beispiel aus der dritten Klasse.

9.6.1 Mehrdeutigkeit bei Überladung und Generizität

Wenn wir von Generizität und Überladung reden, dann sind das nicht Konzepte einer Sprache in unserem Sinne, sondern es handelt sich eher um Metakonzepte, von denen unterschiedliche Sprachkonzepte betroffen sind[1]. Die Definition von Klassen bzw. von Objekten und Funktionen wird von ihnen ebenso beeinflußt wie der Methodenaufruf.

Das folgende Beispiel zeigt, daß Mehrdeutigkeiten, die durch die Kombination von Überladung und Generizität in einer Sprache auftreten, sich in einer inkonsistenten statischen Semantik der Sprache widerspiegeln. Nehmen wir an, daß eine Sprache die Definition generischer Klassen mit Minimalanforderungen erlaubt. Dann könnte eine Klasse folgendermaßen aussehen:

```
class X(P) is
  f(x:P) is ... end
  f(x:T) is ... end
end
...
o : X(T)
o.f(t)
```

X ist eine generische Klasse mit Parameter P, und f ist überladen definiert. Das Problem tritt bei der Instantiierung der Klasse X auf. Zu dem Bezeichner f in o existieren nach der Instantiierung mit dem Typ T zwei unterschiedliche Definitionen mit gleicher Schnittstelle. An der Aufrufstelle $o.f$ kann anhand des Typs von f nicht entschieden werden, welche Funktion aufgerufen werden soll. Bei der formalen Spezifikation der dynamischen Semantik wird das Problem dann greifbar. Die Semantik des Funktionsaufrufs ist durch das Call-Kommando definiert.

```
if Call(ct) then
    ct := firsttask(ct.id)
    caller(reclevel + 1) := ct
    reclevel := reclevel + 1
endif
```

Durch die ASM-Funktion *firsttask* wird das erste Kommando der aufzurufenden Funktion identifiziert. Im imperativen Fall und bei dem im Anhang beschriebenen Funktionsaufruf ist *firsttask* Teil der statischen Semantik und wird bei der semantischen Analyse initialisiert. Wären nun Programme erlaubt, die zwei Funktionen mit der gleichen Schnittstelle haben, dann ist die statische Semantik des Funktionsaufrufs verletzt, weil dort erwartet wird, daß *firsttask* eine Funktion im mathematischen Sinne ist. Definieren wir dagegen *firsttask* in der semantischen Analyse als Relation, dann ist die Spezifikation der Sprache nur dann mathematisch konsistent, wenn in der dynamischen Semantik auch vorgesehen ist, daß *firsttask* die Eigenschaften einer Relation haben kann. Das heißt, bei der Verwendung von *firsttask* zum Aufruf einer Funktion könnte es mehrere Ergebnisse geben, die explizit unterschieden werden müssen. Im Funktionsaufruf von *AL* wird *firsttask* als Funktion verwendet. Würde *firsttask*

1) Das gleiche gilt auch für Polymorphie.

bei der Spezifikation eines anderen Konzepts als Relation verwendet, dann ist die Spezifikation insgesamt inkonsistent, weil die kombinierte statische Semantik der Sprachkonzepte nicht konsistent ist.

9.6.2 Fehlerhafte Instantiierung

Angenommen wir haben in unserer Bibliothek einen generisch definierten Funktionsaufruf, der die Berechnung seiner Argumente mit fester Auswertungsreihenfolge definiert. Die Minimalanforderung für die Argumentauswertung ist durch die Montage in Abb. 9.18 beschrieben. Die statische Semantik der Minimalanforderung ist über den Typ von *Arguments* abhängig von der Definition der aufgerufenen Funktion. Der Funktionsaufruf einer konkreten Sprache, der mit Hilfe dieses generischen Konzepts spezifiziert werden soll, definiert eine Argumentliste $Arguments_{QS}$ mit beliebiger Auswertungsreihenfolge. Für eine wohldefinierte Instantiierung muß nun entsprechend Abschnitt 6.4.1 nachgewiesen werden, daß der eingesetzte Parameter das beobachtbare Verhalten der Minimalanforderung simuliert. Wir müssen die Simulationsbeziehung für alle Berechnungsfolgen von $[\![\,\mathsf{Arguments}_{QS}\,]\!]$ nachweisen. Das ist jedoch nur möglich für Argumentlisten der Länge ≤ 1. In den anderen Fällen kann man leicht einen Widerspruchsbeweis führen. Da $\mathsf{Arguments}_{QS}$ eine beliebige Reihenfolge bei der Berechnung der Argumente erlaubt, wählt man $e : \mathsf{Expr}[i], i > 0$ als initiales Kommando von $\mathsf{Arguments}_{QS}$ ($\mathsf{Arguments}_{QS}.initial = e$). Die Simulationsbeziehung zu der Minimalanforderung Arguments kann nicht gezeigt werden, weil dort $\mathsf{Arguments}.initial = \mathsf{Expr}[0]$ gilt. $\mathsf{Arguments}_{QS}$ erfüllt damit nicht die Anforderung (A) aus Definition 6.11 und ist dementsprechend keine wohlgeformte Instanz.

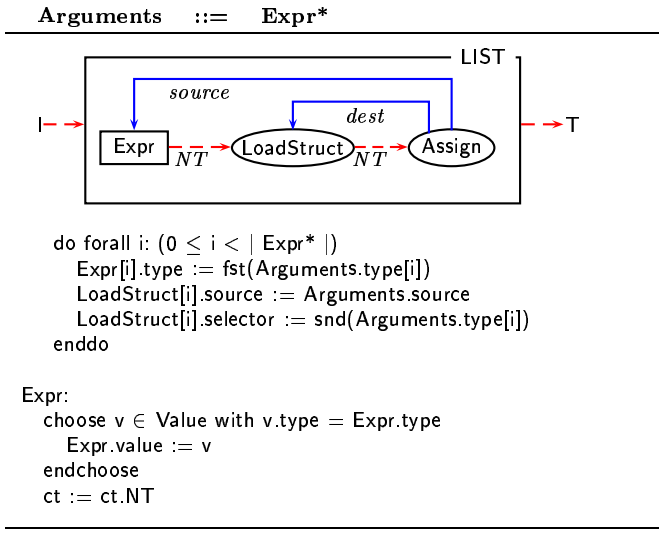


Abbildung 9.18: Berechnung von Funktionsargumenten mit fester Auswertungsreihenfolge

9.7 Zusammenfassung

Die Wiederverwendung von Sprachkonzepten aus unserer Bibliothek macht die Konstruktion von korrekten Übersetzern praktikabel. Sowohl die Spezifikation von Sprachen wird vereinfacht als auch der Verifikationsaufwand wird durch Wiederverwendung korrekter Transformationen

reduziert. Wir haben in diesem Kapitel eine Bibliothek imperativer Sprachkonzepte zusammen mit korrekten Übersetzungen vorgestellt. In der Bibliothek sind die üblichen imperativen Sprachkonzepte, wie Reihungen und Verbunde mit Werte- und Referenzsemantik, bedingte Anweisung, Zuweisung, While- und For-Schleife sowie Funktionen mit unterschiedlichen Parameterarten enthalten. Beim Entwurf der Bibliothek haben wir uns bekannte Ausprägungen imperativer Konzepte berücksichtigt. Die Bibliothek ist daher zur Spezifikation und Transformation von PASCAL- bzw. MODULA-ähnlichen Sprachen geeignet, kann aber auch für die imperativen Teile objektorientierter Sprachen wie JAVA oder SATHER-K eingesetzt werden.

Wir können auch objektorientierte Sprachkonzepte modellieren und übersetzen. Allerdings ist der Aufwand höher, da objektorientierte Konzepte bisher nicht durch die Bibliothek unterstützt werden. Allgemein gilt, daß Sprachkonzepte, die nicht in der Bibliothek vorhanden sind, vom Entwerfer „von Hand“ spezifiziert werden müssen. Zur Vereinfachung dieser Aufgabe haben wir Techniken zur Wiederverwendung vorgestellt. Wir nutzen dabei aus, daß Sprachkonzepte generisch bzw. mit festen Parametern definiert werden können. Da die Sprache *AL* Teil der Bibliothek ist, kann die Spezifikation beliebiger Programmiersprachen durch die Bibliothek unterstützt werden. Die Definition von Minimalanforderungen für die Teilkonzepte eines Sprachkonzepts strukturiert die Verifikation der Übersetzung und macht existierende Abhängigkeiten vom Kontext explizit, die bei der Verifikation der Übersetzung beachtet werden müssen. Gleichzeitig sind generisch definierte Sprachkonzepte mit Minimalanforderungen allgemeiner verwendbar, weil nur die Details der Teilkonzepte festgelegt werden, die für die Gesamtsemantik des Sprachkonzepts relevant sind. Es können alle konkreten Sprachkonzepte eingesteckt werden, die die Minimalanforderungen erfüllen. Die Einhaltung der Minimalanforderungen stellt die Wohlgeformtheit der Sprachspezifikation sicher.

Sprachkonzepte können natürlich nicht beliebig kombiniert werden, da es Abhängigkeiten gibt und Eigenschaften von Sprachkonzepten sich gegenseitig ausschließen. Unsere Untersuchung von Semantikspezifikationen hat gezeigt, daß Probleme bei der Kombination von Sprachkonzepten häufig die statische Semantik betreffen. Ein Beispiel hierfür sind mehrdeutige Definitionen für Funktions- oder Variablennamen, die nicht aufgelöst werden können. Eine fehlerhafte Spezifikation der statischen Semantik kann anhand der attributierten Grammatik erkannt werden, die die statische Semantik definiert. Existieren ungelöste Abhängigkeiten zwischen Attributen, dann kann keine Traversierungsstrategie abgeleitet werden, unterschiedliche Definitionen eines Attributnamens bedeuten, daß die Anforderungen von Sprachkonzepten an die statische Semantik widersprüchlich sind, und zu Attributen gibt es keine Definition, wenn die statische Semantik unterspezifiziert ist. Ist eine Sprachspezifikation nicht wohlgeformt, dann paßt die Instantiierung eines Sprachkonzepts nicht zu den Anforderungen an die Teilkonzepte und daher ist die resultierende Semantik nicht korrekt. Die Wohlgeformtheit ist also eine Eigenschaft zur Kontrolle einer Sprachspezifikation. Eine Unterspezifikation der dynamischen Semantik kann zu Mehrdeutigkeiten führen, bei denen nicht unterschieden werden kann, ob dies ein Fehler oder vom Entwerfer intendiert ist. Für die Übersetzung stellt dieser Fall kein Problem dar, da für die Korrektheit einer Übersetzung die Semantik exakt so (modulo Indeterminismus und Ressourcenbeschränkungen) abgebildet wird, inklusive Mehrdeutigkeiten, wie sie spezifiziert wurde.

10 Implementierung

Bisher haben wir uns nur mit der Spezifikation der Übersetzung und deren Korrektheit beschäftigt. In diesem Kapitel stellen wir die Umsetzung des Übersetzungskalküls in eine objektorientierte Bibliothek vor. Diese Bibliothek stellt Klassen zur Repräsentation von Sprachkonzepten (der Syntax des Kalküls) zur Verfügung und bietet Methoden bzw. Algorithmen zur Implementierungen der Transformation dieser Sprachkonzepte (der Regeln des Kalküls) an. Die Implementierung modelliert auch semantische Eigenschaften. Untertypsbeziehungen zeigen Simulationen und generische Parameter mit Typschranken codieren die Semantik von Teilkonzepten und Minimalanforderungen. Diese Modellierung versetzt uns in die Lage, das Typsystem der Implementierungssprache zur Konsistenzprüfung von Sprachspezifikationen einzusetzen.

Zusätzlich geben wir eine Korrektheitsdefinition für die Implementierung einer Transformation an, benennen Transformationen deren Korrektheit mittels Programmprüfung zugesichert werden kann und zeigen, wie Standarddatentypen (z. B. die Definitionstabelle) in die Implementierung integriert sind. An den jeweiligen Stellen gehen wir auf Abstraktionsmöglichkeiten ein und weisen auf Entwurfsentscheidungen hin.

Der Aufwand für die Verifikation der Implementierung sprengt den Rahmen dieser Arbeit. Nach der Definition von Vor- und Nachbedingungen ist die Programmverifikation eine Fleißarbeit, die keinen neuen Erkenntnisgewinn bedeutet.

10.1 Ein objektorientiertes Implementierungsschema

Eine Sprache ist durch ihre abstrakte Syntax repräsentiert. Die Knotentypen entsprechen Sprachkonzepten. Sie werden durch Klassen beschrieben. Die Spezifikation der Sprachsemantik legt die abstrakte Syntax fest und definiert damit welche Klassen zur Repräsentation benötigt werden. Programme der Sprache sind durch attributierte Strukturbäume repräsentiert. In der Implementierung sind Programme durch entsprechende Objektstrukturen dargestellt, die sich durch Instantiierung der Klassen für Knotentypen der abstrakten Syntax ableiten. Transformationen bauen diese Objektstrukturen um.

10.1.1 Sprachkonzepte

Sprachkonzepte bzw. Typen der abstrakten Syntax werden nach dem gleichen Schema implementiert.

Sei $G = (N, T, P, Z)$ die kontextfreie Grammatik, welche die abstrakte Syntax beschreibt. Dann werden für alle $n \in N$ und $t \in T$ Klassen definiert. Sei $N ::= N_1|N_2|\ldots$ eine Produktion in P und $N_i \to X_1 \ldots X_k$ dann implementieren wir die entsprechende Klasse folgendermaßen:

```
class AST_N_i like N is
  -- ASM::semantics( ... );
```

```
  x_1 : X_1;
  ...
  x_k : X_k;
  create (y_1:X_1, ... y_k:X_k) : N_i is ... end;

  define(context: PROCID; next: ARRAY[*]($TASK); out i, t: $TASK) is
    ...
  end
end
```

- `create` erzeugt einen AST-Knoten n vom Typ N, die Attribute $x_1 \ldots x_k$ werden von den entsprechenden Klassen der X_i definiert und sind Kinder von n. Sie werden von `create` initialisiert.
- `define` beschreibt die Abbildung des Sprachkonzepts in einen *AL*-Term und ist damit unter Umständen die Spezifikation der Semantik, aber auf jeden Fall die Implementierung der Transformation.
- `ASM::semantics` beschreibt die dynamische Semantik von N_i in Form einer generischen abstrakten Zustandsmaschine. Für die Implementierung selbst wird diese Beschreibung nicht benötigt, sehr wohl aber für die Verifikation.
- **like** drückt eine Untertypsbeziehung aus, **out** besagt, daß ein Parameter ein Ausgabeparameter ist. `$Typ` beschreibt einen abstrakten Typ.

