Transformation verteilter Systeme:
Von applikativen zu prozeduralen Darstellungen

Frank Dederichs

Zusammenfassung

Die Entwicklung verteilter Systeme ist eine schwierige Aufgabe. Sie kann nur durch systematisches Vorgehen innerhalb eines formalen Rahmens angemessen bewältigt werden.

Die vorliegende Arbeit verfolgt das Ziel, einen solchen Rahmen zu erweitern und auszugestalten. Sie leistet einen Beitrag zu FOCUS, einer formalen Entwicklungsmethode für verteilte Systeme. FOCUS gliedert den Entwicklungprozeß in vier Phasen, in denen zuerst die Anforderungen an das System (1. Phase: Anforderungsspezifikation), dann seine grobe Struktur (2. Phase: Designspezifikation) und schließlich ein abstraktes (3. Phase: Abstrakte Implementierung), sowie ein konkretes Programm (4. Phase: Konkrete Implementierung) bestimmt werden. In allen Phasen kommen abgestimmte Formalismen zum Einsatz. Methodische Leitlinien unterstützen ihren Einsatz und ermöglichen durchgängige Systementwicklungen. In dieser Arbeit werden die Ausdrucksmittel für die dritte und vierte Entwicklungsphase definiert und formal und methodisch eingebettet.

Für abstrakte Programme wird die applikative Sprache AL zur Verfügung gestellt. Mit ihr können verteilte Systeme auf relativ hohem Abstraktionsniveau in einer mathematisch-logischen Form beschrieben werden. Die Sprache besitzt eine denotationelle Semantik, die auf Strömen und stromverarbeitenden Funktionen beruht. Die Semantik gewährleistet Kompositionalität, trotz des in AL zulässigen Nichtdeterminismuses.

Zur Darstellung konkreter Programme dient die prozedurale Sprache PL. Sie enthält die üblichen imperativen Elemente, ist aber erweitert um Konzepte zur Parallelverarbeitung und (asynchronen) Kommunikation über gerichtete Kanäle. Auch für PL wird eine denotationelle Semantik angegeben, die sich ganz bewußt auf die gleichen Konzepte abstützt wie AL.

Der methodische Übergang von abstrakten zu konkreten Programmen, also von AL nach PL, erfolgt durch Techniken der Programmtransformation. Die Arbeit enthält eine Reihe von Transformationsregeln, die zum einen AL-Programme zueinander in Beziehung setzen und zum anderen applikative Programme in prozedurale Form überführen. Es wird ein Korrektheitsbegiff für Transformationsregeln bestimmt, bezüglich dessen die angegebenen Regeln auf der Grundlage der Sprachsemantiken korrekt bewiesen werden. Unterschiedliche Transformations-strategien werden in Bezug auf ihre methodischen Konsequenzen analysiert.

Viele Kolleginnen und Kollegen haben mich beim Schreiben dieser Arbeit unterstützt. Ihnen allen bin ich zu Dank verpflichtet.
Mein besonderer Dank gilt Herrn Prof. Dr. Manfred Broy für seine fachliche und menschliche Unterstützung und Herrn Prof. Dr. Wilfried Brauer für seine wertvollen Hinweise und Ratschläge.

Inhalt

1. Einleitung 1

1.1 Methodische Entwicklung verteilter Systeme 1

1.2 Programmtransformation 3

1.3 Ziele, Methoden und Aufbau der Arbeit 4

1.4 Vergleich mit anderen Ansätzen 7

2. Bereichstheoretische Grundlagen, Ströme 12

3. Die applikative Sprache AL 21

3.1 Syntax 21

3.2 Denotationelle Semantik 28

3.2.1 Breitensemantik von Ausdrücken 31

3.2.2 Funktionssemantik von Ausdrücken 37

3.2.3 Semantik von Gleichungssystemen 39

3.2.4 Semantik von Funktionen 40

3.2.5 Semantik von Agenten 42

3.2.6 Semantik von Programmen 44

3.3 Kompositionalitätsresultate 45

3.4 Nichtdeterminismus 50

3.5 AL-Programme und Agentennetze 56

4. Die prozedurale Sprache PL 61

4.1 Syntax 61

4.2 Denotationelle Semantik 67

4.2.1 Semantik von Ausdrücken 68

4.2.2 Semantik von Anweisungen 69

4.2.3 Semantik von Funktionen, Agenten und Programmen 75

4.3 AL und PL: Gemeinsamkeiten und Unterschiede 76

5. Transformationelle Programmentwicklung 86

5.1 Grundlagen 86

5.2 Transformation von AL-Programmen 90

5.3 Übergang von AL nach PL 105

5.3.1 Transformation stromrepetitiver Agenten 106

5.3.2 Transformation von Agentennetzen 122
5.3.3 Behandlung nicht-repetitiver Rekursion 132

6. Ausblick 139

Quellenverzeichnis 141

1. Einleitung

1.1 Methodische Entwicklung verteilter Systeme

Die Entwicklung verteilter (Software-)Systeme ist eine schwierige Aufgabe – auch und gerade im Vergleich zur sequentiellen Programmierung. Verteilte Systeme bestehen aus mehreren Komponenten, die miteinander verbunden sind und bei der Lösung einer Aufgabe zusammenwirken. Ihre Laufzeit ist in vielen Fällen zumindest prinzipiell unbeschränkt. Die Zahl der möglichen Systemzustände und Verhaltensmuster wächst dadurch sehr stark an, (kombinatorische Explosion) und es ist schwer, solche Systeme zu überblicken, und praktisch unmöglich, sie erschöpfend zu testen.
Darüber hinaus werden verteilte Systeme häufig in Umgebungen eingesetzt, in denen es auf ihre Korrektheit und Zuverlässigkeit entscheidend ankommt. Sie steuern Produktionsprozesse, kontrollieren Verkehrssicherungssysteme und überwachen Maschinen und Anlagen aller Art. Ausfälle oder Fehler können hier lebensbedrohliche, ja katastrophale Auswirkungen haben.
Aus all dem folgt, daß verteilte Anwendungen besonders sorgfältig entwickelt werden müssen, ein systematisches Vorgehen ist zwingend notwendig. Grundlage dafür ist ein konzeptueller Rahmen, der nicht nur Mittel zur Beschreibung von Systemversionen auf unterschiedlichen Abstraktionsstufen bereitstellt, sondern auch methodische Leitlinien für den Übergang zwischen diesen Stufen beinhaltet.

Während bereits eine Vielzahl *formaler Beschreibungstechniken* für verteilte Systeme existieren, z.B. temporale Logik [Kröger 87], Petrinetze [Reisig 85], Prozeßalgebren, insbesondere CCS [Milner 80] und (T)CSP [Hoare 85a], I/O-Automaten [Jonsson 87], Statecharts [Harel 87] u.v.a., ist die m*ethodische Einbettung* solcher Formalismen erst seit jüngster Zeit Gegenstand verstärkter Forschungsanstrengungen. Zu nennen sind in diesem Zusammenhang die "transition axiom method" von Lamport [Lamport 89] oder UNITY von Chandy und Misra [Chandy, Misra 88]. Weitere Vertreter werden in Abschnitt 1.4 behandelt.

Die vorliegende Arbeit verfolgt das Ziel, einen Ansatz zu erweitern und auszugestalten, der unter dem Namen FOCUS an der Universität Passau und der Technischen Universität München entwickelt wurde (vgl. [Broy 89], [Broy et al. 92a]). FOCUS erhebt den Anspruch, *durchgängige Systementwicklungen* ausgehend von einer ersten Anforderungsspezifikation über mehrere Zwischenstufen bis hin zu einem parallelen Programm, d.h. einem verteilten Softwaresystem, zu ermöglichen.
FOCUS gliedert den Entwicklungsprozeß in vier Phasen:

1. Anforderungsspezifikation

2. Entwurfsspezifikation

3. Abstrakte Implementierung

4. Konkrete Implementierung

Phasenmodelle wie diese sind aus der sequentiellen Programmierung bestens bekannt. Phaseneinteilung und Beschreibungsmittel sind hier jedoch auf die Besonderheiten verteilter Systeme abgestimmt.

In der ersten Phase des Entwicklungsprozesses (*Anforderungsspezifikation*) werden die Anforderungen an das extensionale, vom Benutzer beobachtbare Verhalten des Systems festgelegt. Technisch fixiert man zu diesem Zweck eine Menge nach außen sichtbarer *Systemaktionen* und beschreibt dann das erlaubte bzw. erwünschte Verhalten durch endliche oder unendliche Sequenzen solcher Aktionen.

In der zweiten Phase (*Entwurfsspezifikation*) wird die interne Struktur des geplanten Softwareprodukts entworfen und schrittweise verfeinert. Das System wird als Netz unabhängiger Einheiten dargestellt, die im folgenden *Agenten* genannt werden[1]. Agenten kommunizieren durch asynchronen Nachrichtenaustausch. Formal wird ein Agent durch eine *(Menge) stromverarbeitende(r) Funktion(en)* repräsentiert. Ströme sind endliche oder unendliche Sequenzen von Nachrichten. Eine stromverarbeitende Funktion bildet Tupel von Eingabeströmen auf Tupel von Ausgabeströmen ab. Die auf dieser Ebene nötige Flexibilität und Ausdrucksmächtigkeit gewinnt man durch Verwendung prädikatenlogischer Mittel bei der Agentendefinition. Konkret wird ein Agent durch ein Prädikat beschrieben, das eine Menge geeigneter Stromfunktionen festlegt (eigenschaftsorientierte Charakterisierung). Derartige Darstellungen sind meistens nicht ausführbar, die auftretenden Prädikate brauchen nicht konstruktiv zu sein. Der Übergang zu ausführbaren Darstellungen wird (spätestens) mit Eintritt in die dritte Phase vollzogen. Diese, die folgende Phase 4 sowie der Übergang zwischen beiden, stehen im Zentrum der vorliegenden Arbeit.

In der dritten Phase (*abstrakte Implementierung*) wird das System in einer *algorithmischen Sprache* mit festgelegter Syntax beschrieben. Die Definition einer solchen Sprache (AL) findet sich in Kapitel 3. Um den Übergang von der vorangegangenen Entwurfsphase zu erleichtern, ist sie applikativ und auf die Darstellung von Strömen, stromverarbeitenden Funktionen und Agentennetzen ausgerichtet. Ein in AL codiertes *"abstraktes Programm"* bildet einen ersten Zielpunkt des gesamten Entwicklungsprozesses. Es ist ausführbar und erfüllt die in Phase 1 vom Benutzer/Auftraggeber festgeschriebenen Anforderungen. Trotzdem kann man die Entwicklung an dieser Stelle in der Regel noch nicht als abgeschlossen betrachten: Abstrakte Programme sind zwar ausführbar, aber oft nicht effizient genug. Darüber hinaus ist es vielfach nötig, sie stärker an die geplante Zielumgebung, z.B. eine bestimmte Rechner- oder Prozessortopologie anzupassen. Bei der Entwicklung *"konkreter Programme"* rücken diese Aspekte in den Mittelpunkt.

Effizienzanforderungen können in vielfältiger Weise berücksichtigt werden. In der vorliegenden Arbeit wird vorgeschlagen, von den applikativen Programmen der Phase 3 zu *prozeduralen Programmen* in der Phase 4 (*konkrete Implementierung*) überzugehen. Das System wird auch in

---

[1] Für diese Einheiten gibt es in der Literatur eine ganze Fülle von Bezeichnungen, die alle eher informell benutzt werden und im wesentlichen dasselbe bedeuten. Dazu gehören: Agent, Prozeß, Task, Modul, Komponente, (Sub-)System, usw..

dieser Phase als Netz kommunizierender Agenten repräsentiert. Zur Darstellung der Agenten werden aber nun prozedurale Mittel benutzt. Kapitel 4 enthält die Definition einer dafür geeigneten Sprache (PL). Diese umfaßt neben den üblichen imperativen Konstrukten – lokale Variablen, Zuweisungen und Schleifen – eine Anweisung zur Parallelverarbeitung sowie ein Kanalkonzept zur asynchronen Kommunikation. Zwischen AL und PL besteht eine enge Verwandtschaft und zwar sowohl in Bezug auf die Syntax als auch in Bezug auf die (denotationelle) Semantik. Der Schritt von applikativen zu prozeduralen Implementierungen soll mit Hilfe *transformationeller Techniken* ausgeführt werden. Der folgende Abschnitt geht darauf genauer ein.

1.2 Programmtransformation

Programmtransformation ist eine strikt formale Methode der Softwarekonstruktion, die seit Ende der 70er Jahre von zahlreichen Forschern untersucht wurde und wird. Überblicke über die zahlreichen Arbeiten auf diesem Gebiet finden sich in [Feather 87] oder [Lowry, Duran 89]. Die nachfolgenden Ausführungen stützen sich insbesondere auf [Bauer et al. 89].

Die Idee des Transformationsansatzes besteht darin, aus einer *formalen Problembeschreibung* durch ausschließliche Anwendung *korrektheitserhaltender Transformationsregeln*, eine effiziente, algorithmische Lösung abzuleiten. Transformationsregeln formalisieren dabei algebraische Gesetzmäßigkeiten zwischen unterschiedlichen Programmversionen, z.B. semantische Äquivalenz oder eine Implementierungsbeziehung. Ihre Anwendung ermöglicht ein deduktives (top-down) Vorgehen und garantiert die Korrektheit abgeleiteter Versionen bzgl. der anfänglichen Spezifikation qua Konstruktion. Die Regeln werden schematisch und meist in einer Metasprache dargestellt. Sie "passen" daher nicht nur auf ein, sondern auf viele Programme und können entsprechend wiederverwendet werden. Aufgrund des *formalen Charakters* solcher Entwicklungen ist der gesamte Prozeß maschinell unterstützbar. Tatsächlich wurden zu diesem Zweck bereits eine ganze Reihe von Transformationssystemen implementiert (vgl. [Partsch, Steinbrüggen 83], [Lowry, Duran 89]).

Die Vorteile des Transformationsansatzes sind beeindruckend. Es hat sich aber auch gezeigt, daß es aufwendig ist, den Ansatz zu realisieren. Dies hat folgenden Grund: Bei konsequenter Auslegung des Transformationskonzeptes besteht jeder Schritt bei der Entwicklung eines Softwareprodukts aus der Anwendung einer Regel. Dies ist gerade in den frühen Entwicklungsphasen mühsam und wenig flexibel. Die Bandbreite möglicher Anforderungsspezifikationen, d.h. formaler Problembeschreibungen, ist so groß, daß es schwer fällt, eine passende Regel zu finden. In der Praxis läuft dies darauf hinaus, daß man eine neue Systembeschreibung intuitiv entwickelt, sie als korrekt beweist (bottom-up) und dann gezwungen ist, den gefundenen Zusammenhang nachträglich als Regel zu formulieren. Weil die Problemstellung spezifisch war, ist es auch die Regel, und somit ist es wenig wahrscheinlich, daß man sie wieder verwenden kann.

Diese Analyse begründet auch die Stellung der transformationellen Programmierung in FOCUS. FOCUS strebt eine ausgewogene Mischung von Top-Down- und Bottom-Up-Schritten an. Transformationelle Programmierung wird dabei im wesentlichen in den späten Phasen, insbesondere beim Übergang von Phase 3 zu Phase 4 eingesetzt. Die Entwicklungsschritte auf diesen Ebenen lassen sich in ausreichendem Maß schematisieren und können daher transformationell ausgeführt werden. Darüber hinaus ist das System hier in einer festgelegten Syntax beschrieben,

während in den vorangegangenen Phasen bewußt auf eine strenge syntaktische Fixierung verzichtet wurde.

1.3 Ziele, Methoden und Aufbau der Arbeit

Das globale Ziel dieser Arbeit wurde bereits genannt: Sie soll

- einen Beitrag zur Fortentwicklung von FOCUS, der oben beschriebenen Entwicklungsmethode für verteilte Systeme leisten.

Während das Schwergewicht der Arbeiten zu FOCUS bisher auf den frühen Phasen, d.h. Anforderungs- und Entwurfsspezifikation lag, steht hier die Implementierungsphase, unterteilt in abstrakte und konkrete Implementierung, im Vordergrund. Im einzelnen soll die Arbeit:

- für diese Teilphasen Sprachkonzepte bereitstellen, die sich möglichst gut in den bestehenden Rahmen einfügen und auch untereinander kompatibel sind,
- sie soll Möglichkeiten aufzeigen, wie der Übergang von abstrakten zur konkreten Implementierungen methodisch zu bewältigen ist,
- und schließlich nachweisen, daß transformationelle Techniken auch zur Entwicklung verteilter Systeme eingesetzt werden können.

Die Kapitel 3 und 4 sind der Erreichung des ersten Ziels gewidmet. Sie legen gleichzeitig das Fundament für die Untersuchungen in Kapitel 5, die das zweite und dritte Ziel betreffen.
Es folgt eine kurze Schilderung der zentralen Konzepte und Methoden der Arbeit. Da ein relativ enger Zusammenhang zu den einzelnen Kapiteln besteht, orientiert sich die Darstellung am Aufbau der Arbeit.

Nach einer kurzen Einführung in die technischen Grundlagen der *denotationellen Semantik* und insbesondere der *Stromverarbeitung* in **Kapitel 2**, wird in **Kapitel 3** die *applikative Sprache* AL definiert. In AL können Ströme, stromverarbeitende Funktionen und Agentennetze beschrieben werden. Ein einfacher AL-Agent sieht wie folgt aus:

agent *twothree* ≡ **chan nat** *i* → **chan nat** *o*:
o ≡ **if isempty.***i* **then** ε
else (2 * **ft.***i* ▯ 3 * **ft.***i*) & *twothree*(**rt.***i*) **fi**
end

twothree verarbeitet einen Strom *i* natürlicher Zahlen. Die Gleichung im Rumpf definiert den Ausgabestrom *o*. *twothree* multipliziert jedes Element von *i* entweder mit 2 oder mit 3 (ε beschreibt hier den leeren Strom und ▯ die nichtdeterministische Auswahl).

Agentennetze werden durch Gleichungssysteme repräsentiert. Formal definiert dabei jede Gleichung einen Knoten und jeder Strom eine Kante. Da Gleichungssysteme (mit mehr als einer Gleichung) auch im Rumpf von Agenten auftreten können, ist ein hierarchischer Netzaufbau möglich; einzelne Knoten/Agenten sind selbst wieder durch Netze realisierbar.
AL besitzt eine denotationelle Semantik. Ihr Kernstück ist die Zuordnung von Mengen stromverarbeitender Funktionen zu Agentendefinitionen. Damit ist nicht nur der kompositionale Aufbau von AL-Programmen gewährleistet, sondern auch der geforderte Anschluß an die vorangegangene Entwurfsphase: Dort werden Agenten durch Prädikate definiert, die ebenfalls Funktionsmengen charakterisieren. Ein AL-Agent *f* realisiert (bzw. implementiert) einen prädikativ entworfenen Agenten *F*, wenn die durch *F* beschriebene Funktionsmenge und die Menge, die die AL-Semantik *f* zuordnet, identisch sind (bzw. die erste eine Obermenge der zweiten ist).

Kapitel 4 enthält die Definition der *prozeduralen Sprache* PL. Um die geforderte Kompatibilität zu AL herzustellen, ist PL sowohl syntaktisch als auch semantisch an AL angenähert. Durch die prozeduralen Elemente – Variablen, Zuweisungen, Schleifen –, insbesondere aber durch die Substitution der Ströme durch Kanäle, wird trotzdem ein wesentlicher Schritt in Richtung auf konkrete Rechnerarchitekturen getan. Die Implementierung von PL ist kein Bestandteil dieser Arbeit. Ich glaube aber, daß es ohne grundsätzliche Schwierigkeiten möglich wäre, sie auf einem Rechner mit verteiltem Speicher zu implementieren (vgl. dazu Aussagen in Abschnitt 4.3). Die prozedurale Version von *twothree* hat folgende Gestalt:

agent *twothree* ≡ **chan nat** *i* → **chan nat** *o*:
    var nat *x*;
    while ¬**isclosed.***i* **do** *i*?*x*; *o*!(2 * *x* ▯ 3 * *x*) **od**;
    close.*o*
end

? und ! stehen hier für den lesenden bzw. schreibenden Zugriff auf Kanäle. **close.***o* schließt den Ausgabekanal *o* und **isclosed.***i* prüft, ob der Eingabekanal *i* geschlossen wurde.

Die denotationellen Semantiken von AL und PL sind voll kompatibel. Auch PL-Agenten werden semantisch durch Mengen stromverarbeitender Funktionen gedeutet. Daraus ergeben sich zwei entscheidende Vorteile: Zum einen ist die *Korrektheit des Übergangs* von AL nach PL kanonisch definierbar, nämlich genauso wie der Übergang von prädikativen Agenten zu AL-Agenten. Zum anderen ist es möglich, *gemischten Darstellungen* aus applikativen und prozeduralen Agenten eine präzise Bedeutung zuzuweisen. Obwohl beide Sprachen getrennt eingeführt werden, kann man sie daher als Einheit betrachten; sie bilden eine *Breitbandsprache*. Das ist eine wichtige Voraussetzung für die schrittweise Anwendung lokaler Transformationsregeln.

Transformationelle Techniken wurden bisher vor allem in der sequentiellen Programmierung angewandt. Erst in jüngster Zeit gibt es Anstrengungen, auch verteilte Systeme auf diese Weise zu konstruieren. In **Kapitel 5** wird untersucht, wie transformationelle Techniken in FOCUS eingebettet werden können. Andere Ansätze dieser Art werden im anschließenden Abschnitt 1.4 kurz diskutiert. Das fünfte Kapitel gliedert sich in drei Teile.
Im ersten Teil wird die syntaktische Form von *Transformationsregeln* beschrieben und ein Korrektheitsbegriff festgelegt. Dieser fußt auf den Sprachsemantiken und ist aufgrund der Uniformität der Semantiken einfach zu formulieren. Methodisch bedeutsam ist auch, daß sich Transformationsregeln lokal, d.h. auf kleinere Bestandteile vollständiger Programme anwenden

lassen. Nur dadurch ist ein inkrementelles Vorgehen möglich. Alle Regeln dieser Arbeit sind lokal anwendbar.

Der zweite Abschnitt des Kapitels beschäftigt sich mit der Transformation von AL-Programmen. Im wesentlichen finden sich hier Regeln, die ein AL-Programm in ein anderes überführen, wobei Regeln, die die Verteilungsstruktur verändern von besonderem Interesse sind. Man kann z.B. die Anzahl parallel arbeitender Programmkomponenten variieren, indem man Rückkopplungsschleifen durch Regelanwendung auf- oder abwickelt. Formal handelt es sich dabei um Regeln zur *Transformation von Gleichungssystemen*.

Im dritten Abschnitt wird schließlich der Übergang zu prozeduralen Darstellungen untersucht. Es zeigt sich, daß eine bestimmte syntaktisch ausgezeichnete Klasse von AL-Agenten besonders einfach in PL-Agenten überführt werden kann, deren Rumpf im wesentlichen aus einer while-Schleife besteht. Nach den Erfahrungen aus dem "Sequentiellen", war ein derartiges Resultat zu erwarten. In Anlehnung an die dortige Begriffswahl, nenne ich die entsprechenden Agenten *stromrepetitiv*. Der Agent *twothree* ist von dieser Form.
Obwohl alle stromrepetitiven Agenten definitionsgemäß bestimmte Gemeinsamkeiten aufweisen, können sie sich stark unterscheiden. Eine einheitliche Behandlung der gesamten Klasse wird durch einen Metakalkül möglich, der es erlaubt, zu jedem stromrepetitiven Agenten eine passende Transformationsregel (d.h. eine passende prozedurale Form) abzuleiten. Besondere Aufmerksamkeit muß dabei der Tatsache gewidmet werden, daß AL mit Strömen und PL mit Kanälen arbeitet.
Stromrepetitive Agenten bilden nur eine Teilklasse der Menge aller AL-Agenten. Transformationsregeln für sie sind daher nur beschränkt einsetzbar. Weitgehend universell ist dagegen folgender Ansatz: Aufgrund der syntaktischen Übereinstimmung zwischen bestimmten AL-Gleichungssystemen und der PL-Parallelanweisung, ist es möglich, AL-Agenten durch Transformation der Gleichungssysteme in ihrem Rumpf auf eine Form zu bringen, die als PL-Parallelanweisung interpretiert werden kann.
Dieser Übergang ist (fast) immer möglich. Er führt aber nicht in jedem Fall zu befriedigenden Ergebnissen, da (unter anderem) die rekursive Struktur der applikativen Ebene voll auf die prozedurale Ebene übertragen wird. Es ergeben sich (zur Laufzeit) *dynamisch wachsende*, potentiell unbeschränkte Netzen. Den Abschluß von Kapitel 5 bilden daher Untersuchungen, ob, und wie, auch nicht-repetitive Agenten durch Netze realisiert werden können, deren Strukturen *statisch* sind, d.h. sich zur Laufzeit nicht verändern.
Die *Korrektheit* der meisten in Kapitel 5 angeführten Transformationsregeln wird formal, d.h. bezüglich der denotationellen Semantiken von AL und PL, nachgewiesen. Die semantischen Definitionen aus den Kapiteln 3 und 4 fließen direkt in die entsprechenden Beweise ein. Der dort getriebene Aufwand, der sich nicht zuletzt im Umfang dieser Kapitel äußert, zahlt sich nun insofern aus, als daß die Beweise zwar eine gewisse technische Komplexität aufweisen, konzeptuell aber einfach sind. Als besonders vorteilhaft erweist sich in diesem Zusammenhang die bewußt einheitliche Ausgestaltung der beiden Semantiken.

Kapitel 6 faßt die erarbeiteten Ergebnisse schließlich noch einmal knapp zusammen und gibt einen Ausblick auf weitere Untersuchungen.

1.4 Vergleich mit anderen Ansätzen

In diesem Abschnitt sollen in aller Kürze andere Ansätze geschildert werden, die sich mit der methodischen und insbesondere der transformationellen Entwicklung verteilter Systeme beschäftigen.

(Vor-)Arbeiten von Broy
Broy hat sich in zahlreichen Veröffentlichungen mit der Spezifikation und methodischen Entwicklung verteilter Systeme beschäftigt. Seine Arbeiten bilden ein wesentliches Fundament der vorliegenden Arbeit. Von besonderem Interesse sind in diesem Zusammenhang neben den Arbeiten, die sich mit der Konzeption von FOCUS beschäftigen (vgl. [Broy 89, 90]), speziell jene, in denen transformationelle Techniken eine Rolle spielen.
Konzepte, Sprachelemente und typische Phänomene der parallelen Programmierung werden schon in [Broy 80] untersucht. Obwohl dabei der Übergang zwischen applikativen und prozeduralen Sprachstilen eine Rolle spielt, liegt das Schwergewicht hier darauf, "von einer im Prinzip funktionalen Programmiersprache ausgehend, Sprachelemente für die parallele Programmierung herzuleiten, und diese mit klassischen sequentiellen Sprachelementen zu integrieren" ([Broy 80], S. 4). Bei der angesprochenen funktionalen Sprache handelt es sich um (einen Vorläufer von) CIP-L (vgl. [CIP 85]). Broys Arbeit zielt u.a. darauf ab, das ursprünglich "sequentielle" CIP-L, mit Hilfe "definierender Transformationen" um parallele Konstrukte zu erweiteren.
Dezidiert auf die Darstellung paralleler Systeme ist die in [Broy 86] entwickelte Sprache AMPL ("applicative multiprogramming language") ausgerichtet. AL, die applikative Sprache dieser Arbeit, ist an AMPL angelehnt, unterscheidet sich von ihr jedoch in zweierlei Hinsicht: Zum einen bietet AL dem Programmierer die Möglichkeit zum hierarchischen Netzaufbau, d.h. es erlaubt Gleichungssysteme und nicht nur Ausdrücke im Rumpf von Agenten. Zum anderen verzichtet AL auf ein in AMPL vorhandenes Sprachelement, den sogenannten "Ambiguity-Operator" ∇, der erhebliche Probleme bei der semantischen Behandlung aufwirft (vgl. Abschnitt 3.4). Die präzise semantische Beschreibung von AMPL mit Hilfe denotationeller Techniken (inklusive "power domains") steht im Zentrum von [Broy 86]. Diese Arbeit ist somit eher theoretisch angelegt und streift methodische Aspekte nur am Rande.
Breiter Raum ist der Methodik in [Broy 87a] gewidmet. Grundzüge des transformationellen Übergangs von applikativen zu prozeduralen verteilten Programmen werden hier angedeutet. Insbesondere finden sich Hinweise, wie unbeschränkte Netze durch statische Netze abgelöst werden können (vgl. [Broy 87a], S. 254-256).
Aus den vielen Fallstudien, die im Rahmen von FOCUS erstellt wurden (siehe [Broy et al. 92b]) soll schließlich noch [Broy 88a] erwähnt werden. In dieser Studie geht es um die Entwicklung eines Liftsystems. Sie zeichnet sich dadurch aus, daß alle vier Entwicklungsphasen abgedeckt sind, am Ende also ein prozedurales Programm steht. Die Übergänge zwischen Phasen werden allerdings nicht durch Transformationen vollzogen.