Die Attribute einer Klasse ergeben sich aus der attributierten Grammatik, die die statische Semantik beschreibt. Aus Effizienzgründen wird die Typinformation üblicherweise über eine Definitionstabelle verwaltet tritt dann nicht in Form von Attributen einer Klasse auf.

Die Spezifikation für N ist vollständig, wenn `semantics` und `define` für alle Unterklassen und Alternativen angegeben sind.

Benutzen wir ein Werkzeug wie z. B. GEM-MEX, dann definiert dieses Werkzeug für jede Regel der abstrakten Syntax die Abbildung in die Klassenstruktur. Unser Ziel ist, daß alle Konzepte einer Sprache durch die Bibliothek unterstützt werden, so daß bei der Spezifikation nur noch eine 1:1-Abbildung der Quellsprache in *AL*-Konzepte angegeben werden muß.

10.1.2 Implementierung von *AL*

Die semantischen Konzepte aus *AL* werden entsprechend unserem Implementierungsschema auf Klassen abgebildet. Allerdings sind diese Klassen, abgesehen von den parametrisierten Datentypen, konkret definiert. Das ist einleuchtend, da die Semantik der Kernsprache *AL* fix ist. Kommandos haben bestimmte Eigenschaften. Sie haben einen statischen Typ aus dem Typsystem von *AL*, eine Menge von Nachfolgern und auf jeden Fall auch eine Menge von Datenabhängigkeiten. Steuer- und Datenflußabhängigkeiten sind in der Semantik durch die Funktionen

$$\begin{aligned} \mathit{nexttask}&: \mathit{Task} \times \mathrm{N} \to \mathit{Task} \\ \mathit{data}&: \mathrm{N} \times \mathit{Task} \to \mathit{Task} \end{aligned}$$

beschrieben. Sie werden als eine Reihung von Kommandos implementiert.

Die Methode `transform` bildet einen Kommando-Knoten in eine Zwischensprachdarstellung ab bzw. initiiert die Abbildung eines Teilgraphen. Das ist die konsequente Fortführung der

Semantikdefinition, nur werden dieses Mal *AL*-Kommandos in Termen der Semantik der Zwischensprache definiert.

```
abstract class TASK IS
  statictype : $AL_TYPE;
  nexttask : ARRAY[*]($TASK);
  data : ARRAY[*]($TASK);
  initial, terminal : $TASK;

  transform : $MIS_TERM IS ... end;
end; -- class TASK
```

10.1.3 Abstraktion von Sprachkonzepten

Mit dem in Abschnitt 10.1.1 vorgestellten Schema lassen sich alle Konzepte einer Sprache beschreiben. Nimmt man für jede Sprache die exakten Konstrukte in die Bibliothek auf, dann hätte man zumindest Abbildungen der Sprachen wiederverwendbar gemacht. Definieren wir Sprachkonzepte jedoch generisch, können sie zur Definition unterschiedlicher Sprachen verwendet werden, und gleichzeitig sind lokale Korrektheitsbeweise von Transformationen möglich. Generische Sprachkonzepte werden direkt in generische Klassen abgebildet. Typschranken definieren Einschränkungen an die Parameter.

Beispiel 10.1 Sei eine Schleife so definiert, daß die Schleifenbedingung und der Schleifenrumpf generische Parameter sind, deren Minimalanforderungen durch Typschranken vorgegeben sind. Eine mögliche Implementierung wäre dann

```
class Task_WHILE [ E < Min_EXPR(BOOL), S < STATS ] is like TASK
   expr : E;
   stats : S;
   nexttask: ARRAY[*]($TASK);
   initial: SET($TASK);
   terminal : $TASK;
   inrange: ARRAY[*](BOOL);
   create (e: E, s: S) : Task_WHILE is
     result := new Task_WHILE [ E, S ]
     result.expr:=e;
     result.stats:=s;
     o:=new ONE (nexttask);
     o.condition_task:=e.terminal;
     o.inrange[1]:=true;
     o.nexttask[0]:=nexttask[0],
     o.nexttask[1] :=s.initial;
     s.initial.key:=true;
     result.initial := e.initial;
     result.terminal:= o
   end
end
```

Eine Schleife einer anderen Quellsprache kann dann unter Benutzung von `Task_While` mit einer konkreten Instantiierung von `E` und `S`, unter Einhaltung der Typschranken, definiert werden.

Angenommen wir wollen eine Schleife für eine C-ähnliche Sprache definieren, die ganze Zahlen als Schleifenbedingung akzeptiert, dann würden wir zusätzlich einen Umwickler (engl.:

wrapper) benutzen, der die folgende Interpretation von ganzen Zahlen als boolesche Werte definiert:

$i \neq 0 \rightarrow \mathit{true}$
$i = 0 \rightarrow \mathit{false}$

Die Semantik der Schleife bleibt davon unbeeinflußt.

```
class Task_INT_TO_BOOL [ I < Min_EXPR(INT) ] is like EXPR(BOOL)
   expr : I;
   nexttask: ARRAY[*]($TASK);
   initial: SET($TASK);
   terminal : $TASK;
   create (e: I) : Task_INT_TO_BOOL is
     result := new Task_INT_TO_BOOL [ I ]
     result.expr:=e;
     i := new NEQ(INT) (nexttask);
     i.left := e.terminal;
     i.right := new INT::0;
     result.terminal := i;
     result.initial := e.initial;
   end
end
```

Die Definition der C-Schleife unter Benutzung von `Task_While` sieht dann folgendermaßen aus:

```
class AST_C_WHILE is like C_STAT
   expr : C_EXPR;
   stats : C_STATS;
   create (e: C_EXPR,s: C_STATS) : C_WHILE is ... ;
   ASM::semantics;
   define(in nexttask:ARRAY[*]($TASK); out initial:SET($TASK); terminal:$TASK) is
      task_expression := expr.define(nexttask, e_initial, e_terminal);
      task_wrapper := new Task_INT_TO_BOOL [ Task_EXPR(INT) ] (task_expression);
      task_stats := stats.define(task_expression, s_initial, s_terminal)
      task_while := new Task_WHILE [ INT_TO_BOOL [ Task_EXPR(INT) ], Task_STATS ]
                        ( task_wrapper, task_stats );
      terminal := task_while.terminal;
      initial :=  task_while.initial;
   end
end
```

wobei `C_EXPR` auf `Task_EXPR(INT)` und `C_STATS` auf `Task_Stats` abgebildet werden. Die Abbildungen 7.8, 9.16 und 9.17 veranschaulichen den Sachverhalt. ♢

10.1.4 Transformationen

Wir haben unterschiedliche Arten von Übersetzungsschritten.

1. Die Transformation eines Sprachkonzepts ist durch Graphersetzungsregeln spezifiziert und wird durch eine Methode der entsprechenden AST-Klasse implementiert. Diese Methode baut auf den Transformationen der Teilkonzepte auf und definiert die Reihenfolge

der Transformationen. Die Implementierung wird gegenüber der Spezifikation verifiziert. Sie baut die Objektstruktur (den Programmgraphen) um, welche ein konkretes Programm repräsentiert. Dabei können neue Objekte (Ecken) eingeführt und existierende Objekte gelöscht werden. Zusätzlich können Daten- und Steuerflußabhängigkeiten (Kanten) hinzugefügt oder weggenommen werden.

2. Die Abbildung des Speicherzustands (siehe Abschnitt 7.4.2) ist eine Transformation der zweiten Art von Übersetzungen. Bei der Spezifikation haben wir keine konkrete Abbildungsfunktion angegeben, sondern nur bestimmte Eigenschaften der Ergebnisse dieser Funktion gefordert. Diese Eigenschaften muß auch das Resultat der Implementierung erfüllen. Anstatt nun nachzuweisen, daß ein konkreter Algorithmus diese Eigenschaften an das Ergebnis zusichert, verwenden wir den Programmprüfungsansatz (Goerigk et al., 1998a). Das führt zu einer zweistufigen Implementierung. Der Algorithmus wird unverifiziert implementiert. Zusätzlich wird jedoch ein Überprüfer für das Ergebnis implementiert. Im Beispiel der Adreßabbildung überprüfen wir bestimmte Eigenschaften der erzeugten Adressen (Eindeutigkeit, Ausrichtung, Überlappungsfreiheit). Der Überprüfer ist mit der Ausrichtung und den Größen von Grundtypen parametrisiert.

3. Die letzte Klasse von Transformation betrifft die Abbildung von Basisdatentypen. Diese Transformationen sind zwar in Abhängigkeit von Parametern der Quelle und des Ziels implementiert, können aber unabhängig definiert werden. Sie werden einmal symbolisch für beliebige Parameter als korrekt nachgewiesen. Für den Integerdatentyp sind die Parameter z. B. Minimum, Maximum und Arithmetik.

10.1.5 Hilfsdatenstrukturen

Statische Funktionen der Semantik beschreiben Informationen, die sowohl zur abstrakten Interpretation eines Programms als auch zur Transformation benutzt werden. Dazu gehört z. B. Typinformation. Diese Information ist Teil des Initialzustands der ASM und wird während der semantischen Analyse berechnet. In *AL* gilt, daß alle Informationen direkt in dem Programmgraphen dargestellt sind bzw. als Attribute in den Ecken des Graphen gespeichert werden. Daher ist es naheliegend diese Information als Attribute der Knotenobjekte zu implementieren. Allerdings kann man sich auch eine effizientere Implementierung dieser statischen Information in Form einer Typ- und/oder Definitionstabelle vorstellen. In diesem Fall muß die Implementierung sicherstellen, daß Anfragen mit bestimmten Parametern das gleiche Ergebnis liefern, wie der direkte Attributzugriff innerhalb eines Objekts.

10.2 Implementierungskorrektheit

Zum Nachweis der Korrektheit einer Implementierung verwenden wir zwei unterschiedliche Ansätze. Wir verifizieren die Implementierung von Ersetzungsregeln mit den bekannten Programmverifikationstechniken nach Hoare und Dijkstra. Dazu leiten wir aus der linken Seite einer Ersetzungsregel eine Vorbedingung für die Transformation ab, die rechte Seite definiert die Nachbedingung. Diese Vor- und Nachbedingung übertragen wir auf die Implementierung und beweisen, mit Hilfe eines Verifikationskalküls der Implementierungssprache, die Überführung der Vor- in die Nachbedingung. Ist die Verifikation eines Überprüfers für das Resultat eines Übersetzungsschrittes einfacher durchzuführen als die Verifikation der Transformationsmethode, dann können wir alternativ die Korrektheit der Implementierung durch Programmprüfung

sicherstellen. Das hat den Vorteil, daß Implementierungen von Transformationen auch unverifiziert generiert und sogar ausgetauscht werden können, ohne die Korrektheit des Systems zu gefährden.

Das Zusammenfügen der Übersetzung ergibt sich durch das induktive Zusammenspiel von `define` bzw. `transform` Methoden. Die Korrektheit der Implementierung ergibt sich wieder induktiv aus lokalen Korrektheitseigenschaften, da jede Transformation auf der lokalen Korrektheit der Teiltransformationen aufbaut. Im letzten Schritt ($Program ::= \ldots$) wird die Simulationsbeziehung für komplette Programme nachgewiesen.

Ein Vorteil der Verwendung einer OO-Sprache mit beschränkter Generizität ist, daß wir das Typsystem der Implementierungssprache zur Überprüfung von Sprachspezifikationen einsetzen können. Sind die Instantiierungen korrekt und werden die Typschranken eingehalten, dann ist die Sprachspezifikation konsistent.

10.2.1 Korrektheit durch Programmverifikation

Die Spezifikation einer Transformation ist durch eine Graphersetzungsregel bzw. durch eine Termersetzungsregel angegeben. Die linke und die rechte Seite einer Ersetzungsregel definieren die Vor- und Nachbedingung der Ersetzung. Beide Bedingungen sind Prädikate über Programmgraphen, deren Ecken Kommandos und deren Kanten Daten- und Steuerflußabhängigkeiten sind. Die Vorbedingung P wird durch Ausführung der Termersetzungsregel in die Nachbedingung Q überführt. Genauer beschreibt die linke Seite einen Ausschnitt aus einem Graphen, der in einen Teilgraphen umgebaut wird, der durch die rechte Seite beschrieben ist.

$$\{\ P(G_{alt})\ \}\quad l \rightarrow r \quad \{\ Q(G_{neu})\ \}$$

Unser Implementierungsschema aus dem vorigen Abschnitt definiert einen Isomorphismus $\mathcal{I}$ von Kommandographen auf Objektstrukturen. Wir bilden:

- Kommandosorten auf Klassen,
- Kommandos einer Sorte auf Objekte der entsprechenden Klasse,
- Kanten auf Attribute und
- andere statische Information, wie z. B. Typinformation, ebenfalls auf Attribute

ab.

Sei ein Graph durch die Eckenmenge V und Relationen $data(\mathrm{N}, V, V)$ und $nexttask(\mathrm{N}, V, V)$ über dieser Knotenmenge beschrieben, dann sind $P(G_{alt})$ und $Q(G_{neu})$ Prädikate über Relationen. Die Vor- und Nachbedingung für die Implementierung sind Prädikate über dem isomorphen Bild des Graphen, wobei die Klassenattribute *data* und *nexttask* die Relationen gleichen Namens repräsentieren. Damit läßt sich jedoch sofort eine isomorphe Abbildung $\mathcal{I}$ auf den Vor- und Nachbedingungen P und Q ableiten und wir erhalten die Formeln $\mathcal{I}(P(G_{alt}))$ und $\mathcal{I}(Q(G_{neu}))$. Diese Formeln beschreiben Aussagen über den Objekten und ihren Attributen, die den Programmgraphen in der Implementierung repräsentieren.

Sei nun `transform` eine Implementierung der Ersetzungsregel $l \to r$ über den Objektstrukturen $\mathcal{I}(G_{alt})$ und $\mathcal{I}(G_{neu})$, dann haben wir die folgende Beziehungen zu zeigen.

$$\begin{array}{ccc} \{P(G_{alt})\} & l \to r & \{Q(G_{neu})\} \\ \mathcal{I}\Big\Updownarrow & & \Big\Updownarrow\mathcal{I} \\ \{\mathcal{I}(P(G_{alt}))\} & \texttt{transform} & \{\mathcal{I}(Q(G_{neu}))\} \end{array}$$

10.2.2 Korrektheit durch Resultatsprüfung

Wir betrachten ein Programm π mit Eingabe x und Ausgabe y. Sei $P(x)$ eine Vorbedingung von π und $Q(x, y)$ eine Nachbedingung. Das Programm π ist genau dann **partiell korrekt**, wenn für alle x, die $P(x)$ erfüllen, π entweder x zurückweist oder eine Ausgabe y berechnet für die $Q(x, y)$ gilt. In Hoare-Notation würde man $\{P(x)\}\ y := \pi(x)\ \{Q(x, y)\}$ schreiben. Die Idee zur Programmprüfung wird durch die folgende Funktion π' beschrieben:

```
π'(x : T): T' is
  y := π(x);
  if prüfe_π(x,y) then return y;
  else Error
  endif
end
```

Die boolesche Funktion $prüfe_\pi$ muß die Nachbedingung Q implizieren. π und $prüfe_\pi$ dürfen die Eingabe nicht verändern, müssen also seiteneffektfrei sein. Die Funktion π' implementiert eine Art Bootstrapping-Ansatz, um partiell korrekte Programme zu erzeugen. Wir erlauben, daß Ergebnisse zurückgewiesen werden, obwohl sie korrekt sind. Das geschieht genau dann, wenn der Resultatsprüfer die Korrektheit eines Ergebnisses nicht verifizieren kann. Die Anwendung dieser Technik ist sinnvoll, wenn die Verifikation der Funktion $prüfe_\pi$ sehr viel einfacher oder weniger aufwendig ist, als die Verifikation von π. Eine ausführlichere Darstellung dieses Ansatzes zur Implementierungsverifikation gibt (Goerigk et al., 1998a). In (Heberle et al., 1999) bzw. Gaul et al. (1999) sind Anwendungen für die Analyse- und die Codeerzeugung beschrieben. (Dold und Vialard, 1999) beschreibt den formale Nachweis der Korrektheit der Codeerzeugung mit PVS.

Beispiel 10.2 (Speicherabbildung) Die Speicherabbildung wurde in Abschnitt 7.4.2 durch die Angabe von Eigenschaften spezifiziert, die hinreichend für die wir zum Nachweis der Korrektheit der Abbildung benötigt haben.

Die Implementierung besteht dann aus zwei Teilen:

1. Der Implementierung der Speicherabbildung und
2. einer Routine, die die geforderten Eigenschaften des Ergebnisses der Speicherabbildung überprüft.

Während die Speicherabbildung nicht verifiziert werden muß, wird die Korrektheit des Überprüfers durch Programmverifikation nachgewiesen. Die Vor- und Nachbedingungen an den Überprüfer sind durch die Eigenschaften der Speicherabbildung definiert. Alle Relativadressen

eines Programms werden auf Eindeutigkeit, Überlappungsfreiheit und korrekte Ausrichtung überprüft. ♢

Die Anwendung von Programmprüfung hat neben der einfacheren Programmverifikation noch weitere Vorteile. Einerseits können Implementierungen ausgetauscht werden, andererseits können diese Implementierungen unverifiziert generiert und sogar in einer anderen Implementierungssprache erstellt werden.

10.3 Zusammenfassung

In diesem Kapitel haben wir ein objektorientiertes Implementierungsschema für unseren Übersetzungskalkül vorgestellt. Wir haben dabei semantische Eigenschaften direkt in Konzepte der Implementierungssprache umgesetzt. Zum Beispiel wird die Kapselung von Teilkonzepten durch Generizität implementiert und Simulationsbeziehungen drücken sich durch Untertypsbeziehungen aus. Bestimmte Eigenschaften der Implementierungssprache vereinfachen die Umsetzung der Spezifikation in ein konkretes Programm. Aus diesem Grund haben wir auch die Sprache Sather-K gewählt, da sie die benötigten objektorientierten Konzepte unterstützt, während z. B. JAVA in der Standardversion keine Generizität kennt. Ein weiterer Vorteil der objektorientierten Implementierungssprache ist, daß wir existierende Arbeiten zur korrekten Konstruktion von Klassen-Bibliotheken für unsere Verifikationszwecke ausnutzen konnten.

Jede Implementierung von neuen Transformationen muß gegenüber der Spezifikation verifiziert werden. Dafür muß man den mühseligen Prozeß der Programmverifikation mit den bekannten Techniken von Hoare und Dijkstra in Kauf nehmen. Eine Erleichterung schafft in bestimmten Fällen die Technik der Programmprüfung, die wir für die Speicherabbildung anwenden. Wir nutzen dabei aus, daß die Überprüfung des Ergebnisses eines Algorithmus einfacher zu verifizieren ist als der unter Umständen hoch komplexe Algorithmus, der dieses Ergebnis erzeugt hat. Dieser Ansatz gibt uns auch die Freiheit unterschiedliche Implementierungen bzw. unterschiedliche Algorithmen zur Lösung eines Problems einzusetzen.

Die Struktur der erzeugten Übersetzer ist vergleichbar mit der Struktur von Übersetzern, die objektorientiert entworfen und implementiert wurden wie z. B. die Übersetzer für SATHER (STOUTAMIRE und OMOHUNDRO, 1996), PIZZA (ODERSKY und WADLER, 1997) und GENERICJAVA (BRACHA ET AL., 1998).

11 Zusammenfassung und Ausblick

In dieser Arbeit haben wir einen Rahmen zur Konstruktion von korrekten Übersetzern vorgestellt, die reale imperative Programmiersprachen, wie PASCAL oder ANSI-C, in maschinennahe Zwischensprachen übersetzen.

11.1 Einordnung und Ergebnisse der Arbeit

Die Transformation der Quell- in die Zielsprachsemantik stellt den zentralen Übersetzungsschritt in einem Übersetzer dar. Dort findet der Übergang von der semantischen Welt der Quellsprache in die semantische Welt der Zielmaschine statt. Daher bestimmt die Transformationsphase auch die Aufgaben der Analyse- und der Codeerzeugungsphase. Aufgrund der Unterschiedlichkeit der Aufgaben der einzelnen Phasen kommen unterschiedliche Beweismethoden bei der Verifikation zum Einsatz. In der Analysephase werden Programmtexte auf die abstrakte Struktur abgebildet, die die Semantik repräsentiert. Einem Programmtext selbst ist keine Semantik zugeordnet, daher kann dieser Schritt nicht verifiziert werden. Die Korrektheit wird durch Überprüfung des Resultats der Analyse sichergestellt. In der Codeerzeugungsphase findet ebenfalls kein Wechsel der semantischen Welt statt, und die Korrektheit kann ebenfalls durch die verifizierte Überprüfung der Resultate garantiert werden. Die Verifikation der Transformation wird mit Simulationsbeweisen gegenüber der Quell- und Zwischensprachsemantik durchgeführt. Die Definition dieser Semantik ist an die abstrakte Syntax geknüpft. Jedem wohlgeformten abstrakten Strukturbaum wird durch Angabe eines abstrakten Interpretierers eine operationelle Semantik zugeordnet. Statische Informationen, die für die Interpretation benötigt werden, müssen in der semantischen Analyse berechnet werden.

Wir haben gezeigt, daß die Semantikspezifikation imperativer Programmiersprachen generalisiert werden kann, weil die semantischen Kernelemente dieser Sprachen sprachunabhängig sind. Diese Generalisierung resultiert in einer Spezifikationssprache für operationelle Semantik, die wir in Kapitel 5 beschrieben haben und die wir zur Wiederverwendung von Sprachspezifikationen und korrekten Übersetzungen ausnutzen.

Die strukturelle Definition einer operationellen Programmiersprachsemantik und unsere Technik zum modularen Korrektheitsnachweis von Übersetzungen haben wir in den Kapiteln 6 und 7 dargelegt. Dort haben wir auch Konsistenzbedingungen in Form von Minimalanforderungen an Sprachkonzepte eingeführt, mit denen eine Sprachspezifikation auf Widerspruchsfreiheit geprüft werden kann.

Wir haben Transformationsschemata identifiziert und verifiziert, die in jedem Übersetzer vorkommen und die zur Definition neuer korrekter Transformationen wiederverwendet werden können (Kapitel 8). Damit reduzieren sich die Beweisverpflichtungen bei der Definition neuer Transformation auf ein Minimum. Man hat nur zu zeigen, daß es sich bei der neuen Transformation um die Instanz eines bekannten Schemas handelt.

Zusätzlich haben wir Anforderungen definiert und semantisch begründet, unter denen die Semantik einer Programmiersprache aus einer Bibliothek wiederverwendbarer Komponenten

zusammengesetzt werden kann. Die semantischen Konzepte der Bibliothek und die Techniken zur Komposition haben wir in Kapitel 9 angegeben. Mit Hilfe der Bibliothek ist es möglich, (verifizierte) Transformationen semantischer Konzepte wiederzuverwenden und so die Transformationsphase zu erzeugen. Mit unserem Ansatz wird die Transformationsphase von Übersetzern wiederverwendbar, was nicht nur für die Konstruktion verifizierter, sondern auch für die Konstruktion unverifizierter Übersetzer ein Fortschritt ist.

Im Rahmen des Gesamtprojekts der Arbeit wird eine verifizierte Implementierung der Spezifikationsbibliothek erzeugt. In der Fallstudie wird damit ein korrekter Übersetzer erzeugt, der eine PASCAL-ähnliche Quellsprache in Maschinencode der DEC-Alpha übersetzt.

Wir haben mit unserem Rahmen alle Anforderungen zur Konstruktion von realistischen **und** verifiziert implementierten Übersetzern erfüllt, die wir am Anfang der Arbeit identifiziert hatten. Die Semantik der Quell- und der Zwischensprachen ist operationell und im gleichen formalen Rahmen spezifiziert. Die Spezifikation der Semantik und die Verifikation von Transformationen sind modular und können wiederverwendet werden. Sogar die maschinelle Überprüfung von Beweisen durch PVS ist möglich. Mit der operationellen Semantik und einem geeigneten Korrektheitsbegriff können wir Optimierungen angeben und verifizieren. Da auch die Zwischensprachen frei und passend zur verwendeten Codeerzeugungstechnik bzw. zur Zielmaschine gewählt werden können, haben wir alle Voraussetzungen erfüllt, um effizienten Maschinencode zu erzeugen.

11.2 Ausblick und zukünftige Arbeiten

Die zukünftigen Arbeiten sollen einerseits die Anwendung unseres Ansatzes weiter vereinfachen und andererseits das Anwendungsspektrum vergrößern.

Die Bibliothek vordefinierter Spezifikationen und verifizierter Transformationen ist für die Konstruktion zentral. Als nächstes wollen wir objektorientierte Sprachkonzepte und weitere Zwischensprachen einführen. Erste Untersuchungen und Beispielspezifikationen von OO-Konzepten haben gezeigt, daß sie vor allem aus Sicht der statischen Semantik Erweiterungen notwendig machen. Zur Definition der dynamischen Semantik kann jedoch auf existierende imperative Konzepte zurückgegriffen werden.

Eine große Vereinfachung der Implementierungsverifikation und die Möglichkeit zur Verwendung unverifizierter Algorithmen ergibt sich durch den Einsatz von Programmprüfungstechniken. Bisher überprüfen wir nur die Ergebnisse der Speicherabbildung. Es ist noch zu untersuchen, inwieweit dieser Ansatz auch für andere Übersetzungen der Transformationsphase trägt.

In Zusammenhang mit der Definition und Verifikation von Optimierungen ist vor allem die Übersetzung von *AL* in die Zwischensprache FIRM (TRAPP ET AL., 1999) interessant. Auf dieser Repräsentation können effektive Optimierungen, speziell für objektorientierte Programme, effizient ausgeführt werden (TRAPP, 1999). FIRM-Programme sind Graphen über Instruktionen, zwischen denen Daten- und Steuerflußabhängigkeiten existieren. Die Ähnlichkeit zu unserer Sprache *AL* ist offensichtlich und macht uns optimistisch, daß wir auf der Ebene von *AL* effektive Optimierungen definieren und verifizieren können.

Aus Semantiksicht gibt es zwischen zwei korrekten Übersetzungen eines Sprachkonzepts keinen Unterschied. Aus Optimierungssicht ist es aber durchaus erwünscht, daß ein Sprachkonzept

von einem Übersetzer, abhängig von Kontext, unterschiedlich übersetzt wird. Die Auswahl der jeweiligen Übersetzung hängt dabei von den jeweiligen Kosten ab. Als Kostenkriterium bietet sich hier vor allem die Größe des erzeugten Codes an, da auf der Ebene der Zwischensprache noch keine Aussagen über tatsächliche Ausführungskosten gemacht werden können. Gibt es mehrere Transformationsmöglichkeiten, dann sollte kostengesteuert zwischen diesen ausgewählt werden. Damit ergibt sich jedoch ein Optimierungsproblem, das noch genauer untersucht werden muß. Ein erster Ansatzpunkt ist die Übertragung von Techniken, die bisher zur Generierung von Übersetzer-Backends eingesetzt werden. Dort werden mehrere Übersetzungen mit unterschiedlichen Kosten für ein Sprachkonzept angegeben (Boesler, 1998; Proebsting, 1995).