Arbeiten von Barstow
Barstow verfolgt in seinen Veröffentlichungen [Barstow 85, 88] ein ehrgeiziges Ziel: Es geht ihm um "automatische Programmentwicklung für Ströme". Idealerweise sollen dabei kommunizierende, prozedurale Programme aus einer abstrakten Problembeschreibung durch automatische Anwendung von Transformationsregeln abgeleitet werden.
Ausgangspunkt eines automatischen Deduktionsvorgangs ist eine Spezifikation im Vorbedingungs- / Nachbedingungsstil, wobei in diesen Bedingungen Stromvariable auftreten dürfen (vgl. [Barstow 85], S. 233). Solche Spezifikationen werden in applikative Darstellungen

überführt, in denen Ausgabeströme durch Kombination bestimmter Relationen beschrieben werden. In [Barstow 85] sind acht "Stromrelationen" angegeben. Sie entsprechen direkt einfachen AL-Agenten. Barstow motiviert die Festlegung auf gerade diese Relationen durch Erfahrungen aus seinem speziellen Anwendungsgebiet: der Programmentwicklung für Ölförderanlagen.
Die applikativen Programme werden dann in prozedurale Form überführt. Zielsprache ist dabei wie PL eine "konventionelle" prozedurale Sprache, die um Zugriffsoperationen auf Ströme/Kanäle angereichert wurde ("produce", "consume"). Der Übergang erfolgt mit Transformationsregeln, die für die einzelnen Basisrelationen definiert sind (vgl. [Barstow 85], S. 234).

In [Barstow 88] berichtet der Autor über die Implementierung des Transformationssystems ΦNIX (vgl. auch [Lowry, Duran 89], S. 320-321), das die oben skizzierte Entwicklungsmethode auf der Maschine realisieren soll. Insbesondere findet sich dort ein einfaches Beispiel, das ausschließlich transformationell entwickelt wurde. Von vollständiger Automatisierung kann aber bislang noch keine Rede sein, da die Auswahl der anzuwendenden Transformationsregel weiterhin dem Benutzer überlassen bleibt.
Konzeptuell gibt es einen engen Zusammenhang zwischen Barstows Ansatz und dem Ansatz dieser Arbeit. Der Hauptunterschied besteht neben der Tatsache, daß AL breiter angelegt zu sein scheint als die Kombinatorsprache in [Barstow 85], vor allem in der semantischen Behandlung: Barstow verzicht vollständig auf eine formale Fundierung seiner Sprachen und macht (konsequenterweise) auch über die Korrektheit seiner Regeln keine Aussage. Dieser Aspekt ist für die vorliegende Arbeit zentral.

PROCOS
PROCOS ist das Akronym des ESPRIT BRA Projektes 3104 und steht für "Provably Correct Systems". Ziel dieses, von einer internationalen Forschergruppe betriebenen Vorhabens, ist es, den Kenntnisstand auf dem Gebiet der systematischen Entwicklung komplexer, kommunizierender Systeme voranzutreiben (vgl. [Bjørner et al. 89]). Das anvisierte Arbeitsfeld ist dabei weitgespannt. Es reicht von der Definition einer Spezifikationssprache über die Festlegung einer Programmiersprache bis hin zur Entwicklung einer Maschinensprache und schließt die Beschreibung der zugehörigen Hardware mit ein. Den Übergängen zwischen den Abstraktionsstufen wird besondere Beachtung geschenkt.
Eine Spezifikationssprache SL_0 und eine OCCAM-ähnliche Programmiersprache PL (nicht zu verwechseln mit der prozeduralen Sprache aus Kapitel 4) sind inzwischen definiert worden. Beschreibungen finden sich z.B. in [Olderog 91].
Eine SL_0-Spezifikation besteht aus zwei Teilen, einem "Spurspezifikationsteil" und einem "Zustandsspezifikationsteil". Der erste Teil legt die Abfolge der Kommunikationsaktionen auf den Interface-Kanälen des geplanten Systems fest. Der zweite bezieht sich auf die kommunizierten Werte und deren Effekt auf den Systemzustand.
SL_0-Spezifikationen werden durch Anwendung korrektheitserhaltender Transformationsregeln in PL-Programme überführt. Die entstehenden Zwischenversionen heißen "mixed terms". Sie enthalten sowohl OCCAM-Operatoren, wie SEQ, ALT, PAR usw., als auch abgeleitete Spezifikationen. In [Olderog 91] ist eine Sprache MIX definiert, mit der alle Zwischenstufen, die im Zuge einer Entwicklung entstehen, beschreibbar sind und die SL_0 und PL als (echte) Untermengen enthält.
Alle drei Sprachen besitzen eine uniforme prädikative Semantik. Konkret werden SL_0-Spezifikationen, gemischte Terme und PL-Programme durch Prädikate über denselben freien Variablen gedeutet. Grundlage ist ein "kombiniertes state-trace-readiness Modell". Transformationsregeln setzen Spezifikationen/gemischte Terme und Programme/gemischte Terme zueinander in Beziehung. Ihre Korrektheit wird durch prädikatenlogische Argumentation auf der Grundlage der einheitlichen prädikativen Semantik nachgewiesen. Olderog benennt in seinem

Papier zwei Gruppen von Transformationsregeln (vgl. [Olderog 91], S. 71). Die eine führt zu sequentiellen, die andere zu parallelen Programmen.
Die grundlegenden Ansätze von PROCOS und FOCUS weisen starke Ähnlichkeiten auf. Beiden geht es um systematische Programmentwicklung mit Hilfe formaler Techniken. Beide betonen die Notwendigkeit einer durchgängigen Methode. Technisch gibt es aber bedeutende Unterschiede. FOCUS ist primär auf asynchrone Kommunikation ausgerichtet, während PROCOS synchrone Mechanismen betont. PROCOS legt die Syntax einer Spezifikationssprache fest, während FOCUS zwar (mathematisch orientierte) Sprachmittel anbietet, dem Anwender aber Raum für selbstgewählte Notationen läßt. Schließlich steht das Konzept der Programmtransformation in PROCOS insofern stärker im Vordergrund, als schon Spezifikationen transformationell entwickelt werden sollen. Auf jeden Fall ist PROCOS einer der wenigen (mir bekannten) Ansätze, in denen die Korrektheit von Transformationsregeln bezüglich unabhängig definierter Semantiken bewiesen wird.

PROSPECTRA
Das ESPRIT-Projekt PROSPECTRA (PROgram Development by SPECification and TRAnsformation) ist ein Transformationsprojekt "reinsten Wassers". Jeder Schritt einer Programmentwicklung wird hier "konzeptuell und technisch als die Transformation eines Programms aufgefaßt" (vgl. [Krieg-Brückner 90], Vol. I, S. 1-1).
Im Zentrum des inzwischen abgeschlossenen Projektes stand die Konzeption einer Sprachfamilie (PANNDA-S, TRAFOLA, CONTROLA), die Bereitstellung einer Bibliothek von Transformationsregeln, sowie die Implementierung eines umfassenden Unterstützungssystems. Alle Bestandteile wurden methodische eingebettet.
Ausgangspunkt einer PROSPECTRA-Entwicklung ist eine PANNDA-S-Spezifikation. Endpunkt ist ein ADA-ähnliches Programm. Aus der algebraisch orientierten PANNDA-S-Spezifikation wird durch schrittweise Transformation eine imperative Implementierung abgeleitet. Die Korrektheit der Transformationsschritte beruht dabei auf einem Implementierungsbegriff für algebraische Spezifikationen (vgl. [Breu 90]), der sich wiederum auf die PANNDA-S-Semantik abstützt.
Verteilte Systeme werden in PROSPECTRA nur am Rande behandelt. Es ist aber möglich, Systemspezifikationen auf der Grundlage von Strömen und stromverarbeitenden Funktionen in PANNDA-S auszudrücken (vgl. [Weber 90]). Der Modellbegriff der Semantik ist so gestaltet, daß endliche und unendliche Ströme monomorph spezifiziert werden können. Schwierigkeiten entstehen in den frühen Phasen, wenn neben Sicherheits- auch Lebendigkeitseigenschaften ausgedrückt werden sollen. Letzere sind in der Regel nicht monoton (bzgl. der Präfixordnung auf Strömen). PAnndA-S stützt sich aber auf monotone Funktionen und Prädikate.
Die Randstellung von verteilten Systemen in PROSPECTRA wird aber vor allem in der Tatsache deutlich, daß es keine speziellen Transformationsregeln für diesen Anwendungsbereich gibt.

RAISE
Wie PROCOS und PROSPECTRA ist auch RAISE ein von der EG im Rahmen des ESPRIT-I -Programms gefördertes Projekt (#315). RAISE (Rigorous Approach to Industrial Software Engineering) wird von einem dänisch/britischem Industriekonsortium getragen und sieht sein Ziel entsprechend darin, den Gebrauch formaler Methoden für die Software- und Systementwicklung in einem industriellen Umfeld zu fördern (vgl. [Eriksen, Prehn 91], S. 1).
Die Ergebnisse des mittlerweile ebenfalls beendeten Projekts bestehen aus einer Spezifikationssprache RSL, einer Methodik, die die Anwendung von RSL unterstützt aber auch Managementaspekte berücksichtigt, sowie einer Anzahl integrierter Werkzeuge, die auf Sprache und Methodik abgestimmt sind.

RSL ist eine "wide-spectrum" Sprache, in der sich sowohl abstrakte, eigenschaftsorientierte Spezifikationen, als auch implementierungsnahe Systemdarstellungen aufschreiben lassen. "Low-Level-Designs" können automatisch in Programmcode umgesetzt werden. Werkzeuge zur Erzeugung von ADA- bzw. C++-Code sind geplant bzw. als Prototypen verfügbar (vgl. [Eriksen, Prehn 91], S. 31).
RSL unterstützt verschiedene Spezifikationsstile. Dazu gehören eine algebraisch-axiomatische Variante und vor allem ein modellorientierter Stil, der maßgeblich von VDM bzw. Z beeinflußt ist. Verteilte Systeme können auf unterschiedlichen Abstraktionsstufen repräsentiert werden. Die Sprachmittel dafür orientieren sich inhaltlich an CCS, auch wenn sie syntaktisch an CSP erinnern (vgl. [George, Milne 91]).
Die RAISE-Methodik beruht auf dem Begriff der Verfeinerung; Transformationstechniken spielen keine Rolle. Eine gegebene Spezifikation S wird in zwei Schritten verfeinert: Als erstes entwickelt man intuitiv eine neue Spezifikation S' und zeigt dann (Bottom-Up), daß S' zu S äquivalent ist bzw. S implementiert. Solche nachträglichen Verifikationsschritte sind auch in FOCUS wichtig. Bei Annäherung an die Implementierung werden sie jedoch zunehmend durch reine "Top-Down-Schritte", nämlich durch Transformationsanwendungen, abgelöst. RAISE setzt ausschließlich auf Verifikationsschritte und bietet dazu auch Werkzeugunterstützung. Pragmatisch bedeutsam ist schließlich noch das in RAISE verfolgte Konzept, den Anwender nicht zur Formalisierung zu zwingen, sondern statt dessen unterschiedliche Formalisierungsgrade anzubieten.

Die angeführte Liste von fünf Ansätzen, die in der einen oder anderen Weise mit dem Konzept der vorliegenden Arbeit vergleichbar sind, kann keinen Anspruch auf Vollständigkeit erheben. Eine Vielzahl neuerer Arbeiten wie z.B. [Franchez, Forman 91], [Back, Sere 91] oder [Pepper 91] gehören sicher mit in diesen Zusammenhang. Sie behandeln aber (meines Erachtens) einzelne Aspekte der methodischen Entwicklung verteilter Systeme oder stützen sich auf spezialisierte Techniken und sollen daher an dieser Stelle nicht ausführlicher diskutiert werden.

2. Bereichstheoretische Grundlagen, Ströme

Die Semantik der beiden in den anschließenden Kapiteln eingeführten Sprachen AL und PL wird mit Hilfe denotationeller Techniken angegeben. Dieser Abschnitt beschreibt kurz die dazu notwendigen bereichstheoretischen Grundlagen (vgl. [Mosses 90], [Gunter, Scott 90]) und führt dann Ströme und stromverarbeitende Funktionen als grundlegende Konzepte ein.

Sei **D** eine Menge und $\sqsubseteq$ eine darauf definierte *partielle Ordnung*. Eine *Kette* $(d_i \mid i \in \mathbf{Nat})$ in **D** ist eine abzählbar unendliche Folge aufsteigend angeordneter Elemente aus **D**:

$$d_0 \sqsubseteq d_1 \sqsubseteq \ldots \sqsubseteq d_i \sqsubseteq \ldots .$$

Ein $d \in \mathbf{D}$ heißt *obere Schranke* von $(d_i \mid i \in \mathbf{Nat})$, wenn alle Kettenglieder d_i kleiner oder gleich d sind:

$$\forall i: d_i \sqsubseteq d.$$

Die kleinste obere Schranke einer Kette heißt *Supremum*. Sie wird, sofern sie existiert, mit $\bigsqcup_i (d_i \mid i \in \mathbf{Nat})$ bzw. kurz mit $\bigsqcup_i d_i$ bezeichnet. **D** heißt *Bereich* (engl. "Domain") oder *vollständige Halbordnung* (engl. "complete partial order", (ω-)cpo), falls gilt:

- jede Kette in **D** besitzt ein Supremum (in **D**),
- **D** enthält ein kleinstes Element $\bot$.

Mit der Bereichsordnung $\sqsubseteq$ ist die Intention verbunden, Elemente $d_1, d_2 \in \mathbf{D}$ bezüglich ihres "Informationsgehalts" zu vergleichen: Gilt $d_1 \sqsubseteq d_2$, dann ist die von d_1 transportierte Information mit der von d_2 verträglich, d_2 transportiert jedoch möglicherweise zusätzliche Information. d_1 heißt dann *Approximation* von d_2. $\bot$ approximiert alle anderen Elemente, es steht für die leere Information ("undefiniert"). Ein $d \in \mathbf{D}$ heißt *endlich*, falls für alle Ketten $(d_i \mid i \in \mathbf{Nat})$ aus **D** gilt:

$$d \sqsubseteq \bigsqcup_i d_i \Rightarrow \exists i: d \sqsubseteq d_i.$$

Andernfalls heißt d *unendlich*. d heißt *total*, wenn kein echt größeres Element existiert, und *partiell* sonst.
Neue Bereiche lassen sich mit Hilfe geeigneter Operatoren aus bereits definierten Bereichen ableiten. Aus der Vielzahl möglicher Konstruktionen sollen hier nur die im folgenden benötigen eingeführt werden.

Flache Bereiche. Sei **D** eine Menge und $\perp$ ein Element, das nicht in **D** vorkommt. Dann ist $\mathbf{D}^{\perp} = \mathbf{D} \cup \{\perp\}$ der *flache Bereich* über **D** mit der Ordnung ($d_1, d_2 \in \mathbf{D}^{\perp}$):

$$d_1 \sqsubseteq d_2 \quad \Leftrightarrow \quad d_1 = d_2 \vee d_1 = \perp.$$

Während $\perp$ offensichtlich das kleinste Element aus $\mathbf{D}^{\perp}$ ist, sind die Elemente aus **D** selber paarweise unvergleichbar und total.