Wichtig ist auch die Automatisierung und Werkzeugunterstützung unseres Ansatzes. Mit Gem-Mex (Anlauff et al., 1999) existiert schon ein graphisches Werkzeug, mit dem Montages-Spezifikationen für Sprachen angegeben und automatisch ein Übersetzer nach C inklusive Scanner, Parser und semantischer Analyse erzeugt werden kann. Darauf könnte ein Werkzeug zur Generierung von Übersetzern aufsetzen. Könnte man in Gem-Mex zusätzlich die Transformationen in eine bestimmte Zwischensprache auswählen, dann wären wir in der Lage, aus dieser Spezifikation automatisch eine korrekte Transformation abzuleiten. Die Voraussetzung dafür ist die Wohlgeformtheit der Sprachspezifikation. Ein Teil der Bedingungen für Wohlgeformtheit kann statisch durch das Typsystem der Implementierungssprache überprüft werden. Es ist allerdings eine offene Frage, wieviele dieser Bedingungen automatisch und damit von einem Werkzeug überprüft werden können.

Die letzte Klasse zukünftiger Arbeiten befaßt sich mit der breiteren Anwendung unserer Theorien zur modularen Spezifikation und Verifikation von Transformationen. Es scheint, daß man die vorgestellten Techniken auch zur Spezifikation der Semantik von Entwurfsmustern und zur Ableitung einer korrekten Implementierung dieser Entwurfsmuster anwenden kann. Diese Feststellung ist naheliegend, da man Entwurfsmuster als höhere Sprachkonzepte ansehen kann (Assmann et al., 1999), die eine Erweiterung von Programmiersprachen definieren.

Für die Konstruktion korrekter objektorientierter Bibliotheken können die theoretischen Ergebnisse dieser Arbeit ebenfalls hilfreich sein. Wir haben die Korrektheit der Konstruktion unter bestimmten Einschränkungen an die Verwendung der vordefinierten Komponenten nachgewiesen. Nimmt man diese Einschränkungen als Kriterium zur Kontrolle einer objektorientierten Anwendung mit Bibliothekskomponenten, dann können damit kritische bzw. fehlerhafte Verwendungen der Komponenten festgestellt werden.

Eine weitere Anwendung beruht auf der Generalisierung von Semantikspezifikationen. Das Interesse am Internet und damit einhergehend an der Beschreibungssprache XML, die man zur Beschreibung der Struktur von Dokumenten einsetzt, wird immer größer. Offen ist jedoch noch die Frage der Semantik von XML-Dokumenten und die korrekte Konvertierung zwischen Dokumenttypen. Auf der Basis einer gemeinsamen Semantiksprache ließe sich mit einer Bibliothek von Transformationen automatisch eine korrekte Konvertierung zwischen unterschiedlichen XML-Dokumenttypen ableiten.

Beware of bugs in the above code; I have only proved it correct, not tried it.
D.E. Knuth

Literaturverzeichnis

Assmann, U., Heberle, A., Löwe, W. et al. (1999): Language Concepts and Design Patterns. Working Paper.

Aho, A. V., Sethi, R. und Ullman, J. D. (1986): *Compilers: Principles, Techniques and Tools*. Addison-Wesley.

Aho, A. V., Sethi, R. und Ullman, J. D. (1988): *Compilerbau*, Bd. 1. Addison-Wesley.

Anlauff, M., Kutter, P. und Pierantonio, A. (1999): Tool Support for Language Design and Prototyping with Montages. In *Proceedings of Compiler Construction (CC'99)*, Lecture Notes in Computer Science. Springer.

Balzert, H. (1998): *Lehrbuch der Software-Technik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*. Spektrum Akademischer Verlag.

Blum, M. und Kannan, S. (1989): Programs That Check Their Work. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*.

Boesler, B. (1998): *Codeerzeugung aus Abhängigkeitsgraphen*. Diplomarbeit, Universität Karlsruhe.

Börger, E. (1999): High Level System Design and Analysis using Abstract State Machines. In *Current Trends in Applied Formal Methods (FM-Trends 98*, herausgegeben von Hutter, Stephan, Traverso und Ullmann, LNCS. Springer.

Börger, E. und Durdanović, I. (1996): Correctness of compiling Occam to Transputer Code. *Computer Journal*, 39(1): 52–92.

Börger, E. und Rosenzweig, D. (1994): The WAM – Definition and Compiler Correctness. In *Logic Programming: Formal Methods and Practical Applications*, herausgegeben von Beierle, C. und Plümer, L., Studies in Computer Science and Artificial Intelligence, Kap. 2, S. 20–90. North-Holland.

Börger, E. und Schulte, W. (1998a): A Modular Design for the Java VM architecture. In *Architecture Design and Validation Methods*, herausgegeben von Börger, E. Springer.

Börger, E. und Schulte, W. (1998b): Defining the Java Virtual Machine as Platform for Provably Correct Java Compilation. In *23rd International Symposium on Mathematical Foundations of Computer Science*, LNCS. Springer. To appear.

Börger, E. und Schulte, W. (1998c): Programmer Friendly Modular Definition of the Semantics of Java. In *Formal Syntax and Semantics of Java*, herausgegeben von Alves-Foss, J., LNCS. Springer.

Bracha, G., Odersky, M., Stoutamire, D. und Wadler, P. (1998): Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. *ACM SIGPLAN Notices*, 33(10): 183–200.

Brown, D. F., Moura, H. und Watt, D. A. (1992): Actress: an Action Semantics Directed Compiler Generator. In *Compiler Compilers 92*, Bd. 641 von *Lecture Notes in Computer Science*.

Bundesamt für Sicherheit in der Informationstechnik, B. (1994): BSI-Zertifizierung. BSI 7119, Bonn.

Chirica, L. und Martin, D. (1986): Toward Compiler Implementation Correctness Proofs. *ACM Transactions on Programming Languages and Systems*, 8(2): 185–214.

Diehl, S. (1996): *Semantics-Directed Generation of Compilers and Abstract Machines*. Dissertation, University of the Saarland, Germany.

Dijkstra, E. W. (1976): *A Discipline of Programming*. Prentice-Hall.

Dold, A., Gaul, T., Goerigk, W. et al. (1996a): Definition of the Language IS. Verifix Working Paper [Verifix/UKA/1], Uni Karlsruhe, CAU Kiel und Uni Ulm.

Dold, A., Gaul, T., Goerigk, W. et al. (1996b): Zwischenbericht *Verifix*. DFG-Zwischenbericht. Karlsruhe, Kiel, Ulm.

Dold, A. und Vialard, V. (1999): Formal Verification of a Compiler Back-End Generic Checker Program. In *Proceedings of PSI '99: Andrei Ershov Third International Conference on Perspectives Of System Informatics*, Bd. 1755 von *Lecture Notes in Computer Science*, Novosibirsk, Russia. Springer Verlag.

Donzeau-Gouge, V., Kahn, G. und ..., B. L. (1980): Formal definition of the ADA programming language. Techn. Ber., INRIA, Versailles, France.

Dörr, H. (1995): *Efficient Graph Rewriting and Its Implementation*. Nr. 922 in LNCS. Springer. Veröffentlichung der Dissertation.

Ehrich, H.-D., Gogolla, M. und Lipeck, U. (1989): *Algebraische Spezifikation abstrakter Datentypen*. B.G. Teubner.

Frick, A., Neumann, R. und Zimmermann, W. (1997): Eine Methode zur Konstruktion robuster Klassenhierarchien. *Informatik – Forschung und Entwicklung*, 12(4): 186–195.

Gaul, T. (1996): Bechmarking code-generation for IS to DEC-Alpha. Verifix Working Paper [Verifix/UKA/11], University of Karlsruhe.

Gaul, T., Heberle, A., Heuzeroth, D. und Zimmermann, W. (1995): An Evolving Algebra Specification of the Operational Semantics of MIS. Verifix Working Paper [Verifix/UKA/3], University of Karlsruhe.

Gaul, T., Heberle, A., Zimmermann, W. und Goerigk, W. (1999): Construction of Verified Software Systems with Program-Checking: An Application To Compiler Back-Ends. Techn. Ber., Trento, Italy.

GAUL, T. und ZIMMERMANN, W. (1995): An Evolving Algebra for the Alpha Processor Family. Verifix Working Paper [Verifix/UKA/4], University of Karlsruhe.

GLESNER, S. (1999): *Natürliche Semantik für imperative und objektorientierte Programmiersprachen.* Dissertation, Universität Karlsruhe.

GOERIGK, W., DOLD, A., GAUL, T. ET AL. (1996): Compiler Correctness and Implementation Verification: The Verifix Approach. In *Compiler Construction*, Bd. 1060 von *LNCS*. Springer. Poster Session, International Conference on Compiler Construction 1996.

GOERIGK, W., GAUL, T. und ZIMMERMANN, W. (1998a): Correct Programs without Proof? On Checker-Based Program Verification. In *Proceedings ATOOLS'98 Workshop on "Tool Support for System Specification, Development, and Verification"*, Advances in Computing Science, Malente. Springer Verlag.

GOERIGK, W., ZIMMERMANN, W., GAUL, T. ET AL. (1998b): Praktikable Konstruktion korrekter Übersetzer. In *Softwaretechnik '98*, Bd. 18 von *Softwaretechnik-Trends*, S. 26–33. GI.

GOOS, G. (1997): Sather-K – The Language. *Software – Concepts and Tools*, 18: 91–109.

GOOS, G. und ZIMMERMANN, W. (1999): Verification of Compilers. In *Correct Systems*, LNCS. Springer Verlag.

GOSLING, J., JOY, B. und STEELE, G. (1996): *The Java Language Specification.* The Java Series. Addison-Wesley.

GUREVICH, Y. (1995): Evolving Algebras: Lipari Guide. In *Specification and Validation Methods*, herausgegeben von Börger, E. Oxford University Press.

GUREVICH, Y. (1997): May 1997 Draft on the ASM Guide. Techn. Ber., EECS Dept., University of Michigan.

GUREVICH, Y. und HUGGINS, J. K. (1993): The Semantics of the C Programming Language. In *LNCS*, Bd. 702, S. 274–308. Springer-Verlag.

HANNAN, J. (1994): Operational Semantics-Directed Compilers and Machine Architectures. *ACM Transactions on Programming Languages and Systems*, 16(4): 1215–1247.

HEBERLE, A., GAUL, T., GOERIGK, W. ET AL. (1999): Construction of Verified Compiler Front-Ends with Program-Checking. In *Proceedings of PSI '99: Andrei Ershov Third International Conference on Perspectives Of System Informatics*, Lecture Notes in Computer Science, Novosibirsk, Russia. Springer Verlag. To appear.

HEBERLE, A. und HEUZEROTH, D. (1997): Algebraische Spezifikation eines generischen Integer-Datentyps. Verifix Working Paper [Verifix/UKA/14], IPD, Universität Karlsruhe.

HEBERLE, A. und LÖWE, W. (1998): On ASM-Based Specification of Programming Language Semantics and Reusable Correct Compilations. In *Proceedings of the 5th International Workshop on Abstract State Machines*, herausgegeben von Glässer, U. und Schmitt, P. H., S. 68–90.

Heberle, A., Löwe, W. und Trapp, M. (1998a): Safe Reuse of Source to Intermediate Language Compilations. In *Proceedings of the Ninth International Symposium on Software Reliability Engineering, Fast Abstracts and Industrial Tracks*, herausgegeben von Chillarege, R.

Heberle, A., Löwe, W. und Trapp, M. (1998b): Safe Reuse of Source to Intermediate Language Compilations – Elaborated Version. URL: http//i44www.info.uni-karlsruhe.de/ heberle/publications/issre-details.ps.gz.

Heuzeroth, D. (1998): *Spezifikation und Verifikation von standardisierten Transformationen am Beispiel der Übersetzung der imperativen Sprache IS.* Diplomarbeit, University of Karlsruhe.

Heuzeroth, D. und Heberle, A. (1998): The Formal Specification of IS. Verifix Working Paper [Verifix/UKA/2 revised], IPD, Universität Karlsruhe.

Hoare, C. (1969): An axiomatic basis for computer programming. *Communications of the ACM*, 12: 576–583.

Hoffmann, U. (1998): *Compiler Implementation Verification through Rigorous Syntactical Code Inspection.* Dissertation, Technische Fakultät der Christian-Albrechts-Universität zu Kiel, Kiel.

IEEE (1985): IEEE Standard for binary floating-point arithmetic. Techn. Ber., ANSI/IEEE. Std. 754–1985.

Kahn, G. (1987): Natural Semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, Bd. 247 von *LNCS*, S. 22–39. Springer-Verlag.

Kernighan, B. und Ritchie, D. (1978): *The C Programming Language.* Prentice-Hall, Englewoods Cliff, NY.

Kutter, P. und Pierantonio, A. (1997a): The Formal Specification of Oberon. *Springer Journal of Universal Computer Science*, 3(5): 443 – 503.

Kutter, P. und Pierantonio, A. (1997b): Montages: Specifications of Realistic Programming Languages. *J.UCS*, 3(5): 416 – 442.

Loeckx, J., Ehrich, H.-D. und Wolf, M. (1996): *Specification of Abstract Data Types.* John Wiley & Sons Ltd. and B.G. Teubner.

Merke, A. (1998): Generische ASM Transformationen. Studienarbeit, Universität Karlsruhe.

Meyer, B. (1992): Design by Contract. *IEEE Computer (Special Issue on Inheritance and Classification*, 25(10): 40–52.

Mosses, P. (1979): SIS — Semantics Implementation System, Reference Manual and User Guide. DAIMI Report MD-30, DAIMI, University of Århus, Denmark.