Potenzmengen. Geordnet durch die übliche Mengeninklusion $\subseteq$ und mit $\emptyset$ als kleinstem Element bildet auch die Potenzmenge $\wp(\mathbf{D})$ über **D** einen Bereich. Das Supremum einer Kette $(\mathbf{D}_i \mid i \in \mathbf{Nat})$, $\mathbf{D}_i \subseteq \mathbf{D}$, ist hier gerade die Vereinigung aller Kettenglieder: $\bigsqcup_i \mathbf{D}_i = \bigcup_i \mathbf{D}_i$.
Für unsere Zwecke ist jedoch nicht der Potenzmengenbereich selber, sondern der *Präbereich* $\wp(\mathbf{D})\backslash\emptyset$ bedeutsam. $\wp(\mathbf{D})\backslash\emptyset$ ist zwar kettenvollständig, bildet jedoch keinen Bereich, da ein kleinstes Element fehlt.

Produktbereiche. Seien $\mathbf{D_1}, \mathbf{D_2}, \ldots, \mathbf{D_n}$ Bereiche mit den Ordnungen $\sqsubseteq_i$ und den kleinsten Elementen $\perp_i$. Dann ist das *Produkt* $\mathbf{D_1} \times \ldots \times \mathbf{D_n} = \{(d_1, \ldots, d_n) \mid d_i \in \mathbf{D_i}\}$ mit der komponentenweisen Ordnung:

$$(d_1, \ldots, d_n) \sqsubseteq (d_1', \ldots, d_n') \quad \Leftrightarrow \quad \forall i: d_i \sqsubseteq_i d_i'$$

und dem kleinsten Element $(\perp_1, \ldots, \perp_n)$ ebenfalls ein Bereich. Dabei nehmen wir an, daß $\times$ assoziativ ist, d.h.:

$$\mathbf{D_1} \times (\mathbf{D_2} \times \mathbf{D_3}) = (\mathbf{D_1} \times \mathbf{D_2}) \times \mathbf{D_3} = \mathbf{D_1} \times \mathbf{D_2} \times \mathbf{D_3}.$$

$\times$ ist also der Operator des *assoziativen kartesischen Produkts* (vgl. dazu die Definition des "smash"-Produkts in [Gunter, Scott 90]). Die Assoziativitätskonvention impliziert, daß geschachtelte Tupel stets mit den linearisierten Versionen identifiziert werden: ((a,b),(c,(d,e))) ist äquivalent zu (a,b,c,d,e). Dadurch wird die formale Komposition von Funktionen mit mehrstelligen Ein- und Ausgaben erleichtert.

Für $n \in \mathbf{Nat}$, $n \neq 0$, sei $\mathbf{D}^n = \mathbf{D} \times \ldots \times \mathbf{D}$ (n-mal) der Bereich aller n-stelligen Tupel über D und $\mathbf{D}^*$ der Bereich aller endlichen Tupel über **D**.

Summenbereiche. Seien $\mathbf{D_1}, \mathbf{D_2}, \ldots, \mathbf{D_n}$ paarweise disjunkte Bereiche. Dann ist die *Summe* $\mathbf{D_1} \cup \ldots \cup \mathbf{D_n} = \mathbf{D_1}\backslash\{\perp_1\} \cup \ldots \cup \mathbf{D_n}\backslash\{\perp_n\} \cup \{\perp\}$ der Bereich mit der Ordnung ($d, d' \in \mathbf{D_1} \cup \ldots \cup \mathbf{D_n}$):

$$d \sqsubseteq d' \quad \Leftrightarrow \quad d = \perp \vee \exists i: d, d' \in \mathbf{D_i} \wedge d \sqsubseteq_i d'$$

und dem kleinsten Element $\perp$. Die $\perp_i$ der Summanden werden hier mit dem neuen $\perp$ identifiziert.

Funktionsbereiche. Funktionsbereiche spielen in unserem Ansatz eine herausgehobene Rolle. Sie sind für monotone und stetige Funktionen definiert. Eine Funktion f: $\mathbf{D_1} \to \mathbf{D_2}$ heißt *monoton*, wenn für alle d,d'$\in \mathbf{D_1}$ gilt:

$$d \sqsubseteq_1 d' \Rightarrow f(d) \sqsubseteq_2 f(d').$$

f heißt *stetig*, wenn es monoton ist und zusätzlich für jede Kette $(d_i \mid i \in \mathbf{Nat})$ aus $\mathbf{D_1}$ gilt:

$$\bigsqcup_i f(d_i) = f(\bigsqcup_i d_i).$$

f heißt *strikt*, wenn gilt:

$$f(\perp_1) = \perp_2.$$

Die Menge aller stetigen Funktionen von $\mathbf{D_1}$ nach $\mathbf{D_2}$ wird mit $[\mathbf{D_1} \to \mathbf{D_2}]$ bezeichnet. Sie bildet mit der punktweisen Ordnung (f,g$\in[\mathbf{D_1} \to \mathbf{D_2}]$):

$$f \sqsubseteq g \quad \Leftrightarrow \quad \forall d \in \mathbf{D_1}: f(d) \sqsubseteq_2 g(d).$$

einen Bereich, dessen kleinstes Element die vollständig undefinierte Funktion $\perp$: $\mathbf{D_1} \to \mathbf{D_2}$ ist, die jedes d$\in \mathbf{D_1}$ auf $\perp(d) = \perp_2$ wirft.
Die Teilmenge aller strikten, stetigen Funktionen wird mit $[\mathbf{D_1} \to \mathbf{D_2}]_s$ bezeichnet. Sie bildet ebenfalls einen Bereich. Sei f$\in[\mathbf{D_1} \to \mathbf{D_2}]$, dann bezeichnet strict f die zugehörige strikte Funktion aus $[\mathbf{D_1} \to \mathbf{D_2}]_s$. Der Operator "strict" ist selber eine stetige Abbildung von $[\mathbf{D_1} \to \mathbf{D_2}]$ nach $[\mathbf{D_1} \to \mathbf{D_2}]_s$.

f $\circ$ g bezeichnet die Komposition von Funktionen f$\in [\mathbf{D_2} \to \mathbf{D_3}]$ und g$\in [\mathbf{D_1} \to \mathbf{D_2}]$. Auch "$\circ$" ist eine stetige Abbildung von $[\mathbf{D_2} \to \mathbf{D_3}] \times [\mathbf{D_1} \to \mathbf{D_2}]$ nach $[\mathbf{D_1} \to \mathbf{D_3}]$. Die Komposition stetiger Funktionen liefert also wieder eine stetige Funktion. Für monotone Funktionen gilt der folgende zentrale Satz:

Satz 2.1 (Knaster, Tarski): Ist $\mathbf{D}$ ein Bereich und f: $\mathbf{D} \to \mathbf{D}$ monoton, dann besitzt die Gleichung

$$d = f(d)$$

eine (eindeutig bestimmte) kleinste Lösung d^*. d^* heißt der *kleinste Fixpunkt* von f und wird mit fix(f) bezeichnet □

Monotonie garantiert die Existenz kleinster Fixpunkte. Sie gewährleistet jedoch nicht, daß kleinste Fixpunkte als Grenzwerte finiter Iterationsprozesses erreicht werden können. Dazu benötigt man die stärkere Eigenschaft der Stetigkeit.

Satz 2.2 (Kleene): Ist **D** ein Bereich und f: **D** $\rightarrow$ **D** stetig, dann besitzt f einen kleinsten Fixpunkt fix(f) und es gilt:

$$fix(f) = \bigsqcup_i f^i(\bot),$$

wobei für alle $d \in \mathbf{D}$: $f^0(d) = d$ und $f^{i+1}(d) = f^i(f(d))$. □

Die Existenz von Fixpunkten ist entscheidende Voraussetzung dafür, daß rekursiven Funktions- und Stromdefinitionen eine Bedeutung zugewiesen werden kann. Kleinste Fixpunkte besitzten darüber hinaus eine intuitive Interpretation, die von der operationellen Interpretation der sog. Kleene-Kette ($f^i(\bot) \mid i \in \mathbf{Nat}$) herrührt. Um die Semantikbeschreibungen der vorgestellten Sprachen einfach zu halten, stützen wir uns im folgenden jedoch auch auf die größten Fixpunkte rekursiver Definitionen. Dafür ist folgende Aussage wichtig:

Satz 2.3 (Größte Fixpunkte mengenwertiger Funktionen): Sei **M** eine nicht-leere Menge und f: $\wp(\mathbf{M})\backslash\emptyset \rightarrow \wp(\mathbf{M})\backslash\emptyset$ eine Funktion für die gilt:

i) f ist stetig bzgl. $\subseteq$,
ii) $\exists \mathbf{N} \in \wp(\mathbf{M})\backslash\emptyset: \mathbf{N} \subseteq f(\mathbf{N})$.

Dann besitzt f einen (eindeutig bestimmten) *größten Fixpunkt* FIX(f).

Beweis: a) Wir zeigen zuerst die Existenz eines Fixpunktes. Sei **N** eine nicht-leere Menge, für die gilt: $\mathbf{N} \subseteq f(\mathbf{N})$. Aufgrund der Monotonie von f bildet ($f^i(\mathbf{N}) \mid i \in \mathbf{Nat}$) eine Kette. Wegen der Stetigkeit von f gilt für die kleinste obere Schranke $\bigcup_i f^i(\mathbf{N})$ dieser Kette:

$$f(\bigcup_i f^i(\mathbf{N})) = \bigcup_i f^{i+1}(\mathbf{N})) = \bigcup_i f^i(\mathbf{N})$$

Also ist $\bigcup_i f^i(\mathbf{N})$ ein Fixpunkt von f.
b) Sei $FIX \subseteq \wp(\mathbf{M})\backslash\emptyset$ die Menge aller Fixpunkte von f. Wegen a) ist *FIX* nicht-leer. Wir zeigen, daß die Vereinigung über alle Fixpunkte $(\bigcup_{\mathbf{N} \in FIX} \mathbf{N}) \in \wp(\mathbf{M})\backslash\emptyset$ wieder ein Fixpunkt ist. Dies ist dann offensichtlich der größte. Es gilt wegen der Monotonie von f:

$$\bigcup_{\mathbf{N} \in FIX} \mathbf{N} \subseteq f(\bigcup_{\mathbf{N} \in FIX} \mathbf{N}).$$

Also erfüllt $\bigcup_{\mathbf{N} \in FIX} \mathbf{N}$ die Voraussetzungen von a) und daher ist $\bigcup_i f^i(\bigcup_{\mathbf{N} \in FIX} \mathbf{N})$ ein Fixpunkt von f. Also gilt (*): $\bigcup_i f^i(\bigcup_{\mathbf{N} \in FIX} \mathbf{N}) \in FIX$. Weil definitionsgemäß gilt: $f^0(\bigcup_{\mathbf{N} \in FIX} \mathbf{N}) = \bigcup_{\mathbf{N} \in FIX} \mathbf{N}$ folgt:

$$\bigcup_{\mathbf{N} \in FIX} \mathbf{N} \subseteq \bigcup_i f^i(\bigcup_{\mathbf{N} \in FIX} \mathbf{N})$$

und wegen (*)

$$\bigcup_i f^i(\bigcup_{\mathbf{N}\in FIX} \mathbf{N}) \subseteq \bigcup_{\mathbf{N}\in FIX} \mathbf{N}.$$

Daraus folgt $\bigcup_i f^i(\bigcup_{\mathbf{N}\in FIX} \mathbf{N}) = \bigcup_{\mathbf{N}\in FIX} \mathbf{N}$. Also ist auch die Vereinigung aller Fixpunkte ein Fixpunkt von f und offensichtlich der größte: $FIX(f) = \bigcup_{\mathbf{N}\in FIX} \mathbf{N}$. □

Ströme und *stromverarbeitende Funktionen* bilden die semantische und konzeptuelle Grundlage für die in den folgenden Abschnitten definierten Sprachen. Ganz allgemein sind Ströme eines der wichtigsten Konzepte für die Modellierung verteilter Systeme. Dies gilt insbesondere dann, wenn die Kommunikation in solchen Systemen über den Austausch von Nachrichten erfolgt.

D sei eine höchstens abzählbare Menge von *Daten* oder *Nachrichten* und ⊥ ein Element, das nicht in **D** vorkommt. Der Bereich $\mathbf{D}^\omega$ der Ströme über **D** ist dann wie folgt definiert:

$$\mathbf{D}^\omega = \mathbf{D}^* \cup (\mathbf{D}^* \times \{\bot\}) \cup \mathbf{D}^\infty.$$

Dabei kann man $\mathbf{D}^*$ als die Menge aller endlichen Sequenzen (Worte) über **D** auffassen, $\mathbf{D}^\infty$ als die Menge aller unendlichen Sequenzen[1] und $(\mathbf{D}^* \times \{\bot\})$ als die Menge aller endlichen Sequenzen, die explizit durch das Anhängen von ⊥ abgeschlossen wurden. $\varepsilon \in \mathbf{D}^*$ bezeichnet die leere Sequenz und für $d \in \mathbf{D}$ sei ‹d› die einelementige Sequenz die nur d und sonst kein weiteres Element enthält.
$s_1 \hat{} s_2$ bezeichnet Konkatenation von Sequenzen. Für $s_1 \in \mathbf{D}^\infty$ und $s_2 \in \mathbf{D}^\omega$ möge dabei gelten: $s_1 \hat{} s_2 = s_1$. Beachte, daß D^ω unter der Konkatenation nicht abgeschlossen ist, denn es gilt:

$$s_1 \in (\mathbf{D}^* \times \{\bot\}) \wedge s_2 \in \mathbf{D}^\omega \Rightarrow s_1 \hat{} s_2 \notin \mathbf{D}^\omega.$$

Damit $\mathbf{D}^\omega$ zu einem Bereich wird, muß eine geeignete Ordnung festgelegt werden. Wir definieren daher für $s_1, s_2 \in \mathbf{D}^\omega$:

$$s_1 \sqsubseteq s_2 \Leftrightarrow s_1 = s_2 \vee \exists s_3 \in \mathbf{D}^*, s_4 \in \mathbf{D}^\omega: s_1 = s_3 \hat{} \langle\bot\rangle \wedge s_2 = s_3 \hat{} s_4.$$

Demnach ist $\mathbf{D}^* \cup (\mathbf{D}^* \times \{\bot\})$ die Menge der endlichen Elemente aus $\mathbf{D}^\omega$ und $\mathbf{D}^\infty$ die Menge der unendlichen. Die Ströme aus $\mathbf{D}^* \cup \mathbf{D}^\infty$ sind total und die aus $(\mathbf{D}^* \times \{\bot\})$ partiell. ‹⊥› ist das kleinste Element aus $\mathbf{D}^\omega$. Statt ‹⊥› schreiben wir auch häufig einfach ⊥.

Ein Strom repräsentiert die vollständige *Kommunikationsgeschichte* eines *Kanals*, über den ein Agent A mit einem Agenten B kommuniziert. Entsprechend der definitorischen Zerlegung von $\mathbf{D}^\omega$ in disjunkte Teilmengen lassen sich drei Fälle unterscheiden:

---

[1] $\mathbf{D}^\infty$ kann auch als Menge der totalen Funktionen von **Nat** nach **D** interpretiert werden.

- A sendet B eine endliche Anzahl von Nachrichten und divergiert dann, ohne daß B das Ende der Übertragung erkennen kann. Der zugehörige Strom ist ein Element aus $\mathbf{D}^* \times \{\bot\}$.
- A sendet B eine endliche Anzahl von Nachrichten und terminiert dann "ordnungsgemäß". B kann dies feststellen und seine weiteren Aktionen darauf einrichten, z.B. nicht mehr unnötig auf weitere Nachrichten von A warten. Der zugehörige Strom ist ein Element aus $\mathbf{D}^*$.
- A sendet B eine unendliche Anzahl von Nachrichten. Für potentiell nichtterminierende Programme ist das eine durchaus realistische Option. Der zugehörige Strom ist ein Element aus $\mathbf{D}^\infty$

Auf dem Bereich der Ströme sind einige Grundfunktionen definiert:

.&.: $\mathbf{D}^\bot \times \mathbf{D}^\omega \to \mathbf{D}^\omega$,
ft: $\mathbf{D}^\omega \to \mathbf{D}^\bot$,
rt: $\mathbf{D}^\omega \to \mathbf{D}^\omega$,
isempty: $\mathbf{D}^\omega \to \mathbf{Bool}^\bot$.

Sie sind durch folgende Axiome festgelegt ($d \in \mathbf{D}$, $s \in \mathbf{D}^\omega$):

$$d\&s = \langle d \rangle \hat{} s, \qquad \bot\&s = \bot,$$

sowie

$$ft(d\&s) = d, \qquad rt(d\&s) = s,$$
$$isempty(d\&s) = false,$$
$$ft(\varepsilon) = \bot, \qquad rt(\varepsilon) = \bot,$$
$$isempty(\varepsilon) = true,$$
$$ft(\bot) = \bot, \qquad rt(\bot) = \bot,$$
$$isempty(\bot) = \bot.$$

Die Präfixoperation & dient dazu, einem Strom s ein Element d voranzustellen. Sie ist strikt in ihrem ersten und nicht-strikt in ihrem zweiten Argument. Dies ist konsistent mit der Kommunikationsinterpretation von Strömen und der Intuition, die sich mit $\bot$ verbindet: Wenn d&s eine Sequenz kommunizierter Daten repräsentiert und d selber das Ergebnis eines Berechnungsprozesses ist, dann bedeutet $d = \bot$, daß der sendende Agent bei der Berechnung der ersten zu übertragenden Nachricht divergiert. Danach kann aber keine weitere Nachricht mehr übertragen werden, der gesamte Strom ist daher $\bot$. Es gilt das folgende einfache, aber wichtige Lemma:

Lemma 2.4: &, ft, rt und isempty sind monoton und stetig. □

Eine *stromverarbeitende Funktion* oder kurz *Stromfunktion* $f: (\mathbf{D}^\omega)^m \to (\mathbf{D}^\omega)^n$ ist eine Abbildung, die ein m-Tupel von Eingabeströmen auf ein n-Tupel von Ausgabeströmen abbildet[1]. Wenn f monoton und stetig ist, d.h. $f \in [(\mathbf{D}^\omega)^m \to (\mathbf{D}^\omega)^n]$, dann kann man es als Beschreibung eines kommunizierenden *Agenten* ansehen, der über m Eingabekanäle von seiner Umgebung mit Nachrichten versorgt wird und über n Ausgabekanäle Nachrichten an sie zurücksendet.
Monotonie und Stetigkeit sind dabei bezüglich der $\sqsubseteq$-Ordnung auf D^ω definiert. Neben der formalen haben sie hier eine intuitiv-operationelle Interpretation:

- Monotonie beschreibt die Eigenschaft, daß Agenten ausgegebene Nachrichten nicht mehr manipulieren können. Verlängert man den Strom auf einem ihrer Eingabekanäle, so reagieren sie, wenn überhaupt, durch Ausgabe zusätzlicher Nachrichten. Bereits ausgegebene Nachrichten werden nicht verändert. Monotonie ist darüber hinaus Voraussetzung für das parallele Arbeiten mehrerer Agenten: Kein Agent braucht seine Eingabe vollständig zu kennen, jeder kann seine Berechnungen schon beginnen, wenn nur ein Teilstück der Eingabe vorliegt. Entsprechend sind Teilstücke der Eingabe auch nur für Teilstücke der Ausgabe verantwortlich. In diesem Sinn reflektiert Monotonie den kausalen Zusammenhang zwischen Ein- und Ausgaben.
- Stetigkeit impliziert, daß für die Erzeugung eines endlichen Ausgabestroms auch nur ein endlicher Eingabestrom notwendig ist. Jede einzelne Ausgabenachricht ist damit Reaktion auf höchstens endlich viele Eingabenachrichten.

$[(\mathbf{D}^\omega)^m \to (\mathbf{D}^\omega)^n]$ ist die Menge aller stetigen, (m,n)-stelligen stromverarbeitenden Funktionen.
Die Ordnung auf $[(\mathbf{D}^\omega)^m \to (\mathbf{D}^\omega)^n]$ ist wie üblich punktweise definiert.
Seien $f,g \in [(\mathbf{D}^\omega)^m \to (\mathbf{D}^\omega)^n]$. f heißt *partiell korrekt* bzgl. g, wenn gilt:

$$f \sqsubseteq g.$$

Jede von f erzeugte Ausgabe ist dann konsistent mit den von g erzeugten Ausgaben. Eventuell divergiert f aber früher als g.
Umgekehrt heißt g *robust korrekt* bzgl. f: g liefert eventuell auch dann noch Nachrichten (z.B. Fehlermeldungen), wenn f bereits divergiert hat. f heißt *total korrekt* bzgl. g, wenn f = g ist.

Die Sätze von Knaster/Tarski und Kleene gelten natürlich auch für monotone bzw. stetige Stromfunktionen. Für die Zwecke der vorliegenden Arbeit sind folgende Verallgemeinerungen wichtig:

Satz 2.5 (Fixpunkte von Gleichungssystemen): Seien $f_1, f_2, \ldots, f_n \in [(\mathbf{D}^\omega)^n \to \mathbf{D}^\omega]$ und

---

[1] Später bezeichnen wir auch Abbildungen der Art $f: (\mathbf{D}^\perp)^p \times (\mathbf{D}^\omega)^q \to (\mathbf{D}^\omega)^r$ als stromverarbeitende Funktionen. Die zusätzliche Parametrisierung erhöht die Flexibilität beim Programmieren.

$$G = \begin{pmatrix} s_1 = f_1(s_1, \ldots, s_n) \\ \ldots \\ s_n = f_n(s_1, \ldots, s_n) \end{pmatrix}$$

ein System wechselseitig rekursiver Gleichungen. Dann besitzt G eine (bzgl. $\sqsubseteq$) kleinste Lösung $(s_1^*, \ldots, s_n^*)$, für die gilt:

$$(s_1^*, \ldots, s_n^*) = \bigsqcup_i (s_1^i, \ldots, s_n^i),$$

wobei $(s_1^0, \ldots, s_n^0) = (\bot, \ldots, \bot)$ und $(s_1^{i+1}, \ldots, s_n^{i+1}) = (f_1(s_1^i,\ldots,s_n^i), \ldots, f_n(s_1^i,\ldots,s_n^i))$. $(s_1^*, \ldots, s_n^*)$ heißt der *kleinste Fixpunkt* von G wird wie oben durch fix(G) bezeichnet. □

Gleichungssysteme können zusätzlich parametrisiert sein. Dann hängt der Fixpunkt monoton und stetig von den Parametern ab.

Satz 2.6 (Stetige Parameterabhängigkeit): Seien $f_1, f_2, \ldots, f_n \in [\mathbf{D} \times (\mathbf{D}^\omega)^n \to \mathbf{D}^\omega]$ und $(d_i \mid i \in \mathbf{Nat})$ sei eine Kette in **D** mit Supremum d^*. Für $i \in \mathbf{Nat}$ sei

$$G_i = \begin{pmatrix} s_1 = f_1(d_i, s_1, \ldots, s_n) \\ \ldots \\ s_n = f_n(d_i, s_1, \ldots, s_n) \end{pmatrix}$$

und G^* analog für d^*.
Dann ist $(\mathrm{fix}(G_i) \mid i \in \mathbf{Nat})$ eine Kette in $(\mathbf{D}^\omega)^n$ für die gilt: $\bigsqcup_i \mathrm{fix}(G_i) = \mathrm{fix}(G^*)$. □

Jedes (parametrisierte) Gleichungssystem der obigen Form bestimmt daher eine stetige Stromfunktion aus $[\mathbf{D} \to (\mathbf{D}^\omega)^n]$. Anders ausgedrückt: Stetige Stromfunktionen können mit Hilfe von Gleichungssystemen definiert werden. Dabei sind sogar rekursive Definitionen erlaubt. Dies ergibt sich aus folgendem Satz.

Satz 2.7 (Stetige Funktionsabhängigkeit): Für $1 \le j \le n$ seien $(f_{ji} \mid i \in \mathbf{Nat})$ Ketten in $[(\mathbf{D}^\omega)^n \to \mathbf{D}^\omega]$ mit den Suprema f_j^*. Wie oben sei

$$G_i = \begin{pmatrix} s_1 = f_{1i}(s_1, \ldots, s_n) \\ \ldots \\ s_n = f_{ni}(s_1, \ldots, s_n) \end{pmatrix}$$

und G^* analog für die f_j^*.
Dann ist wiederum $(\mathrm{fix}(G_i) \mid i \in \mathbf{Nat})$ eine Kette in $(\mathbf{D}^\omega)^n$, für die gilt: $\bigsqcup_i \mathrm{fix}(G_i) = \mathrm{fix}(G^*)$. □

Satz 2.7 transformiert die Aussage von Satz 2.6 auf die nächsthöhere Ebene: Jedes Gleichungssystem definiert nicht nur eine stetige Stromfunktion sondern auch ein stetiges

Funktional. Dadurch wird es möglich, eine Funktion f durch ein Gleichungssystem zu definieren, in dem f angewandt auftritt.

Auf die Möglichkeit stromverarbeitende Funktionen und Gleichungssysteme zur Modellierung verteilter Systeme einzusetzen, hat Kahn in seiner grundlegenden Arbeit [Kahn 74] hingewiesen. Zahlreiche andere Autoren haben dies aufgegriffen und in vielfacher Hinsicht erweitert (etwa in Bezug auf Nichtdeterminismus, vgl. z.B. [Keller 78], [Brock, Ackerman 81], [Park 82], [Broy 87b] für einen Überblick siehe [Løvengreen 85]). Die methodische Handhabbarkeit und formale Eleganz des Ansatzes beruht wesentlich auf den in diesem Abschnitt rekapitulierten Resultaten.

3. Die applikative Sprache AL

Die im folgenden beschriebene applikative Sprache AL ist an die von Broy in [Broy 86] entwickelte Sprache AMPL ("an applicative multiprogramming language") angelehnt. Wie diese bietet sie die Möglichkeit, wechselseitig rekursive Funktionen und Systeme wechselseitig rekursiver Stromgleichungen zu definieren. Im Unterschied zu AMPL ist AL jedoch typisiert und erlaubt zudem den hierarchischen Aufbau von Agentennetzen.

3.1 Syntax

Die AL-Syntax ist in einer BNF-artigen Notation beschrieben. Dabei steht {X | Y} für die Auswahl zwischen X und Y, wobei Klammern soweit möglich weggelassen werden. X^* bezeichnet eine endliche, eventuell auch leere Liste, deren Elemente durch Kommata getrennt sind, d.h.: X^* ::= <> | X | X,X | … . Analog steht X^+ für eine nicht-leere Liste derselben Art. Folgen zwei Listen X^*, Y^* direkt aufeinander, so wird das trennende Komma nur dann aufgeführt, wenn keine der Listen leer ist, also: $X^*, Y^* ::= X^* | Y^* | X^+, Y^+$. Schlüsselworte sind fett gedruckt, Nichtterminalsymbole durch ‹ › eingeschlossen.

Die Syntaxbeschreibung stützt sich auf drei paarweise disjunkte Identifikatormengen:

FID:	Identifikatoren für *Funktionen* und *Agenten*.
OID:	Identifikatoren für *Objekte*.
SID:	Identifikatoren für *Ströme*.

ID = OID ∪ SID ist die Menge aller Identifikatoren für *Werte*. Zur Unterscheidung von semantischen Objekten sind die syntaktischen Bezeichner im folgenden stets *kursiv* gedruckt: $s \in$ SID ist also beispielsweise ein Strombezeichner, während s ein Strom ist.

In der anschließenden Grammatik seien: ‹prg_id›, ‹agt_id›, ‹fct_id› ∈ FID, ‹obj_id› ∈ OID und ‹str_id› ∈ SID.

‹program› ::= **program** ‹prg_id› ≡ ‹stream›* → ‹stream›+:
{‹agent› | ‹function›}*
‹eq_sys›
end

‹agent› ::= **agent** ‹agt_id› ≡ ‹object›*, ‹stream›* → ‹stream›+:
‹eq_sys›
end

‹function› ::= **funct** ‹fct_id› ≡ ‹object›* → ‹sort›:
‹exp›
end

‹stream› ::= **chan** ‹sort› ‹str_id›+

‹object› ::= ‹sort› ‹obj_id›+

‹eq_sys› ::= ‹equation›+

‹equation› ::= ‹str_id›+ ≡ ‹exp›

‹exp› ::= ε | ⊥ | ‹obj_id› | ‹str_id› | ‹primitive object› |
‹exp›&‹exp› | {**ft** | **rt** | **isempty**}.‹exp› |
‹exp›▯‹exp› | **if** ‹exp› **then** ‹exp› **else** ‹exp› **fi** |
{‹agt_id› | ‹fct_id› | ‹primitive function›} (‹exp›*)

AL ist eine typisierte Sprache. Jeder *Sorte* (jedem *Typ*) **u** entspricht eine Menge **U** geeigneter semantischer Werte. Wie in Kapitel 2 definiert, bezeichnet $\mathbf{U}^{\perp}$ den zugehörigen flache Bereich über **U**. Im Rahmen dieser Arbeit werden alle benötigten Sorten als gegeben vorausgesetzt. Dies gilt insbesondere für **nat**, die Sorte der natürlichen Zahlen **Nat** = {0, 1, 2, …}, und **bool**, die Sorte der booleschen Werte **Bool** = {true, false}. Komplexe und strukturierte Sorten wie **sequ nat**

oder **stack bool** können mit Hilfe algebraischer Techniken knapp und präzise beschrieben werden (vgl. [Bauer, Wössner 81]). In diesem Sinne bezeichnen die verwendeten Sorten die Trägermengen der Modelle abstrakter Datentypen. Die Definition der Datentypen selber ist jedoch kein Bestandteil der Sprache und erfolgt wenn nötig auf einer Metaebene.