Mosses, P. D. (1976): Compiler Generation Using Denotational Semantics. In *Proceedings of the 5th Symposium on Mathematical Foundations of Computer Science*, Bd. 45 von *LNCS*, S. 436–441. Springer. ISBN 3-540-07854-1.

Mosses, P. D. (1992): *Action Semantics.* Cambridge University Press.

Necula, G. C. und Lee, P. (1998): The Design and Implementation of a Certifying Compiler. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, S. 333–344.

Nielson, H. und Nielson, F. (1992): *Semantics with Applications.* Wiley.

Odersky, M. und Wadler, P. (1997): Pizza into Java: Translating theory into practice. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, S. 146–159.

Owre, S., Rushby, J. und Shankar, N. (1992): PVS: A Prototype Verification System. In *Proceedings 11th International Conference on Automated Deduction CADE*, herausgegeben von Kapur, D., Bd. 607 von *Lecture Notes in Artificial Intelligence*, S. 748–752, Saratoga, NY. Springer-Verlag.

Paakki, J. (1995): Attribute Grammar Paradigms – A High-Level Methodology in Language Implementation. *ACM Computing Surveys*, 27(2): 196–255.

Palsberg, J. (1992): *Provably Correct Compiler Generation.* Dissertation, Department of Computer Science, University of Aarhus. xii+224 pages.

Pettersson, M. (1995): *Compiling Natural Semantics.* Dissertation, Linkoeping University.

Plotkin, G. D. (1981): Structural Operational Semantics. Techn. Ber., Aarhus University.

Pnueli, A., Shtrichman, O. und Siegel, M. (1998): Translation Validation for Synchronous Languages. *Lecture Notes in Computer Science*, 1443.

Poetzsch-Heffter, A. (1997): Prototyping Realistic Programming Languages Based on Formal Specifications. *Acta Informatica*, 34(10): 737–772.

Polak, W. (1981): Compiler Specification and Verification. In *Lecture Notes in Computer Science*, herausgegeben von G. Goos, J. H., Nr. 124 in LNCS. Springer Verlag.

Proebsting, T. A. (1995): BURS Automata Generation. *ACM Transactions on Programming Languages and Systems*, 17(3): 461–486.

Reif, W. (1999): Formale Methoden für sicherheitskritische Software – Der KIV Ansatz. *Informatik Forschung und Entwicklung*, 14(4): 193–202.

Schmidt, D. A. (1986): *Denotational Semantics.* Allyn and Bacon, Boston. ISBN 0-205-08974-7.

Schöning, U. (1995): *Theoretische Informatik kurz gefaßt.* BI-Wissenschaftsverlag, zweite Aufl.

Stoutamire, D. und Omohundro, S. (1996): The Sather 1.1 Specification. Techn. Ber. TR-96-012, International Computer Science Institute, Berkeley.

Strachey, C. (1966): Towards a Formal Semantics. In *Formal Language Description Languages for Computer Programming*, herausgegeben von Steel, T. B., S. 198–220. North-Holland.

STRACHEY, C. und WADSWORTH, C. (1974): Continuations: A Mathematical Semantics for Handling Full Jumps. Techn. Ber. Tech. Monograph PRG-11, Programmming Research Group, University of Oxford.

STROUSTRUP, B. (1986): *The C++ Programming Language*. Addison-Wesley.

TENNENT, R. D. (1978): *A denotational semantics of the programming language Pascal*. Programming Research Group, Oxford University.

TRAPP, M. (1999): *Optimierung objekt-orientierter Programme*. Dissertation, Universität Karlsruhe. In Vorbereitung.

TRAPP, M., BOESLER, B. und LINDENMAIER, G. (1999): Documentation of the Intermediate Representation FIRM. Techn. Ber. 1999-44, Universität Karlsruhe.

VAN LEEUWEN, J. (Hrsg.) (1990): *Algorithms and Complexity*, Bd. A von *Handbook of Theoretical Computer Science*. Elsevier.

WAITE, W. M. und GOOS, G. (1984): *Compiler Construction*. Springer Verlag, Berlin, New York Inc.

WALLACE, C. (1994): The Semantics of the C++ Programming Language. In *Specification and Validation Methods*, herausgegeben von Börger, E., S. 131–164. Oxford University Press.

WAND, M. (1984): A Semantic Prototyping System. *ACM SIGPLAN Notices*, 19(6): 213–221.

WIRSING, M. (1990): Algebraic Specification. In *Handbook of Theoretical Computer Science Vol. B*, herausgegeben von van Leeuwen, J., S. 675–788. MIT-Press.

ZENTRALSTELLE FÜR SICHERHEIT IN DER INFORMATIONSTECHNIK, Z. (1990): IT-Evaluationshandbuch. Bundesanzeiger Verlagsgesellschaft, Köln.

ZIMMERMANN, W. und GAUL, T. (1997): On the Construction of Correct Compiler Back-Ends: An ASM Approach. *Journal of Universal Computer Science*, 3(5): 504–567.

A Grundlagen

Wir führen Begriffe und Notationen ein, die für das Verständnis der Arbeit gebraucht werden. Im nächsten Abschnitt beschreiben wir Grundlagen algebraischer Spezifikation, machen dann in Abschnitt A.2 allgemeine Aussagen über unendliche Zustandsfolgen und führen schließlich in Abschnitt A.3 die Syntax und Semantik von Graphersetzungsregeln ein, die wir zur Beschreibung von Transformationen verwenden.

A.1 Algebraische Spezifikation

In diesem Abschnitt definieren wir die Grundbegriffe algebraischer Spezifikation und erklären die Konstruktion von Spezifikationen als Grundlage der Konstruktion von abstrakten Zustandsmaschinen. Die Definitionen sind zum großen Teil aus (Loeckx et al., 1996), (Ehrich et al., 1989) und (Wirsing, 1990) entnommen.

Definition A.1
*Eine **Signatur** Σ ist ein Paar $\langle S, F\rangle$ von Mengen, den Elementen S (Sorten) und den Operationen F. Eine Operation besteht aus einem $(k+2)$-Tupel $f : s_1 \times \ldots \times s_k \to s$ mit $s_1, \ldots, s_k, s \in S$ und $k \geq 0$; f ist der Name, k die Stelligkeit, $s_1, \ldots, s_k$ sind die Argumentsorten und s ist die Resultatssorte der Operation.*

Jede Signatur Σ definiert eine Menge syntaktisch korrekter Ausdrücke, die sich aus freien Variablen und den Funktionssymbolen der Signatur aufbauen lassen.

Definition A.2
*Sei X eine Menge von Variablen über S (disjunkt zu den nullstelligen Funktionen in F). Die Menge der **Terme** $T(\Sigma, X)$ ist induktiv definiert:*

1. $X \subseteq T(\Sigma, X)$
2. *wenn $f :\to s$ eine Operation in F ist, dann ist $f \in T(\Sigma, X)$*
3. *wenn $f : s_1 \times \ldots \times s_k \to s$, $k \geq 1$ eine Operation in F ist und $t_i \in T(\Sigma, X)$ für $1 \leq i \leq k$ gilt, dann ist $f(t_1, \ldots, t_k) \in T(\Sigma, X)$.*

Mit dieser Definition sind Variablen und Konstanten einer Sorte Terme und die Anwendung von Operationen auf Terme ebenfalls. Die Menge der **Grundterme** über Σ ist durch $T(\Sigma, \emptyset)$ definiert.

Mit Σ können wir die Syntax einer Spezifikation beschreiben, das sagt jedoch noch nichts über die Bedeutung aus. Durch Angabe von Axiomen können wir das Verhalten der Operationen eingrenzen und damit eine Klasse von Algebren definieren, die dieses Verhalten beschreiben. Axiome sind prädikatenlogische Gleichungen über $T(\Sigma, X)$.

Definition A.3
*Eine **Spezifikation** ist ein Tripel $Spec = (\Sigma, X, E)$ wobei Σ eine Signatur, X ein System von Variablen und E eine Menge von Axiomen sind. Oft lassen wir X weg, so daß $Spec = \langle \Sigma, E \rangle$.*

Modelle für Spezifikationen sind heterogene Algebren A mit der entsprechenden Signatur, die die Axiomenmenge erfüllen (geschrieben $A \models E$). Algebren definieren Interpretationen von Signaturen, die jeder Sorte eine Trägermenge und jedem Operator eine Operation auf den angegebenen Trägermengen zuordnen.

Definition A.4
*Sei $\Sigma = \langle S, F \rangle$ eine Signatur. Eine Σ-**Algebra** $A = \langle [\![S]\!], [\![F]\!] \rangle$ weist*

- *jeder Sorte $s \in S$ eine Menge $[\![s]\!]$ zu, die Trägermenge der Sorte s*
- *jeder Operation $f \in F, f : s_1 \times \ldots \times s_k \to s$, $k \geq 0$ eine totale Funktion $[\![f : s_1, \ldots, s_k \to s]\!] : [\![s_1]\!] \times \ldots, [\![s_k]\!] \to [\![s]\!]$ zu, die Interpretation der Funktion f.*

Wir interessieren uns für Funktionen zwischen Signaturen und zwischen Algebren bzw. Spezifikationen.

Definition A.5
*Ein **Signatur-Morphismus** $f : \Sigma_1 \to \Sigma_2$ ist ein Paar $f = \langle g, h \rangle$, wobei für g und h gilt: $g : S_1 \to S_2$ ist eine Abbildung der Sorten und $h : F_1 \to F_2$ ist eine Familie von Abbildungen $h = \{h_{\overline{s},s'} : F_{1;\overline{s},s} \to F_{2;g(\overline{s}),g(s)}\}_{\overline{s} \in S_1^*, s \in S_1}$*

Definition A.6
*Seien A und B Σ-Algebren. Ein **Algebra-Morphismus** $h : A \to B$ ist eine Familie von Abbildungen*

$$h = \{h_s : [\![s]\!]_A \to [\![s]\!]_B\}_{s \in S}$$

so daß für alle $s, s_1, \ldots, s_k \in S, f : s_1 \times \ldots \times s_k \to s \in F$ und alle $a_1 \in [\![s_1]\!]_A, \ldots, a_k \in [\![s_k]\!]_A$ gilt:

$$[\![f]\!]_A(h_s(a_1, \ldots, a_k)) = h_s([\![f]\!]_B(a_1, \ldots, a_k))$$

Ein Algebra-Morphismus heißt **injektiv** (**bijektiv**), wenn alle Komponenten $h_s, s \in S$ diese Eigenschaft haben. Einen bijektiven Algebra-Morphismus nennen wir Algebra-Isomorphismus oder einfach Isomorphismus.

Für die Konstruktion des Zustandsraums (Mengen von $\Sigma - Algebren$) der abstrakten Zustandsmaschinen, die wir zur Semantikbeschreibung verwenden, benötigen wir Operationen zum Aufbau von Spezifikationen.

Jede Spezifikation $Spec = \langle \Sigma, E \rangle$ beschreibt eine Menge $Alg(\Sigma)$ von Σ-Algebren, die die Axiome E erfüllen ($\forall A \in Alg(\Sigma) : A \models E$). Es gilt

$$\begin{aligned} Sig(\langle \Sigma, E \rangle) &\hat{=} \Sigma \text{ und} \\ Mod(\langle \Sigma, E \rangle) &\hat{=} \{A \in Alg(\Sigma) \mid A \models E\}, \end{aligned}$$

Die **Erweiterung**

$$Spec = extend(Spec_1, \Sigma, E)$$

einer Spezifikation $Spec_1 = \langle \Sigma_1, E_1 \rangle$ ist als

$$Spec = \langle Sigma_1 \cup Sigma, E_1 \cup E \rangle$$

definiert.

Operationen auf Spezifikationen basieren auf Signaturmorphismen. Jeder Signaturmorphismus $\sigma : \Sigma \to \Sigma'$ induziert eine Funktion $translate_\sigma$, die Elemente aus $Alg(\Sigma)$ in Elemente aus $Alg(\Sigma')$ überführt. Für $Spec = \langle \Sigma, E \rangle$ gilt:

$$\begin{aligned} Sig(translate_\sigma(\Sigma, E)) &\hat{=} \Sigma' \\ Mod(translate_\sigma(\Sigma, E)) &\hat{=} \{A \in Alg(\Sigma') \mid A|_\sigma \in Alg(\Sigma, E)\} \end{aligned}$$

Dabei ist $A|_\sigma$ folgendermaßen definiert.

Definition A.7
*Seien $\Sigma = \langle S, F \rangle$ und Σ' zwei Signaturen und $\sigma : \Sigma \to \Sigma'$ ein Signaturmorphismus. Sei A' eine Σ'-Algebra. Das σ-**Redukt** von A', die Σ-Algebra $A'|_\sigma$, ist definiert durch:*

$$\begin{aligned} \forall s \in S : (A'|_\sigma)(s) &\hat{=} A'(\sigma(s)) \\ \forall f \in F : (A'|_\sigma)(f) &\hat{=} A'(\sigma(f)) \end{aligned}$$

Informell beschreibt $A'|_\sigma$ eine Σ-Algebra mit der Semantik von A'. Gilt $\Sigma \subseteq \Sigma'$, dann ist $A'|_\sigma$ genau die Einschränkung der Σ'-Algebra auf die Trägermengen und Funktionen von Σ.

Umbenennungen in Spezifikationen $Spec = \langle \Sigma, E \rangle$ können wir durchführen, indem wir einen injektiven Signaturmorphismus $\sigma : \Sigma \to \Sigma'$ definieren und die Spezifikation damit übersetzen.

$$rename_\sigma(\Sigma, E) = translate_\sigma(\Sigma, E)$$

Zwei Spezifikationen $Spec_1, Spec_2$ mit gleicher Signatur Σ können durch die Operation $\cup$ vereinigt werden.

$$\begin{aligned} &Sig(Spec_1 \cup Spec_2) \hat{=} \Sigma, \\ &Mod(Spec_1 \cup Spec_2) \hat{=} Mod(Spec_1) \cap Mod(Spec_2) \end{aligned}$$

Das bedeutet die Axiome von $Spec_1$ und $Spec_2$ werden vereinigt.

$$(\Sigma, E_1) \cup (\Sigma, E_2) = (\Sigma, E_1 \cup E_2)$$

Die Vereinigung von Spezifikationen ist kommutativ, assoziativ und idempotent.