Die Vereinigung (präzise, die bereichstheoretische Summe) aller Wertemengen bildet das flach geordnete Universum $\mathbf{Dom}^{\perp}$ der *Objekte*. Jedes Element aus $\mathbf{Dom}^{\perp}$ ist als *primitives Objekt* in AL verfügbar. Dies führt zu Programmtermen, in denen neben syntaktischen auch semantische Elemente vertreten sind, allerdings wohlunterschieden.
Konsistent mit dem Datentypansatz, der die enge Verbindung zwischen Daten und den zugehörigen Operationen betont, kann darüber hinaus auf *primitive Funktionen* zurückgegriffen werden. Eine primitive Funktion ist eine strikte Abbildung zwischen den jeweiligen semantischen Bereichen.

In der Kopfleiste von Programm- bzw. Agentendefinitionen kann einer Sorte das Schlüsselwort **chan** vorangestellt werden. **chan u** steht für den Bereich $\mathbf{U}^{\omega}$ der Ströme über **U**. Wir nennen $\mathbf{U}^{\perp}$ einen *Objektbereich* und $\mathbf{U}^{\omega}$ einen *Strombereich*. Die Namensgebung – **chan** anstelle etwa von **str** – hat ihren Grund in der (syntaktischen) Interpretation eines Bezeichners *i* vom Typ **chan u** als Kommunikationskanal zwischen zwei parallel arbeitenden Agenten. Die Semantik von AL ordnet einem Kanal *i* den endlichen oder unendlichen Strom i aller Nachrichten zu, die während des Programmablaufes über *i* gesandt wurden. i repräsentiert damit die vollständige Kommunikationsgeschichte (engl. "communication history") von *i*. In der Literatur ist aus diesem Grund auch der Begriff "History"-Semantik gebräuchlich (vgl. z.B. [Brock, Ackerman 81], [Jonsson 89]). Wie für variablenfreie, applikative Sprachen üblich, steht in AL die semantische Stromauffassung im Vordergrund. Wir sprechen daher des weiteren auch meistens von Strömen bzw. Strombezeichnern und nicht von Kanälen. In der prozeduralen Sprache PL ist die Unterscheidung zwischen dem "Behältnis" – dem Kanal – und seinem akkumulierten Inhalt – dem Strom aller kommunizierten Daten – stärker ausgeprägt. Um die Verwandtschaft zwischen diesen Auffassungen trotz aller Unterschiede (vgl. Abschnitt 4.3) zu betonen und die syntaktischen Oberflächen der Sprachen aneinander anzunähern, wird auf beiden Ebenen dasselbe Schlüsselwort verwandt.

$\mathbf{Dom}^{\omega}$ bezeichnet die Vereinigung aller Strombereiche mit kleinstem Element ‹⊥›. **ft**, **rt**, **isempty** und & sind die eingebauten Operatoren zur Strommanipulation. Sie beziehen sich direkt auf die entsprechenden semantischen Funktionen auf $\mathbf{Dom}^{\omega}$ (vgl. Kapitel 2). Wichtig ist, daß ein AL-Agent die Identität zweier Ströme nur mit Hilfe von **isempty** überprüfen kann und zwar aus folgendem Grund: Die boolesche Gleichheit ist als primitive Operation nur auf Objektbereichen, nicht jedoch für Ströme verfügbar. Es ist daher syntaktisch unzulässig, einen Ausdruck der Art *s* = *t* für *s*,*t*∈ SID zu bilden. Die zugehörigen Ströme s,t∈ $\mathbf{Dom}^{\omega}$ können nur elementweise auf Gleichheit überprüft werden. Da es keine Möglichkeit gibt, den undefinierten Strom ‹⊥› im Programmablauf zu erkennen, ist ein solcher Test nur dann in endlicher Zeit ausführbar, wenn sowohl s als auch t endlich und total sind, d.h. s,t∈ $\mathbf{Dom}^{*} \subseteq \mathbf{Dom}^{\omega}$.

Die Syntax der Sprache unterscheidet zwischen Funktions- und Agentendeklarationen. *Funktionsdeklarationen* werden auf der semantischen Ebene durch (Mengen von) Abbildungen zwischen Objektbereichen gedeutet und mit Hilfe von reinen Objektausdrücken gebildet:
Ein Ausdruck E∈ ‹exp› heißt *Objektausdruck*, falls er bei Auswertung ein Objekt aus $\mathbf{Dom}^{\perp}$ liefert und *Stromausdruck* sonst. E heißt *reiner Objektausdruck*, falls die Stromoperatoren **ft** und **isempty**, die Ströme auf Objekte abbilden, nicht in ihm vorkommen.

Alle für applikative Sprachen typischen Funktionsdeklarationen sind mit diesen Mitteln formulierbar. Allerdings ist AL keine Sprache höhere Ordnung, so daß Funktionen nicht als Parameter übergeben werden können.

Beispiel 3.1 (Fakultät):

funct $fac \equiv$ **nat** $n \rightarrow$ **nat**:
if $n = 0$ **then** 1 **else** $n * fac(n - 1)$ **fi**
end
□

Agenten bzw. Agentendeklarationen werden durch (Mengen von) stromverarbeitende(n) Funktionen gedeutet. Ihre Eingabe besteht aus Objekten und/oder Strömen, ihre Ausgabe jedoch nur aus Strömen. Mehrstellige Resultate sind möglich. Der Rumpf eines Agenten ist ein Gleichungssystem, das ebenso aufgebaut ist, wie der Gleichungsteil vollständiger Programme.

Beispiel 3.2 (Stromaddition):

agent $add^* \equiv$ **chan nat** $i, j \rightarrow$ **chan nat** o:
$o \equiv$ **if** $\neg$**isempty.**$i \wedge \neg$**isempty.**j
then **ft.**i + **ft.**j & add^*(**rt.**i, **rt.**j)
else ε **fi**
end

add^* ist die punktweise Erweiterung der Addition auf Ströme natürlicher Zahlen. *sum* ist auf add^* abgestützt:

agent $sum \equiv$ **chan nat** $i \rightarrow$ **chan nat** o:
$o \equiv add^*(i, s)$,
$s \equiv 0 \,\&\, o$
end

Das n-te Element des Ausgabestroms von *sum* ist die Summe der ersten n Elemente des Eingabestroms i. Intern benutzt *sum* den rekursiv definierten Strom s.
□

Als nicht-numerisches Beispiel möge die folgende *interaktive Schlange* (vgl. [Broy 88a]) dienen.

Beispiel 3.3 (Interaktive Speicherschlange): Sei **data** ein beliebiger Typ von Basisdaten und **queue data** der zugehörige Typ aller Schlangen über **data** mit den dafür üblichen Operationen:

eq: $\rightarrow$ **queue data**,
iseq: **queue data** $\rightarrow$ **bool**,
stock: **queue data** $\times$ **data** $\rightarrow$ **queue data**,
head: **queue data** $\rightarrow$ **data**,
tail: **queue data** $\rightarrow$ **queue data**.

Eine *interaktive Speicherschlange* ist ein Modul in einem verteilten System, das seiner Umgebung diese Datenstruktur in verkapselter Form zur Verfügung stellt. Die Umgebung kommuniziert mit der Schlange durch Nachrichten; sie sendet ihr Daten, die die Schlange speichert, oder die Aufforderung, gespeicherte Daten wieder auszugeben. Wir definieren:

imsg = **data** $\cup$ {?},
omsg = **data** $\cup$ {error}.

Der folgende Agent beschreibt eine interaktive Schlange:

agent *interactive_queue* $\equiv$ **chan imsg** $i \rightarrow$ **chan omsg** o:
$o \equiv$ *hq*(eq, i)
end

Er stützt sich auf die Hilfsfunktion *hq*:

agent *hq* $\equiv$ **queue data** s, **chan imsg** $i \rightarrow$ **chan omsg** o:
$o \equiv$ **if ft.**i = ? $\wedge$ iseq(s)
then error & *hq*(s, **rt.**i)
else if ft.i = ? **then** head(s) & *hq*(tail(s), **rt.**i)
else *hq*(stock(s, **ft.**i), **rt.**i) **fi fi**
end

Die Operationen des Datentyps **queue data** werden dabei als primitive Funktionen benutzt. Man beachte die Verwandtschaft zwischen ihnen und den Stromoperationen. Der Datentyp **queue** wird auch in Abschnitt 5.3.1 noch eine Rolle spielen. □

Ein vollständiges AL-Programm besteht aus einer Menge von Funktions- und Agentendeklarationen und einem System von Gleichungen. Wie bei Agenten werden in einer Kopfzeile Eingabe- und Ausgabeströme angeführt. Über diese Ströme kann das Programm von einer eventuell vorhandenen Umgebung mit Eingaben versorgt werden und Resultate an diese zurückreichen.

Beispiel 3.4 (Sieb des Erathostenes):

program *erathostenes* ≡ → **chan nat** s_3:

agent add^* ≡ **chan nat** *i*, *j* → **chan nat** *o*:

o ≡ **if** ¬**isempty.***i* ∧ ¬**isempty.***j*
then **ft.***i* + **ft.***j* & add^*(**rt.***i*, **rt.***j*)
else ε **fi**
end,

agent *sum* ≡ **chan nat** *i* → **chan nat** *o*:

o ≡ add^*(*i*, *s*),
s ≡ 0 & *o*
end,

agent *filter* ≡ **nat** *n*, **chan nat** *i* → **chan nat** *o*:

o ≡ **if** **ft.***i* mod *n* = 0 **then** *filter*(*n*, **rt.***i*) **else** **ft.***i* & *filter*(*n*, **rt.***i*) **fi**
end,

agent *sieve* ≡ **chan nat** *i* → **chan nat** *o*:

o ≡ **ft.***i* & *sieve*(*s*)
s ≡ *filter*(**ft.***i*, **rt.***i*)
end,

s_1 ≡ 1 & s_1,
s_2 ≡ *sum*(s_1),
s_3 ≡ *sieve*(**rt.***s_2*)

end

Die erste Gleichung definiert den unendlichen Strom 1^∞. s_2 ist dann der Strom aller natürlichen Zahlen 1 2 3… und s_3 der Strom aller Primzahlen. *sieve* erzeugt für jede Primzahl p ≥ 2 eine Instanz des Agenten *filter*, der alle Vielfachen von p aus seinem Eingabestrom herauslöscht. Da alle Zahlen größer als p diesen Filter passieren müssen, realisiert das Programm eine parallele Version des bekannten Sieb des Erathostenes (vgl. [Kahn, MacQueen 77]).

□

Um eine sinnvolle Bedeutung zu haben, müssen AL-Programme einer Reihe von *Kontext-bedingungen* genügen. Dazu gehört als erstes die Forderung nach sortenkorrekter (typkorrekter) Ausdrucksbildung. *Sortenkorrektheit* ist dann gewährleistet, wenn an allen syntaktischen Positionen, an denen (Teil-)Ausdrücke bestimmter Sorte erwartet werden, auch nur solche auftreten. Da Agenten mehrstellige Ausgaben aufweisen können, muß darüber hinaus die *Stelligkeit* von Ausdrücken besonders beachtet werden. Ein- und Ausgaben von Agenten oder Funktionen lassen sich kanonisch als Tupel von Werten auffassen. Analog stehen Ausdrücke für Wertetupel. Ein solches Tupel (x_1, …, x_n) ist ein Element des kartesischen Produkts $X_1 \times \ldots \times$

X_n, wobei $X_i = \mathbf{U}^{\perp}$ oder $X_i = \mathbf{U}^{\omega}$. Da der Produktoperator annahmegemäß assoziativ ist (vgl. Kapitel 2), gibt es keine verschachtelten Tupel.

Für ein vollständiges AL-Programm existieren grundsätzlich zwei Arten von *Bindungsbereichen* (engl. "scopes"): Einerseits der Bereich, der durch das Programm selber aufgespannt wird und andererseits die Bereiche, die den einzelnen Funktions- und Agentendeklarationen zugeordnet sind. Im Bindungsbereich des Programms werden die Funktionalitäten aller verwendeten Funktionen und Agenten festgelegt, sowie die Typbindung aller Strombezeichner, die im Gleichungsteil des Programms auftreten, vorgenommen. Alle in den Funktions- bzw. Agentenrümpfen vorkommenden Strom- und Objektbezeichner sind lokal. Bezogen auf den jeweiligen Bindungsbereich kann die Sortenzugehörigkeit und Stelligkeit jedes Ausdrucks bestimmt werden.
Auf Typisierungs- und Stelligkeitsaspekte soll hier nicht im Detail eingegangen werden. Es ist jedoch wichtig, daß die eingebauten Funktionen **ft**, **rt**, **isempty** und & nur auf einstellige Ausdrücke angewendet werden dürfen und daß für Ausdrücke der Art **if** C **then** E_1 **else** E_2 **fi** oder $E_1 \square E_2$ gelten muß, daß E_1 und E_2 die gleiche Stelligkeit besitzen und (komponentenweise) der gleichen Sorte angehören. Entsprechende Regeln gelten für Aufrufe $f(E_1, \ldots, E_n)$. Wir gehen im folgenden davon aus, daß alle Ausdrücke typ- und stelligkeitskorrekt gebildet sind.

Sei *prg* ein vollständiges AL-Programm

program *prg* ≡ **chan** $\mathbf{v_1}$ i_1, …, **chan** $\mathbf{v_n}$ i_n → **chan** $\mathbf{w_1}$ o_1, …, **chan** $\mathbf{w_m}$ o_m:
DK,
GS
end,

dann heißen die $i_1, \ldots, i_n$ *Eingabeströme* und die $o_1, \ldots, o_m$ *Ausgabeströme* von *prg*. Ein Strom *s* (genauer, ein Strombezeichner *s*), der in GS auftritt, jedoch in der Kopfleiste nicht angeführt ist, heißt *intern*. Die Sortenzugehörigkeit von internen Strömen ergibt sich aus dem Kontext; sie muß für jedes *s* eindeutig bestimmt sein. *prg* muß darüber hinaus die folgenden, allgemein üblichen Bedingungen erfüllen:

- Alle Ein- und Ausgabeströme von *prg* sind paarweise verschieden bezeichnet.
- Alle im Deklarationsteil DK aufgeführten Agenten und Funktionen sind paarweise verschieden bezeichnet.
- Alle in *prg* verwendeten Agenten und Funktionen sind in DK deklariert.