Spezifikationen mit unterschiedlichen Signaturen können kombiniert werden, indem wir ihre Übersetzungen in die Vereinigung der Signaturen vereinigen.

Definition A.8
*Sei $Spec_1 = \langle \Sigma_1, E_1 \rangle$ und $Spec_2 = \langle \Sigma_2, E_2 \rangle$ und sei $in_k : \Sigma_k \to \Sigma_1 \cup \Sigma_2$ die kanonische Einbettung definiert durch $in_k(x) = x$ für $x \in \Sigma_k, k = 1, 2$. Dann ist die **Summe** von $Spec_1$ und $Spec_2$ so definiert:*

$$Spec_1 + Spec_2 \hat{=} translate_{in_1}(Spec_1) \cup translate_{in_2}(Spec_2)$$

Bemerkung: Die Summe nimmt nicht die disjunkte Vereinigung von $Mod(Spec_1)$ und $Mod(Spec_2)$. Das hat den Vorteil, daß gemeinsame Teile von $Spec_1$ und $Spec_2$ nicht dupliziert werden; es hat den Nachteil, daß eine nicht gewünschte Spezifikation das Ergebnis ist, wenn gleiche Sorten oder Funktionssymbole in $Spec_1$ und $Spec_2$ unterschiedliche Bedeutung haben. ♢

A.1.1 Die Vereinigung von ASMs als Summe algebraischer Spezifikationen

Mit der oben eingeführten Summenbildung über algebraischen Spezifikationen können wir die Vereinigung von abstrakten Zustandsmaschinen formal definieren. Zu diesem Zweck müssen wir die Originaldefinition mit $\mathcal{A} = (\Sigma, Alg(\Sigma), \rightarrow, I)$ ein wenig anders interpretieren.

Wir nehmen an, daß die Initialzustände I der ASM $\mathcal{A}$ durch eine algebraische Spezifikation $\langle \Sigma, Init \rangle$ und die Zustandsübergangsrelation durch eine Menge R von ASM-Regeln definiert sind. Es gilt $I = Mod(\langle \Sigma, Init \rangle)$. Die ASM $\mathcal{A}$ ist dann durch $(\langle \Sigma, Init \rangle, R)$ spezifiziert.

Zwei ASMs $\mathcal{A}_1$ und $\mathcal{A}_2$ können vereinigt werden, $\mathcal{A}_1 \uplus \mathcal{A}_2$, indem die Summe über den Spezifikationen ihrer Initialzustände gebildet wird und ihre Zustandsübergangsrelationen vereinigt werden.

Definition A.9 (Vereinigung von ASMs)
Die Vereinigung $\mathcal{A}_1 \uplus \mathcal{A}_2$ von ASMs $\mathcal{A}_1 = (\langle \Sigma_1, Init_1 \rangle, R_1)$ und $\mathcal{A}_2 = (\langle \Sigma_2, Init_2 \rangle, R_2)$ ergibt die ASM $\mathcal{A} = (\langle \Sigma_1, Init_1 \rangle + \langle \Sigma_2, Init_2 \rangle, R_1 \cup R_2)$.

A.2 Aussagen über unendliche Zustandsfolgen

Sei U ein aufzählbares Universum, U^∞ die Menge aller endlichen oder unendlichen Folgen über U und $\delta : U^\infty \rightarrow U^\infty$ eine Funktion die benachbarte gleiche Elemente einer Folge löscht. Für $s, s' \in U^\infty$ gelte $s \sqsubseteq s'$ genau dann, wenn s ein endliches Präfix von s' ist. Dann gelten die folgenden Fakten.

Fakt A.1 $\sqsubseteq$ ist eine partielle Ordnung mit kleinstem Element $\langle\rangle$.

Fakt A.2 Sei $\langle s_i : i \in \mathrm{N} \rangle$ eine unendliche Folge über endlichen Folgen s_i aus U^∞, wobei für alle i gilt $s_i \sqsubseteq s_{i+1}$. Dann existiert eine eindeutige, kleinste, obere Schranke $s = \bigsqcup_{i=0}^{\infty} s_i$ mit $s_i \sqsubseteq s$ für alle i. Speziell haben alle Zustandsfolgen unserer abstrakten Maschinen diese Eigenschaft.

Eine Funktion $f : U^\infty \rightarrow V^\infty$ über den Universen U, V ist genau dann monoton, wenn $s \sqsubseteq s' \Rightarrow f(s) \sqsubseteq f(s')$ für $s, s' \in U^\infty$. Sie ist stetig, wenn zusätzlich $f\left(\bigsqcup_{i=0}^{\infty} s_i\right) = \bigsqcup_{i=0}^{\infty} f(s_i)$ für jede unendliche Sequenz gilt, die wie oben definiert ist.

Fakt A.3 Die punktweise Anwendung einer Funktion $f : U \rightarrow V$ auf eine Folge aus U^∞ definiert eine stetige Funktion $\widetilde{f} : U^\infty \rightarrow V^\infty$. Speziell definiert auch die punktweise Anwendung von $\overline{\rho}, \Psi_1, \Psi_2$, wie wir sie in Kapitel 6 verwenden, stetige Funktionen $\widetilde{\overline{\rho}}, \widetilde{\Psi}_1, \widetilde{\Psi}_2$ und auch δ ist stetig.

Fakt A.4 Sind zwei Funktionen $f_1 : U^\infty \rightarrow V^\infty$ und $f_2 : V^\infty \rightarrow W^\infty$ stetig, dann ist auch $f_2 \circ f_1$ stetig. Speziell ist auch $\widetilde{\overline{\rho}} \circ \widetilde{\Psi}_2$ stetig.

A.3 Graphersetzung

Wir benutzen einen Formalismus zur Beschreibung unserer Graph-/Termersetzungen, der von DÖRR (1995) eingeführt wurde. Dort werden Graphersetzungsregeln und deren Semantik formal definiert. Wir geben hier nur eine informelle Definition an, die zum Verständnis der beschriebenen Graphersetzungen ausreicht.

Wir betrachten Graphen mit benannten Kanten und Attributen, die Knoten zugeordnet sind.

Definition A.10 (Attributierter Graph)
Sei $G = (V, E, l)$ ein Graph mit Knotenmenge V, Kantenmenge E und Benennungen l. Sei A eine Menge von Attributen, D eine Menge von Attributwerten. Eine partielle Funktion $f : (V \times A) \to D$ definiert die Attributierung. Die Erweiterung des Graphen G um f definiert den ***attributierten Graphen***

$$ag = (V, E, l, f).$$

Definition A.11 (Graphersetzungsregel)
Seien $g_l = (V_l, E_l, l_l)$ und $g_r = (V_r, E_r, l_r)$ Graphen mit $l_l(v) = l_r(v)$ für alle $v \in V_l \cap V_r$ und M eine Einbettungsfunktion, welche die Einbettung von g_r an der Stelle von g_l definiert. Eine ***Graphersetzungsregel*** *r ist ein Tupel $r = (g_l, g_r, M)$. Der Graph g_l ist die linke Seite und g_r die rechte Seite der Regel.*

Ein Teilgraph wird mit folgenden graphischen Elementen beschrieben:

- eckige Kästen beschreiben ganze Graphen,
- Kreise bzw. Ovale beschreiben konkrete Knoten des Graphen,
- gestrichelte Knoten stellen Kontextknoten dar, die vom Kontext des Teilgraphen abstrahieren. Gestrichelte Knoten sind entweder benannt oder leer. Leere gestrichelte Knoten besagen, daß an dieser Stelle ein Teilgraph vorhanden sein kann, aber nicht muß. Kontextknoten werden bei einer Ersetzung nicht verändert.

Die Einbettungsfunktion M einer Graphersetzungsregel ist informell wie folgt definiert. Die Kontextknoten der linken Seite geben an, an welcher Stelle die Regel im Ausgangsgraphen angewendet werden kann. Derjenige Teilgraph, der mit der Regel „matcht“ und nicht zum Kontext gehört, wird ausgeschnitten und der Nicht-Kontextteil der rechten Seite der Regel wird eingesetzt. Eine formale Definition von Graphersetzungsregel und Einbettung mit Hilfe von Graphisomorphismen ist ebenfalls in (DÖRR, 1995) beschrieben.

Wir verwenden für unsere Spezifikationen attributierte Graphersetzungsregeln, die aus 3 Teilen bestehen:

1. Einer graphisch dargestellen Ersetzungsregel $r = (g_l, g_r, M)$
2. einer Bedingung die zusätzlich zum Passen des Graphmusters für die Anwendung der Regel erfüllt sein muß und
3. einer Transferregel, die neue Attribute einführt und Attributwerte neuer Knoten definiert, die durch die rechte Seite der Ersetzungsregel eingeführt werden.

Abbildung A.1 beschreibt eine Graphersetzungsregel die besagt, daß der Knoten mit der Nummer 3 durch den Graph bestehend aus den Knoten 7 und 8, die durch NT verbunden sind,

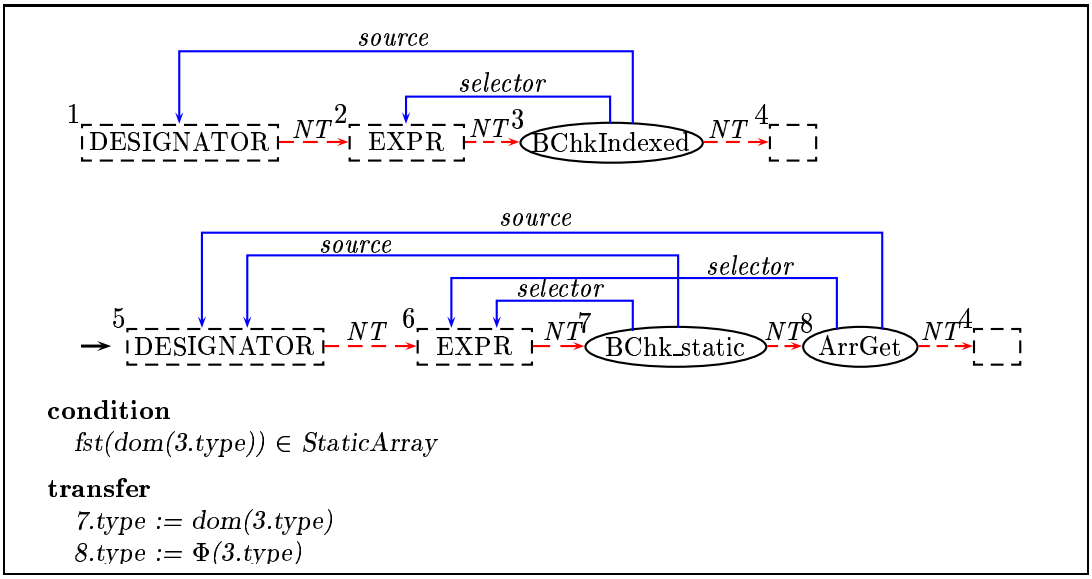


Abbildung A.1: Beispiel einer Graphersetzungsregel

ersetzt wird, sofern die Bedingung erfüllt ist. Ferner werden die Kanten *source* und *selector*, die von Knoten 3 ausgehen, gelöscht und stattdessen für die Knoten 7 und 8 definiert. Entsprechend der Semantik von Montages, zeigen eingehende durchgezogene Kanten (Datenflußkanten) auf den letzten Knoten eines Teilgraphen, z. B. bei 1, 2, 5 oder 6, und gestrichelte Kanten (Steuerflußkanten) gehen vom letzten Knoten (*terminal*) eines Teilgraphen aus. Eingehende Steuerflußkanten verweisen auf den ersten Knoten eines Graphen (*initial*). Die Kontextknoten geben an, an welcher Stelle die Regel im Ausgangsgraphen, also dem Graphen auf den sie angewendet wird, passen soll. Ist die Nummer eines Kontextknotens auf der rechten Seite der Ersetzungsregel eine andere als auf der linken, dann wird vorausgesetzt, daß der Knoten bzw. der evtl. dahinter steckende Graph bereits durch vorangegangene Graphersetzungen vollständig transformiert sein muß. Im Beispiel trifft dies auf den `DESIGNATOR`-Knoten zu, der auf der linken Seite die Nummer 1 und auf der rechten die Nummer 5 hat.

Für eine ausführliche Beschreibung zusammen mit einer formalen Definition von Graphersetzungsregel bzw. Graphersetzung sei auf Dörr (1995) verwiesen. Termersetzungen können mit diesem Formalismus natürlich auch dargestellt werden, da Terme Spezialfälle von Graphen sind.

B ASM Spezifikation der abstrakten Sprache *AL*

In diesem Abschnitt beschreiben wir die statische und dynamische Semantik der Sprache *AL*. Die dynamische Semantik ist durch eine abstrakte Zustandsmaschine (ASM_{AL}) definiert. Wir setzen voraus, daß die statische semantische Analyse schon abgeschlossen ist und jedem Knoten des Kommandographen ein statischer Typ zugeordnet wurde. Programme, die den statischen semantischen Bedingungen, die wir im folgenden zusammen mit der dynamischen Semantik definieren, nicht genügen, werden nicht betrachtet.

Zusätzlich nehmen wir an, daß die üblichen Operationen auf Listen definiert sind. Es gelten die bekannten Gesetze.

$$\begin{aligned}
[] &: \rightarrow \mathsf{T}^* \\
cons &: \mathsf{T} \times \mathsf{T}^* \rightarrow \mathsf{T}^* \\
append &: \mathsf{T}^* \times \mathsf{T}^* \rightarrow \mathsf{T}^* \\
head &: \mathsf{T}^* \rightarrow \mathsf{T} \\
tail &: \mathsf{T}^* \rightarrow \mathsf{T}^* \\
front &: \mathsf{T}^* \rightarrow \mathsf{T}^* \\
last &: \mathsf{T}^* \rightarrow \mathsf{T}
\end{aligned}$$

B.1 Sorten und Funktionen

AL-Programme sind Graphen. Die Ecken entsprechen Kommandos, die Kanten definieren Steuer- und Datenfluß. Kommandos werden in der ASM durch das Universum (die Sorte) Task repräsentiert, welches programmabhängig ist.

Der Speicher wird durch das Universum Reference und die Funktion

$$\mathit{content} : \mathsf{Reference} \rightarrow \mathsf{Value}$$

repräsentiert. *content* setzt Referenzen, die Objekte benennen, mit Werten in Beziehung. Der Speicher ist zweigeteilt in Objekte deren Lebenszeit auf die Lebenszeit einer Funktion beschränkt ist und Objekte, die dynamisch erzeugt werden und über deren Lebenszeit erst einmal nichts bekannt ist. Wir modellieren das durch eine Partitionierung von Reference.

$$\mathsf{HeapReference} \subseteq \mathsf{Reference}$$

AL definiert unterschiedliche Arten von Kommandos.

$$\begin{aligned}
\mathsf{Task} = {} & \mathsf{New} \cup \mathsf{Store} \cup \mathsf{LoadBasic} \cup \mathsf{LoadStruct} \cup \mathsf{NewGlobalFrame} \cup \mathsf{NewLocalFrame} \cup \\
& \mathsf{NewParamFrame} \cup \mathsf{BasicOps} \cup \mathsf{One} \cup \mathsf{All} \cup \mathsf{Call} \cup \mathsf{Return} \cup \mathsf{Read} \cup \mathsf{Write} \cup \mathsf{Alloc}
\end{aligned}$$

Jedes Kommando hat einen statischen Typ.

$$\mathit{type} : \mathsf{Task} \rightarrow \mathsf{Type}$$

Jedes Kommando berechnet einen Wert, der auch undefiniert sein kann. Die dynamische Funktion *value* repräsentiert den einem Kommando zugeordneten Wert.

$$value : \mathsf{Task} \times \mathrm{N} \rightarrow \mathsf{Value}$$

Der zweite Parameter der Funktion beschreibt die aktuelle Rekursionstiefe ($reclevel : \mathrm{N}$). Dieser Parameter ist notwendig, da Kommandos im Kommandographen bei rekursiven Funktions- oder Prozeduraufrufen, abhängig von der Aufruftiefe, unterschiedliche Werte haben können. Das Universum Value enthält die Werte der Basisdatentypen *Int*, *Float*, *Bool* und *String*. *Int* bzw. *Float* sind generische Datentypen, die durch die Parameter *MinInt* und *MaxInt* bzw. *Bit* konkretisiert werden. Es gilt $\mathsf{Bool} = \{true,\ false\}$. Außerdem enthält Value auch Referenzen. $\perp$ steht für den undefinierten Wert und ist verschieden von *undef*.

$$\mathsf{Value} = \mathsf{Int} \cup \mathsf{Float} \cup \mathsf{Bool} \cup \mathsf{String} \cup \mathsf{Reference} \cup \{\perp\}$$

Steuerfluß wird in der Semantik durch einen abstrakten Programmzähler

$$ct : \mathsf{Task}$$

und eine Funktion

$$nexttask : \mathsf{Task} \times \mathrm{N} \rightarrow \mathsf{Task}$$

modelliert. Das Weiterschalten des Befehlszählers drücken wir durch das Makro *proceed* aus, das in Abschnitt B.4 im Detail beschrieben wird.

Zum Laufzeitsystem von *AL* gehören einige ausgezeichnete, global bekannte Referenzen.

$$\begin{aligned} params &: \mathrm{N} \rightarrow \mathsf{Reference} \\ locals &: \mathrm{N} \rightarrow \mathsf{Reference} \\ globals &: \rightarrow \mathsf{Reference} \end{aligned}$$

params verwaltet die aktuellen Schachteln von Parametern, *locals* und *globals* bezeichnen lokale bzw. globale Funktionsschachteln. *locals* und *params* referenzieren abhängig von der Aufruftiefe unterschiedliche Schachteln.

Datenabhängigkeiten sind Kanten im Kommandographen und werden durch die Funktion

$$data : Task \times \mathrm{N} \rightarrow Task$$

modelliert. Zusätzlich sind für die einzelnen Kommandos statische Funktionen definiert, die wir beschreiben, wenn sie verwendet werden.

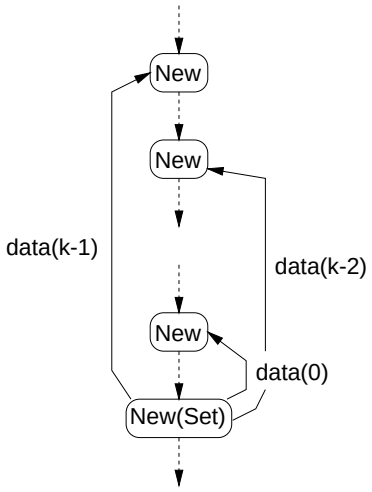
B.2 Basisoperationen

New, LoadBasic, LoadStruct und Store sind Kommandos zum Erzeugen eines Objekts (genauer einer Referenz), zum Lesen und zum Setzen eines Wertes. BasicOps repräsentiert die Standardoperationen der Basisdatentypen.

Mit dem Kommando New kann ein neues Basisobjekt oder ein Kopfobjekt einer Struktur erzeugt werden bzw. es wird eine neue Referenz erzeugt, über die auf den Inhalt eines Objekts

zugegriffen werden kann. Speicherobjekte haben einen Typ und eine Referenz. Strukturierte Daten können nur über diese Referenz und einen Selektor zugegriffen werden. Dementsprechend wird bei der Erzeugung eines neuen strukturierten Objekts eine Referenz erzeugt, mit der auf die Unterteile zugegriffen werden kann. Ein Selektor kann eine Konstante oder das Ergebnis einer Berechnung sein, falls es sich um einen Index handelt. Das New-Kommando für einen strukturierten Typ, wird erst nach der Abarbeitung der New-Kommandos für die Elemente abgearbeitet. Die Elemente werden über die Selektoren zugegriffen, die durch die Funktion *data* beschrieben sind. Im folgenden beschreiben wir die Regeln zur Interpretation eines Kommandos und zeigen zusätzlich den zugrundeliegenden Kommandographen.

```
if New(ct) then
   if BasicType(type(ct))
      extend Reference with r
         type(r) := type(ct)
         content(r) := ⊥
         value(ct, reclevel) := r
      endextend
   else
      extend Reference with r
         type(r) := type(ct)
         value(ct,reclevel) := r
         do forall t : (t ∈ components(type(ct)))
            r.(snd(t)) := value(ct.(snd(t)), reclevel)
         enddo
      endextend
   endif
   proceed
endif
```



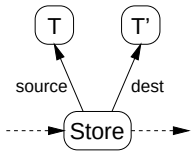
components projeziert aus einem Set-Typ die Menge der Komponenten heraus.

$components : \mathsf{Set} \rightarrow \mathsf{Pair}^{+}$

Statische Semantik von New: Für ein New-Kommando, das ein strukturiertes Objekt erzeugt stehen die New-Kommandos zur Erzeugung der Komponenten im Steuerfluß vor diesem Kommando. Das stellt sicher, daß bei der Erzeugung eines strukturierten Objekts alle Teilobjekte schon existieren. Der Typ einer Struktur definiert entweder Namen für alle Komponenten der Struktur oder Indizes. Indizes beginnen mit 0 und sind ohne Lücken definiert.

Mit dem Kommando Store ändert man den Wert eines Speicherobjekts, genauer man ändert die Interpretation der Funktion *content*. Store benutzt zwei Daten, *source* ist der neue Wert und *dest* ist eine Referenz auf das Objekt, dessen Wert verändert wird.

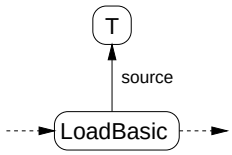
```
if Store(ct) then
  content(value(ct.dest,reclevel)) := value(ct.source,reclevel)
  proceed
endif
```



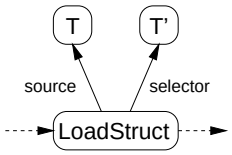
Statische Semantik von Store: Der Typ des Ziels (*dest*) einer Speicheroperation muß gleich dem Typ der Quelle (*source*) der Speicheroperation sein.

Ein LoadBasic-Kommando liest den Wert eines Speicherobjekts. Der Zugriff auf eine Struktur geschieht mit einem LoadStruct-Kommando über das Kopfelement und einen Selektor.

```
if LoadBasic(ct) then
  value(ct,reclevel) := content(value(ct.source,reclevel))
  proceed
endif
```




```
if LoadStruct(ct) then
  value(ct,reclevel) :=
       value(ct.source,reclevel).(value(ct.selector,reclevel))
  proceed
endif
```



Statische Semantik von LoadBasic **und** LoadStruct**:** Der statische Typ der Zugriffsoperation auf einen Basistyp (LoadBasic) ist gleich dem statischen Typ der Quelle (*source*). Der Typ der Zugriffsoperation auf das Element einer Struktur (LoadStruct) ist der Typ der selektierten Komponente.

Die Basisoperationen sind alle nach dem gleichen Prinzip aufgebaut. Sie benutzen die Ergebnisse anderer Kommandos zur Berechnung des eigenen Wertes. Bei der Berechnung können unter Umständen Über- oder Unterläufe auftreten. Die genaue Semantik der Operationen hängt von der Definitionen der Datentypen ab.

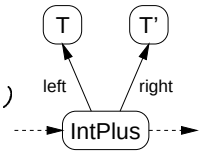
$$unop : \mathsf{Expr} \rightarrow \mathsf{Task}$$
$$binop : \mathsf{Expr} \times \mathsf{Expr} \rightarrow \mathsf{Task}$$

Wir zeigen hier nur das Beispiel des zweistelligen Operators IntPlus. Der Datentyp *Int* ist generisch definiert. Abhängig von der Arithmetik wird entweder in einen expliziten Fehlerzustand verzweigt oder der Fehler wird nur angezeigt, indem die dynamische Funktion *exception* gesetzt wird. Die Integeroperationen kennen die folgenden Ausnahmen.

$$exception \hat{=} \{\mathsf{IntOverFlow}, \mathsf{IntUnderflow}\}$$

Wir zeigen nur die Semantik für die Arithmetik mit Überlauf. Die dynamische Semantik der Plusoperation in Ringarithmetik ist analog.

```
if IntPlus(ct) then
  if fromint(value(ct.left,reclevel))+fromint(value(ct.right,reclevel))
          > fromint(MaxInt) then
    exception := IntOverflow
    ct := OverflowError
  elsif fromint(value(ct.left,reclevel))+fromint(value(ct.right,reclevel))
          < fromint(MinInt) then
    exception := IntUnderflow
    ct := UnderflowError
  else
    value(ct,reclevel) := value(ct.left,reclevel)+_I value(ct.right,reclevel)
    proceed
  endif
endif
```



Um einen Über- bzw. Unterlauf festzustellen, rechnen wir im Ring der ganzen Zahlen und benutzen zur Umwandlung die Funktion.

$$fromint : \mathsf{Int} \rightarrow \mathrm{Z}.$$

Die Regel für IntMult ist analog zu der Regel für IntPlus. Für IntMinus wird anstatt einem Überlauf auf Unterlauf geprüft.

```
if IntDiv(ct) then
   if value(ct.right,reclevel) = 0_I then
      ct := DivByZeroError
   else
      value(ct,reclevel) :=
                  value(ct.left,reclevel) /_I value(ct.right,reclevel)
      proceed
   endif
endif
```

Die Semantik des generischen Integer-Datentyps definiert das Resultat von a/b mathematisch als $\lfloor a/b \rfloor$, man hätte stattdessen auch $\lceil a/b \rceil$ als Resultat festlegen können.

Für IntMod ist die Übergangsregel analog zu obiger Regel. Die Transitionen für Floatoperationen sind ebenfalls entsprechend.

Statische Semantik von *Int*-Operationen: Die Argumente einer *Int*-Operationen sind vom Typ *int*.

Die statische Semantik der Operationen über anderen Datentypen ist analog. Es gibt keine automatischen Typanpassungen in *AL*.

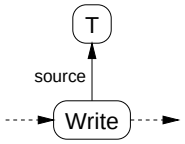
B.3 Ein- und Ausgabe

In unserem Modell findet die Kommunikation mit der Umgebung über Ein- und Ausgabeströme statt. Diese Ströme sind potentiell unendlich und werden durch die dynamischen Funktionen *input* und *output* modelliert.

$input :\rightarrow \mathsf{Char}^*$
$output :\rightarrow \mathsf{Char}^*$

Zur Änderung des Ein- bzw. Ausgabestroms sind in *AL* die Kommandos Read und Write vorhanden. Read entfernt das erste Element aus dem Eingabestrom und übernimmt den gelesenen Wert als Wert des Kommandos. Write fügt einen Wert am Ende des Ausgabestroms an.

```
if Read(ct) then
  value(ct,reclevel) := head(input)
  input := tail(input)
  proceed
endif
if Write(ct) then
  output := append(output, [value(ct.source,reclevel)])
  proceed
endif
```



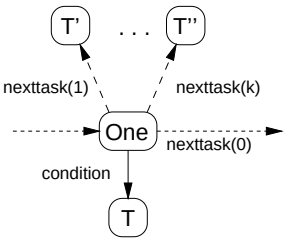
Statische Semantik von Read **und** Write:
Der statische Typ der Read- und der Write-Anweisung ist *String[1]*.