Weiterhin muß für das Gleichungssystem GS gelten:

- Jeder interne Strom *s* und jeder Ausgabestrom *o* tritt genau einmal auf der linken Seite einer Gleichung auf. Dies ist die *definierende Gleichung* für *s* bzw. *o*. Auf rechten Seiten können interne Ströme und Ausgabeströme beliebig häufig vorkommen, insbesondere auch auf den rechten Seiten ihrer definierenden Gleichungen. Eingabeströme dürfen intern nicht redefiniert werden, sie treten ausschließlich auf rechten Seiten auf.

- Für jede Gleichung $s_1, \ldots, s_n \equiv S$ aus GS gilt: S ist ein n-stelliger Ausdruck und die Sorte von s_i stimmt mit der i-ten Komponentensorte von S überein.

Für Agenten und Funktionen gelten analoge Bedingungen. Um die Lesbarkeit der zum großen Teil schematischen Ausführungen in den kommenden Abschnitten zu erhöhen, werden Funktionen im folgenden stets einstellig, Agenten mit höchstens einem Objekt und/oder einem Strom als Eingabe- und genau einem Strom als Ausgabe notiert.

3.2 Denotationelle Semantik

Die Semantik von AL wird durch zwei Abbildungen festgelegt, die den Elementen aus den syntaktischen Kategorien der Sprache mathematische Objekte zuordnen. Wie für denotationelle Ansätze üblich, erfolgt die Definition dieser Abbildungen induktiv über den Aufbau der Sprachkonstrukte: Ist K ein aus $K_1, \ldots, K_n$ zusammengesetztes Konstrukt, so ist $\mathbf{Sem}[\![K]\!]$ eine Funktion von $\mathbf{Sem}[\![K_1]\!], \ldots, \mathbf{Sem}[\![K_n]\!]$. Semantiken, die auf diesem Definitionsprinzip beruhen heißen *synthetisch*. Es ist für sie charakteristisch, daß die Bedeutung komplexer syntaktischer Einheiten aus der Bedeutung ihrer (einfacheren) Bestandteile abgeleitet wird. Im Fall rekursiver Definitionen kommen dabei *Fixpunkttechniken* zum Einsatz.

Synthetische Semantiken gewährleisten *Kompositionalität*. Eine Semantik heißt *kompositional* wenn für alle syntaktischen Konstrukte K, K' und für alle passenden Kontexte cn[.] gilt:

$$\mathbf{Sem}[\![K]\!] = \mathbf{Sem}[\![K']\!] \Rightarrow \mathbf{Sem}[\![cn[K]]\!] = \mathbf{Sem}[\![cn[K']]\!].$$

Ein passender Kontext ist dabei ein Konstrukt mit einer ausgezeichneten Stelle ("Loch"), in das K und K' so eingefügt werden können, daß ein wohlgeformtes Programm(fragment) entsteht. Kompositionalität ist für transformationelle Systementwicklungen von entscheidender Wichtigkeit. Ist eine Semantik kompositional, so kann ein Programmstück K durch ein semantisch äquivalentes Stück K' ersetzt werden, ohne das sich die Bedeutung des vollständigen Programms ändert. Ist die Semantik nicht kompositional, so muß die globale Korrektheit lokaler Transformationen stets durch zusätzliche Beweisschritte nachgewiesen werden.

Im Rahmen einer Methodik, die auf die systematische Entwicklung von Programmen ausgerichtet ist, werden neben Äquivalenzumformungen auch *Implementierungsschritte* ausgeführt. Die Semantik sollte daher nicht nur kompositional sein, sondern zusätzlich *monoton* bezüglich einer geeignet definierten *Implementierungsrelation*:

Sei ‹‹ eine Relation auf den semantischen Bereichen. Ein Konstrukt K *implementiert* ein Konstrukt K', falls gilt: $\mathbf{Sem}[\![K]\!] \mathrel{‹‹} \mathbf{Sem}[\![K']\!]$. Die Semantik heißt monoton bezüglich ‹‹, falls für alle syntaktischen Konstrukte K, K' und für alle passenden Kontexte cn[.] gilt:

$$\mathbf{Sem}[\![K]\!] \mathrel{‹‹} \mathbf{Sem}[\![K']\!] \Rightarrow \mathbf{Sem}[\![cn[K]]\!] \mathrel{‹‹} \mathbf{Sem}[\![cn[K']]\!].$$

Diese Eigenschaft ermöglicht dann lokale Implementierungsschritte.

Konkret werden für AL in den folgenden Abschnitten zwei semantische Funktionen **B** und **F** definiert. **B** ist mengenorientiert (vgl. [CIP 85]). Sie ordnet jedem (mehrdeutigen) Ausdruck die Menge seiner potentiellen Auswertungsergebnisse zu. Die funktionale Semantik **F** baut auf **B** auf. Sie hat insofern funktionalen Charakter, als daß sie

- jedem Ausdruck und jedem Gleichungssystem eine Menge stetiger Funktionen,
- jeder Funktionsdeklaration eine Menge strikter Funktionen über den flachen Grundbereichen und
- jeder Agentendeklaration bzw. jedem vollständigen Programm eine Menge stetiger Funktionen über den präfixgeordneten Strombereichen

zuordnet. **B** wird im folgenden ausschließlich für Ausdrücke und **F** daran anschließend für Ausdrücke, Gleichungssysteme, Funktionen, Agenten und vollständige Programme in genau dieser Reihenfolge definiert. Dazu sind einige vorbereitende Begriffsbildungen notwendig.

FCT sei der Bereich der stetigen Funktionen über dem flach geordneten Universum der atomaren Objekte. **AGT** sei der Bereich der stromverarbeitenden Funktionen:

$$\mathbf{FCT} = \{f \in [(\mathbf{Dom}^{\perp})^p \to \mathbf{Dom}^{\perp}] \mid p \in \mathbf{Nat}\},$$
$$\mathbf{AGT} = \{f \in [(\mathbf{Dom}^{\perp})^p \times (\mathbf{Dom}^{\omega})^q \to (\mathbf{Dom}^{\omega})^r] \mid p,q,r \in \mathbf{Nat},\ r \neq 0\}.$$

Objektparameter sollen an AL-Funktionen und Agenten call-by-value übergeben werden. Für $f \in \mathbf{FCT}$ und $g \in \mathbf{AGT}$ bezeichne daher "strict f" das in allen Parametern strikte Gegenstück zu f und "strict g" das in allen *Objektparametern* strikte Gegenstück zu g (vgl. Kapitel 2). $\mathbf{FCT}_S$ sei der Bereich aller strikten Funktionen und $\mathbf{AGT}_S$ der Bereich aller in den Objektparametern strikten stromverarbeitenden Funktionen. **MAP** sei die Menge aller stetigen Funktionen über den vereinigten Grundbereichen:

$$\mathbf{MAP} = \{f \in [(\mathbf{Dom}^{\perp} \cup \mathbf{Dom}^{\omega})^p \to (\mathbf{Dom}^{\perp} \cup \mathbf{Dom}^{\omega})^q] \mid p,q \in \mathbf{Nat},\ q \neq 0\}.$$

Die Definitionen von **B** und **F** stützten sich auf die Begriffe *Umgebung* und *Zustand*[1] . Eine Umgebung ordnet jedem Funktionssymbol $f \in$ FID eine nichtleere Menge von (stromverarbeitenden) Funktionen zu:

$$\mathrm{ENV} = \mathrm{FID} \to \wp(\mathbf{FCT}_S)\backslash\emptyset \cup \wp(\mathbf{AGT}_S)\backslash\emptyset.$$

Ein Zustand liefert zu jedem $x \in$ OID ein Objekt aus $\mathbf{Dom}^{\perp}$ und zu jedem $s \in$ SID einen Strom aus $\mathbf{Dom}^{\omega}$:

---

[1] Im Zusammenhang mit funktionalen Sprachen spricht man anstelle von *Zuständen* meist von *Belegungen*. Um eine für AL und PL einheitliche Sprechweise zu haben, verwenden wir in dieser Arbeit jedoch in beiden Fällen die Bezeichnung *Zustand*.

$$\text{STATE} = \text{ID} \rightarrow \mathbf{Dom}^{\perp} \cup \mathbf{Dom}^{\omega}.$$

Komponentenweises Ändern von Zuständen σ ist wie üblich definiert:

$$\sigma[e/x](y) = \begin{cases} e & \text{falls } x = y \\ \sigma(y) & \text{sonst} \end{cases},$$

dabei sind x, y Bezeichner und e ein passender semantischer Wert. Anstelle von $\sigma[e_1/x_1] \ldots [e_n/x_n]$ schreiben wir auch $\sigma[e_1/x_1, \ldots, e_n/x_n]$ oder $\sigma[e_i/x_i]$. Letzteres aber nur dann, wenn aus dem Kontext klar ist, daß alle x_i, $1 \leq i \leq n$, gemeint sind. Komponentenweise Ändern von Umgebungen δ ist analog definiert.

Die Bereichsordnung auf $\mathbf{Dom}^{\perp} \cup \mathbf{Dom}^{\omega}$ läßt sich kanonisch, d.h. punktweise, auf Zustände fortsetzen. STATE bildet damit ebenfalls einen Bereich. Für Umgebungen gilt dies nicht in gleicher Weise: Seien δ, δ' zwei Umgebungen, dann schreiben wir $\delta \subseteq \delta'$, wenn gilt:

$$\forall f: \delta(f) \subseteq \delta'(f).$$

δ heißt dann ein *Abkömmling* von δ'. δ heißt *eindeutig* falls gilt:

$$\forall f: |\delta(f)| = 1.$$

$DD(\delta)$ bezeichne die Menge aller *eindeutigen Abkömmlinge* von δ.

$\subseteq$ ist offensichtlich eine partielle Ordnung auf ENV, jedoch bilden ENV und $\subseteq$ keinen Bereich, da kein kleinstes Element existiert. Der potentielle Kandidat hierfür, der jedem f die leere Menge zuordnet, stellt keine zulässige Umgebung dar und zwar aus folgendem Grund:
Jede Funktion $f \in \delta(f)$ beschreibt ein mögliches *Ein/Ausgabeverhalten*, das der mit f bezeichnete Agent in der vorliegenden Umgebung δ zeigen kann. Ein hochgradig nichtdeterministischer Agent wird viele verschiedene Verhaltensweisen zeigen können, ein deterministischer genau eines. Vor diesem Hintergrund würde $\delta(f) = \varnothing$ bedeuten, daß f überhaupt kein Verhalten hat. Dies ist erscheint wenig intuitiv und wird daher durch die Definition des Umgebungsbegriffs ausgeschlossen. Tatsächlich bilden ENV und $\subseteq$ einen Präbereich (engl. "predomain", vgl. Kapitel 2), die zweite Bereichseigenschaft bleibt nämlich erhalten:

Sei $\delta_0 \subseteq \delta_1 \subseteq \ldots$ eine Kette von Umgebungen. Dann existiert ein Supremum $(\bigcup_i \delta_i) \in$ ENV für das gilt:

$$\forall f: (\bigcup_i \delta_i)(f) = \bigcup_i (\delta_i(f)).$$

In den kommenden Abschnitten soll in Bezug auf die Notation von Ausdrücken folgende *Schreibkonvention* gelten:

E[x] bezeichne einen Ausdruck, in dem der Identifikator $x \in$ ID frei vorkommen kann (aber nicht muß), sonst jedoch kein weiterer Bezeichner aus ID. Der Ausdruck E[y/x] entstehe aus E[x], indem jedes Vorkommnis von x durch y ersetzt wird. Analog entstehe E[E'/x] aus E[x], indem jedes Vorkommnis von x durch den Ausdruck E' ersetzt wird. Wenn aus dem Kontext klar ist, daß x durch E' ersetzt werden soll, schreiben wir auch kurz E[E']. Natürlich kann diese Schreibweise auch geschachtelt angewendet werden. Etwa ist E[E'[y]/x] ein Ausdruck der aus E entsteht, indem x durch E'[y] ersetzt wird, wobei in E' der Bezeichner y vorkommen kann.

3.2.1 Breitensemantik von Ausdrücken

Die Breitensemantik, kurz **B**-Semantik, eines Ausdrucks ist eine Menge von Wertetupeln. Dabei ist die Mengenstruktur Konsequenz der möglichen Mehrdeutigkeit, die letztlich auf den Auswahloperator [] zurückzuführen ist, während die Tupelbildung auf mehrstellige Resultate von Agenten zurückgeht. Die semantische Funktion **B** hat folgende Funktionalität

$$\mathbf{B}\colon \langle \text{exp} \rangle \to \text{ENV} \to \text{STATE} \to \wp(\mathbf{Dom}^{\perp}) \setminus \varnothing \cup \wp((\mathbf{Dom}^{\omega})^{*}) \setminus \varnothing.$$

Sie ist axiomatisch definiert. Wir schreiben $\mathbf{B}_{\delta,\sigma}[\![\mathrm{E}]\!]$ anstelle von $\mathbf{B}(\mathrm{E})(\delta)(\sigma)$.