B.4 Steuerfluß

Das Kommando One beschreibt die Ausführung einer Alternative entsprechend dem Wert einer Bedingung. Die Menge der Möglichkeiten ist durch den Bereich von *nexttask* definiert. Ausgewählt wird das Element, dessen Markierung dem Wert des Schlüssels entspricht. Gibt

es kein Element mit entsprechender Markierung, dann passiert bei der Abarbeitung von One nichts und es wird an den Nachfolger des One-Kommandos weitergeschaltet. Existieren mehrere Alternativen mit demselben Schlüssel, dann wird indeterministisch eine der möglichen Alternativen ausgewählt. Das One-Kommando entspricht somit der CASE-Anweisung imperativer Sprachen.

```
if One(ct) then
 if ( ∃v: inrange(ct,v) ∧
         value(ct.condition,reclevel) = ct.nexttask(v).key ) then
  choose v: inrange(ct,v) ∧
            value(ct.condition,reclevel) = ct.nexttask(v).key )
   ct := ct.nexttask(v)
  endchoose
 else
  proceed
 endif
endif
```

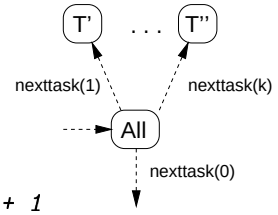


Statische Semantik von One: Der Typ der Bedingung eines One-Kommandos ist ein Basistyp und der Schlüssel jeder Alternative hat den gleichen Typ. Der Schlüssel einer Alternative ist außerdem eine Konstante.

inrange : One × N → Bool ist ein Prädikat, welches den Bereich beschreibt, in dem die Alternativen des One-Kommandos bzgl. der *nexttask*-Funktion liegen.

All ist ein Operator zur Ausführung einer Menge von Aktionen in beliebiger Reihenfolge. Aktionen können wiederum aus einer Menge von Teilaktionen bestehen. Bei der Ausführung von All wird jede Aktion komplett ausgeführt. Das heißt, all ihre Teilaktionen werden ausgeführt, bevor All in die nächste Aktion verzweigt. Zum Beispiel würden bei der Ausdrucksauswertung Teilausdrücke immer komplett und *nicht* verzahnt ausgeführt.

```
if All(ct) then
  if numofexecuted(ct,reclevel) = numofall(ct) then
    do forall v: executed(ct, ct.nexttask(v), reclevel)
      executed(ct,ct.nexttask(v)) := false
    enddo
    numofexecuted(ct) := 0
    proceed
  else
    choose v: parts(ct,v) ∧ ¬executed(ct,ct.nexttask(v))
      executed(ct,ct.nexttask(v),reclevel) := true
      numofexecuted(ct,reclevel) := numofexecuted(ct,reclevel) + 1
      ct := ct.nexttask(v)
    endchoose
  endif
endif
```



numofall : All → N gibt die Anzahl der Verzweigungen des All-Kommandos an, *numofexecuted* : All × N → N speichert die Anzahl der bereits ausgeführten Kommandos, *executed* : All × Task × N → Bool markiert ein Kommando als ausgeführt. und *parts* : All × N → Bool ist ein Prädikat, welches den Bereich beschreibt, in dem die Verzweigungen des All-Kommandos liegen. Auf den ersten Blick erscheint die Definition von All etwas kompliziert. Wir haben jedoch absichtlich diese Definition gewählt, da sie die Eigenschaften einer SASM (siehe Kapitel 6) erfüllt. Initial gilt für ein $t \in$ All:

$$numofexecuted(t, 0) = 0 \wedge \forall i : inrange(t, i) \Rightarrow executed(t, t.nexttask(i), 0) = false$$

Zusätzlich ist die Funktion

$$\mathit{final} : \mathsf{All} \times \mathrm{N} \rightarrow 2^{\mathsf{Task}}$$

definiert, die die letzten Kommandos der Unteraufgaben eines All-Kommandos beschreibt. Diese Information benötigen wir für die Übersetzung von All-Kommandos.

Statische Semantik von All: Für alle Kommandos t, die in der Menge der finalen Kommandos der Unteraufgaben eines All-Kommandos liegen, ist das nächste auszuführende Kommando wieder dieses All-Kommando.

Bemerkung: Die strukturelle Information, die durch *final* beschrieben ist, könnte zwar auch durch Analyse des *AL*-Programms berechnet werden. Allerdings ist diese Information auf der Ebene von Quellsprachen wesentlich einfacher zu berechnen bzw. schon durch die Struktur des Programms beschrieben. ♢

AL kennt Funktionen mit Parametern. Die Funktion *firsttask* bestimmt zu einem Funktionsnamen die Wurzel des Kommandographen, der die dynamische Semantik der Funktion beschreibt. Bei einem Funktionsaufruf wird der Aufrufer der Funktion gespeichert, damit nach Abarbeitung der Funktion an der entsprechenden Stelle aufgesetzt werden kann. Die Verwaltung der Funktionen geschieht über einen Kellermechanismus, der mit Hilfe des Attributs *reclevel* kodiert ist. Bei jedem Aufruf wird *reclevel* erhöht, bei Rücksprung aus einer Funktion wird *reclevel* wieder erniedrigt.

```
if Call(ct) then
    ct := firsttask(ct.id)
    caller(reclevel + 1) := ct
    reclevel := reclevel + 1
endif
```

Statische Semantik von Call: Das Attribut *id* enthält den Namen der gerufenen Funktion und $\mathit{firsttask}(\mathit{id})$ ist definiert.

AL hat ausgezeichnete Kommandos, die zur Übergabe von Parametern und zur Erzeugung lokaler und globaler Variablen benutzt werden. Ein NewParamFrame-Kommando erzeugt ein neues strukturiertes Objekt für die Parameter der nächsten gerufenen Funktion und macht dieses Objekt auch auf der nächsthöheren Aufruftiefe bekannt. Auf einer realen Maschine entspricht das dem Pushen der Parameterschachtel auf den Laufzeitkeller. Der *AL*-Graph in Zusammenhang mit NewParamFrame sieht aus, wie die Erzeugung eines strukturierten Objekts. Das verwundert nicht, da eine Schachtel im Speicher nichts anderes ist, als eine Liste von Speicherobjekten.

```
if NewParamFrame(ct) then
  extend Reference with r
    type(r) := type(ct)
    value(ct,reclevel) := r
    do forall t : (t ∈ components(type(ct)))
      r.(snd(t)) := value(ct.(snd(t)),reclevel)
    enddo
    params(reclevel+1) := r
  endextend
  proceed
endif
```

Eine Schachtel ist ein strukturiertes Objekt, das durch eine Referenz zugegriffen wird.

Statische Semantik von NewParamFrame**:** Der Typ des NewParamFrame-Kommandos ist die Vereinigung der Argumenttypen und des Resultattyps des nächsten Funktionsaufrufs im Steuerfluß. Zwischen dem NewParamFrame-Kommando und diesem Funktionsaufruf stehen nur Zuweisungen an die Argumente der Funktion. Der Returnwert einer Funktion wird in der Parameterschachtel gespeichert. Er wird unter dem ausgezeichneten Namen „*res*“ abgelegt. Das Auslesen des Resultatswerts geschieht direkt nach dem Funktionsaufruf. Spätere Zugriffe auf die Parameterschachtel sind illegal.

Das NewLocalFrame-Kommando erzeugt eine Schachtel für die lokalen Variablen.

```
if NewLocalFrame(ct) then
  extend Reference with r
    type(r) := type(ct)
    value(ct,reclevel) := r
    do forall t : (t ∈ components(type(ct)))
      r.(snd(t)) := value(ct.(snd(t)),reclevel)
    enddo
    locals(reclevel) := r
  endextend
  proceed
endif
```

Statische Semantik von NewLocalFrame**:**

Vor dem NewLocalFrame-Kommando einer Funktion dürfen nur New-Kommandos ausgeführt werden.

Ein NewGlobalFrame-Kommando erzeugt eine Schachtel für die globalen Variablen.

```
if NewGlobalFrame(ct) then
  extend Reference with r
    type(r) :=c (t)
    value(ct,reclevel) := r
    do forall t : (t ∈ components(type(ct)))
      r.(snd(t)) := value(ct.(snd(t)),reclevel)
    enddo
    globals := r
  endextend
  proceed
endif
```

Statische Semantik von NewGlobalFrame**:** Vor dem NewGlobalFrame-Kommando der Funktion *main* dürfen nur New-Kommandos ausgeführt werden.

Beim Rücksprung aus einer Prozedur wird das auf den dynamischen Vorgänger folgende Kommando ausgeführt und die Aufruftiefe verringert. Außerdem werden die lokalen Variablen und die Parameter wieder vom Laufzeitkeller genommen.

```
if Return(ct) then
  ct := caller(reclevel).nexttask(0)
  reclevel := reclevel - 1
  params(reclevel) := undef
  locals(reclevel) := undef
endif
```

Returnwerte werden in der aktuellen Parameterschachtel mitverwaltet. Die aktuelle Schachtel wird zwar beim Rücksprung aus einer Funktion vom Parameterkeller genommen, aber das

Resultat kann auf der nächsttieferen Aufrufebene trotzdem noch zugegriffen werden, da das Parameterobjekt auf dieser Ebene erzeugt wurde.

Das Makro *proceed* benutzt die Funktion *nexttask* mit der Steuerfluß definiert wird. Im einfachen Fall gibt es genau einen Nachfolger eines Kommandos. Damit gilt:

$$proceed == ct := ct.nexttask(0)$$

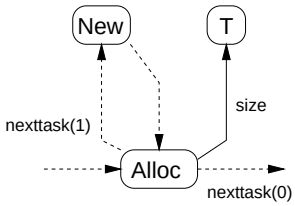
Prinzipiell erlaubt *AL* auch, daß Steuerfluß implizit durch Datenflußabhängigkeiten definiert wird. Das bedeutet für ein *AL*-Programm, daß Steuerflußkonstrukte und -kanten erst noch berechnet und eingefügt werden müssen. Die Strategie zur „sparsamen" Definition von Steuerfluß läßt mehr Freiheiten für Optimierungen. *nexttask* definiert dann eine Menge von möglichen Nachfolgern, auf denen durch Datenflußabhängigkeiten eine Ordnung $\prec^+$ definiert ist. *proceed* wählt dann einen, mit der Ordnung $\prec$ verträglichen, Nachfolger aus dieser Menge aus. Wir könnten das folgendermaßen modellieren:

```
proceed ==
  choose v: inrange(ct,v)
  satisfying ∀ t: inrange(ct,t) ∧ ct.nexttask(t) ≺ ct.nexttask(v)
                    ⇒ executed(ct,ct.nexttask(t))
     ct := ct.nexttask(v)
     executed(ct.nexttask(v)) := true
  endchoose
```

B.5 Halde

Die Sprache *AL* sieht eine Halde zur Verwaltung und Speicherung von Objekten vor, über deren Lebenszeit keine Aussagen gemacht wird. Auf dieser Halde werden auch Objekte verwaltet, deren Größe erst zur Laufzeit berechnet wird. Die Objekte werden durch eine Struktur von New-Kommandos erzeugt. Das Kommando Alloc erzeugt aus einem Elementtyp und einer Größe *size* ein Haldenobjekt mit *size* Elementen des entsprechenden Typs.

```
if Alloc(ct) then
  if numofallocated(ct) = undef then
    extend HeapReference with r
      type(r) := type(ct)
      value(ct,reclevel) := r
      content(r) := ⊥
    endextend
    numofallocated(ct) := 0
    ct := ct.first
  elsif numofallocated(ct) < ct.size then
    value(r.data(numofallocated(ct)),reclevel):= value(ct.elements,reclevel)
    numofallocated(ct) := numofallocated(ct) + 1
    ct := elements
  else
    numofallocated(ct) := undef
    proceed
  endif
endif
```



Im Prinzip realisiert Alloc eine Schleife, die *size* mal einen New-Graphen durchläuft und entsprechend Objekte erzeugt. Das Kopfobjekt des strukturierten Objekts wird erst nach der Erzeugung seiner Teilobjekte angelegt. In diesem Fall zeigt *first* auf das erste Kommando, das

ein Basisobjekt des strukturierten Haldenelements anlegt. Ist das Haldenelement selbst ein Basisobjekt, dann fallen *first* und *element* zusammen.

Statische Semantik von Alloc**:** Alle Elemente, die von Alloc angelegt werden sind Halden-Elemente. Die Argumente eines Alloc-Kommandos enthalten keine weiteren Alloc-Kommandos.

B.6 Initialzustand

Das Kommandouniversum bzw. die Unteruniversen beschreiben das *AL*-Programm. Der Kommandograph wird durch das Programm π vorgegeben. Damit ist die Initialisierung von *next-task*, *data* und *firsttask* festgelegt. Ansonsten gilt:

$$
\begin{aligned}
\mathsf{Reference} &= \emptyset \\
ct &= \mathit{firsttask}('main') \\
\mathit{input} &= [v_1, v_2, \ldots] \\
\mathit{output} &= [] \\
\mathit{reclevel} &= 0 \\
\mathit{globals} &= \bot \\
\forall \mathit{reclevel} \in \mathrm{N} : (\ & \mathit{locals}(\mathit{reclevel}) = \bot \wedge \mathit{params}(\mathit{reclevel}) = \bot\ )
\end{aligned}
$$

Wir sehen das Hauptprogramm als Funktion mit dem ausgezeichneten Namen „*main*“ an. Die Initialisierung von *input* ist eine potentiell unendliche Liste von Werten.

Lebenslauf

Persönliche Daten

Name:	Andreas Heberle
Geburtsdatum:	8. Juni 1965
Geburtsort:	Tübingen
Staatsangehörigkeit:	deutsch
Adresse:	Ludwig-Wilhelm-Str. 1 76131 Karlsruhe E-mail: heberle@ipd.info.uni-karlsruhe.de

Ausbildung

1971 - 1975:	St. Klara Grundschule Rottenburg
1975 - 1984:	Eugen-Bolz-Gymnasium Rottenburg
29. Mai 1984:	Abitur
7/84 - 9/85:	Wehrdienst
10/85 - 8/92:	Studium der Informatik an der Universität Karlsruhe
31. August 1992:	Diplom
seit 9/93:	Wissenschaftlicher Mitarbeiter am Institut für Programmstrukturen und Datenorganisation von Prof. Goos