$\mathbf{B}_{\delta,\sigma}[\![\mathrm{e}]\!] = \{\mathrm{e}\}$ für $\mathrm{e} \in \mathbf{Dom} \cup \{\varepsilon, \perp\}$,

$\mathbf{B}_{\delta,\sigma}[\![x]\!] = \{\sigma(x)\}$ für $x \in$ ID,

$\mathbf{B}_{\delta,\sigma}[\![\mathbf{ft.}\mathrm{E}]\!] = \{\mathrm{ft}(\mathrm{e}) \mid \mathrm{e} \in \mathbf{B}_{\delta,\sigma}[\![\mathrm{E}]\!]\}$,

$\mathbf{B}_{\delta,\sigma}[\![\mathbf{rt.}\mathrm{E}]\!] = \{\mathrm{rt}(\mathrm{e}) \mid \mathrm{e} \in \mathbf{B}_{\delta,\sigma}[\![\mathrm{E}]\!]\}$,

$\mathbf{B}_{\delta,\sigma}[\![\mathbf{isempty.}\mathrm{E}]\!] = \{\mathrm{isempty}(\mathrm{e}) \mid \mathrm{e} \in \mathbf{B}_{\delta,\sigma}[\![\mathrm{E}]\!]\}$,

$\mathbf{B}_{\delta,\sigma}[\![\mathrm{E}_1 \& \mathrm{E}_2]\!] = \{\mathrm{e}_1 \& \mathrm{e}_2 \mid \mathrm{e}_i \in \mathbf{B}_{\delta,\sigma}[\![\mathrm{E}_i]\!]\}$,

$\mathbf{B}_{\delta,\sigma}[\![\mathrm{E}_1 [] \mathrm{E}_2]\!] = \mathbf{B}_{\delta,\sigma}[\![\mathrm{E}_1]\!] \cup \mathbf{B}_{\delta,\sigma}[\![\mathrm{E}_2]\!]$,

$\mathbf{B}_{\delta,\sigma}[\![\mathbf{if}\ \mathrm{C}\ \mathbf{then}\ \mathrm{E}_1\ \mathbf{else}\ \mathrm{E}_2\ \mathbf{fi}]\!] = \{\mathrm{if}(\mathrm{c}, \mathrm{e}_1, \mathrm{e}_2) \mid \mathrm{c} \in \mathbf{B}_{\delta,\sigma}[\![\mathrm{C}]\!], \mathrm{e}_i \in \mathbf{B}_{\delta,\sigma}[\![\mathrm{E}_i]\!]\}$,

$$\text{wobei} \quad \mathrm{if}(\mathrm{c}, \mathrm{e}_1, \mathrm{e}_2) = \begin{cases} \perp & \text{falls } \mathrm{c} = \perp \\ \mathrm{e}_1 & \text{falls } \mathrm{c} = \text{true} \\ \mathrm{e}_2 & \text{falls } \mathrm{c} = \text{false} \end{cases},$$

$\mathbf{B}_{\delta,\sigma}[\![\mathrm{f}(\mathrm{E}_1, \ldots, \mathrm{E}_n)]\!] = \{\mathrm{f}(\mathrm{e}_1, \ldots, \mathrm{e}_n) \mid \mathrm{e}_i \in \mathbf{B}_{\delta,\sigma}[\![\mathrm{E}_i]\!]\}$,

für $f \in$ ‹primitive fct›,

$\mathbf{B}_{\delta,\sigma}[\![f(E_1, \ldots, E_n)]\!] = \{f(e_1, \ldots, e_n) \mid f \in \delta(f), e_i \in \mathbf{B}_{\delta,\sigma}[\![E_i]\!]\}$,

für $f \in$ FID.

Man beachte, daß die e, e_1, … hier entsprechend der Stelligkeit der zugehörigen Ausdrücke Wertetupel bezeichnen (daher $(\mathbf{Dom}^{\omega})^*$, vgl. Kapitel 2). Der Einfachheit halber gehen wir im folgenden davon aus, daß auch Stromausdrücke einstellig sind. Objektausdrücke sind es sowieso.

Die Menge $\mathbf{B}_{\delta,\sigma}[\![E]\!]$ heißt *Breite* von E bezüglich δ und σ. Sie repräsentiert die Gesamtheit aller möglichen Auswertungsergebnisse von E in der Umgebung δ und dem Zustand σ. Aufgrund der für denotationelle Semantiken typischen Repräsentation undefinierter Berechnungen durch $\perp$, ist $\mathbf{B}_{\delta,\sigma}[\![E]\!]$ niemals leer: Gehört E einem Objekttyp an, so liefert seine Auswertung entweder einen Wert oder divergiert. Letzteres wird durch $\perp$ repräsentiert. Ist E ein Stromausdruck, so generiert der Auswertungsprozeß (operationell gesprochen) entweder sukzessive die einzelnen Elemente eines endlichen Stromes und terminiert dann, dies wird durch einen Wert aus $\mathbf{Dom}^*$ repräsentiert, oder er erzeugt einen unendlichen Strom, dies wird durch einen Wert aus $\mathbf{Dom}^{\infty}$ repräsentiert, oder er erzeugt einige Elemente und divergiert dann, dies wird durch einen einen Wert aus $\mathbf{Dom}^* \times \{\perp\}$ repräsentiert. In jedem Fall enthält $\mathbf{B}_{\delta,\sigma}[\![E]\!]$ mindestens ein Element.
E heißt *deterministisch*, falls der Auswahloperator [] nicht in E vorkommt und *nichtdeterministisch* sonst. Determiniertheit ist eine syntaktische Eigenschaft. Aus ihr folgt nicht unmittelbar, daß E auch semantisch *eindeutig* ist, das heißt genau einen Wert bezeichnet. Allerdings gilt folgende Implikationsbeziehung:

$$E \text{ deterministisch} \wedge \delta \text{ eindeutig} \Rightarrow |\mathbf{B}_{\delta,\sigma}[\![E]\!]| = 1$$

In diesem Fall schreiben wir $e = \mathbf{B}_{\delta,\sigma}[\![E]\!]$ anstelle von $\{e\} = \mathbf{B}_{\delta,\sigma}[\![E]\!]$.
E ist ein $\mathbf{B}_{\delta,\sigma}$-*Abkömmling* bzw. ein **B**-*Abkömmling* von E', falls gilt:

$$\mathbf{B}_{\delta,\sigma}[\![E]\!] \subseteq \mathbf{B}_{\delta,\sigma}[\![E']\!] \quad \text{bzw.} \quad \forall\delta: \forall\sigma: \mathbf{B}_{\delta,\sigma}[\![E]\!] \subseteq \mathbf{B}_{\delta,\sigma}[\![E']\!]$$

DD(E) sei die Menge aller deterministischen **B**-Abkömmlinge von E. Das folgende Lemma zeigt, daß es solche Abkömmlinge stets gibt, DD(E) also niemals leer ist.

Lemma 3.5 (Existenz det. Abkömmlinge): Zu jedem Ausdruck E existieren deterministische Ausdrücke $E_1, \ldots, E_n$, so daß für beliebige Umgebungen δ und Zustände σ gilt:

$$\mathbf{B}_{\delta,\sigma}[\![E]\!] = \mathbf{B}_{\delta,\sigma}[\![E_1 [] \ldots [] E_n]\!].$$

Beweis: Aus den semantischen Gleichungen für **B** folgt, daß [] über alle übrigen Konstrukte distributiert. Terminduktion liefert dann die Behauptung.

□

Tatsächlich lassen sich geeignete E_i durch syntaktische Manipulationen aus E ableiten. Die entsprechenden Umformungsregeln sind in Abschnitt 5.2 als Transformationsregeln formuliert. Der synthetische Aufbau der semantischen Gleichungen für **B** gewährleistet die *Monotonie* der Breitensemantik bzgl. der Abkömmlingsbeziehung zwischen Ausdrücken:

Lemma 3.6 (Abkömmlingsmonotonie von B): Für beliebige Ausdrücke $E[x]$, E_1, E_2, Umgebungen δ und Zustände σ gilt:

$$\mathbf{B}_{\delta,\sigma}[\![E_1]\!] \subseteq \mathbf{B}_{\delta,\sigma}[\![E_2]\!] \Rightarrow \mathbf{B}_{\delta,\sigma}[\![E[E_1]]\!] \subseteq \mathbf{B}_{\delta,\sigma}[\![E[E_2]]\!].$$

Beweis: Terminduktion.

□

Kompositionalität – $\mathbf{B}_{\delta,\sigma}[\![E_1]\!] = \mathbf{B}_{\delta,\sigma}[\![E_2]\!] \Rightarrow \mathbf{B}_{\delta,\sigma}[\![E[E_1]]\!] = \mathbf{B}_{\delta,\sigma}[\![E[E_2]]\!]$ – folgt aus Symmetriegründen. Aus Lemma 3.5 ergibt sich die *Abkömmlingsadditivität* von **B** (vgl. [Berghammer 90]), die für das weitere Vorgehen von großer Bedeutung ist. Abkömmlingsadditivität bildet eine der Voraussetzungen dafür, daß Ausdrücke im folgenden Abschnitt funktional interpretiert werden können.

Lemma 3.7 (Abkömmlingsadditivität von B): Für beliebige Ausdrücke E, Umgebungen δ und Zustände σ gilt:

$$\mathbf{B}_{\delta,\sigma}[\![E]\!] = \bigcup_{E' \in DD(E)} \mathbf{B}_{\delta,\sigma}[\![E']\!].$$

Beweis: "$\subseteq$": Wegen Lemma 3.5 gibt es Ausdrücke $E_1, \ldots, E_n \in DD(E)$ für die gilt: $\mathbf{B}_{\delta,\sigma}[\![E]\!] = \mathbf{B}_{\delta,\sigma}[\![E_1]\!]\ldots[\![E_n]\!] = \bigcup_i \mathbf{B}_{\delta,\sigma}[\![E_i]\!] \subseteq \bigcup_{E' \in DD(E)} \mathbf{B}_{\delta,\sigma}[\![E']\!]$.
"$\supseteq$": Gemäß Definition gilt für jedes $E' \in DD(E)$: $\mathbf{B}_{\delta,\sigma}[\![E']\!] \subseteq \mathbf{B}_{\delta,\sigma}[\![E]\!]$ und damit unmittelbar die Behauptung.

□

Weitere Monotonie und Additivitätsresultate ergeben sich, wenn man die Abhängigkeit von **B** von δ, dem Umgebungsparameter, untersucht: Für festes E und gegebenes σ ist **B** monoton und stetig bezüglich der Inklusionsordnung auf Umgebungen:

Lemma 3.8 (Umgebungsmonotonie, -stetigkeit von B): Für beliebige Ausdrücke E, Umgebungen δ_1, δ_2 und Zustände σ gilt:

$$\delta_1 \subseteq \delta_2 \Rightarrow \mathbf{B}_{\delta_1,\sigma}[\![E]\!] \subseteq \mathbf{B}_{\delta_2,\sigma}[\![E]\!].$$

Für eine Kette von Umgebungen $\delta_0 \subseteq \delta_1 \subseteq \ldots$ mit Supremum $(\bigcup_i \delta_i)$ gilt:

$$\mathbf{B}_{(\bigcup_i \delta_i),\sigma}[\![E]\!] = \bigcup_i \mathbf{B}_{\delta_i,\sigma}[\![E]\!].$$

Beweis: a) Monotonie: Terminduktion.
b) Stetigkeit: "$\subseteq$": Sei $e \in \mathbf{B}_{(\bigcup_i \delta_i),\sigma}[\![E]\!]$. Dann gibt es wegen der Abkömmlingsadditivität von **B** einen deterministischen Abkömmling E' von E für den gilt: $e \in \mathbf{B}_{(\bigcup_i \delta_i),\sigma}[\![E']\!]$. Der Einfachheit halber nehmen wir an, daß in E' nur ein Funktionssymbol *f* vorkommt. Dies jedoch n>0 mal. Für die übrigen Fälle verläuft der Beweis analog.
Durch Terminduktion über den Aufbau von E' läßt sich zeigen, daß e eine Darstellung besitzt, in der Funktionen $f_1, \ldots, f_n$ angewandt auftreten: $e = A(f_1, \ldots f_n)$. Wobei jedes f_i eindeutig einem Vorkommnis des Bezeichners *f* entspricht und es gilt: $f_1, \ldots, f_n \in (\bigcup_i \delta_i)(f)$. Bis auf die Funktionen f_i ist die Darstellung $A(f_1, \ldots, f_n)$ unabhängig von δ und nur durch E' und σ bestimmt. Aus der Definition des Supremums $(\bigcup_i \delta_i)$ folgt, daß es ein Kettenglied δ_k gibt mit $f_1, \ldots, f_n \in \delta_k(f)$. Also $e \in \mathbf{B}_{\delta_k,\sigma}[\![E']\!] \subseteq \mathbf{B}_{\delta_k,\sigma}[\![E]\!] \subseteq \bigcup_i \mathbf{B}_{\delta_i,\sigma}[\![E]\!]$.
"$\supseteq$": Sei δ_k ein beliebiges Kettenglied, dann gilt definitionsgemäß $\delta_k \subseteq (\bigcup_i \delta_i)$. Wegen der Umgebungsmonotonie folgt daraus die Behauptung.
□

Dual zur Abkömmlingsadditivtät ist die *Umgebungsadditivität* von **B**. Sie bildet die zweite Voraussetzung für die Definition der funktionalen Semantik im folgenden Abschnitt, gilt aber im strengen Sinne nur, wenn gewisse syntaktische Voraussetzungen erfüllt sind.

Lemma 3.9 (Umgebungsadditivität von B): Sei E ein Ausdruck, in dem kein Bezeichner $f \in$ FID mehr als einmal vorkommt. δ und σ beliebig. Dann gilt:

$$\mathbf{B}_{\delta,\sigma}[\![E]\!] = \bigcup_{\delta' \in DD(\delta)} \mathbf{B}_{\delta',\sigma}[\![E]\!].$$

Beweis: E sei ein Ausdruck, der die im Lemma genannte syntaktische Bedingung erfüllt.
"$\subseteq$": Sei $e \in \mathbf{B}_{\delta,\sigma}[\![E]\!]$. Dann gibt es wie im Beweis zu Lemma 3.8 einen deterministischen Abkömmling E' von E für den gilt: $e \in \mathbf{B}_{\delta,\sigma}[\![E']\!]$. O.B.d.A. nehmen wir an, daß die paarweise verschiedenen Bezeichner $f_1, \ldots, f_n \in$ FID genau einmal in E' vorkommen und sonst kein weiteres Symbol aus FID. (Tatsächlich gibt es deterministische Abkömmlinge von E in denen Funktionsbezeichner mehrfach vorkommen. Jeder von diesen ist aber **B**-äquivalent zu einem Abkömmling, in dem Funktionsbezeichner jeweils höchstens einmal vorkommen.)
Wie im Beweis zu 3.8, läßt sich für e eine Darstellung finden, die durch E' und σ bestimmt ist und in der Funktionen $f_1, \ldots, f_n$ angewandt auftreten: $e = A(f_1, \ldots, f_n)$. Dabei gilt: $f_i \in \delta(f_i)$. Weil alle f_i paarweise verschieden sind, gibt es einen deterministischen Abkömmling δ' von δ mit der Eigenschaft $\delta'(f_i) = \{f_i\}$. Also $e \in \mathbf{B}_{\delta',\sigma}[\![E']\!] \subseteq \mathbf{B}_{\delta',\sigma}[\![E]\!] \subseteq \bigcup_{\delta' \in DD(\delta)} \mathbf{B}_{\delta',\sigma}[\![E]\!]$.
"$\supseteq$": Weil für jedes $\delta' \in DD(\delta)$ definitionsgemäß $\delta' \subseteq \delta$ gilt, folgt diese Richtung aus der Umgebungsmonotonie von **B**.
□

Daß die behauptete Mengenidentität zwar unter der im Lemma formulierten Einschränkung, im allgemeinen jedoch nicht gilt, zeigt folgendes Beispiel:

Beispiel 3.10: Sei δ eine Umgebung mit $\delta(f) = \{succ, pred\}$, δ_s, δ_p seien deterministische Abkömmlinge von δ für die gilt: $\delta_s(f) = \{succ\}$ und $\delta_p(f) = \{pred\}$. Dann folgt:

$$\mathbf{B}_{\delta_p,\sigma}[\![f(1)+f(1)]\!] \cup \mathbf{B}_{\delta_s,\sigma}[\![f(1)+f(1)]\!] = \{0, 4\} \subseteq \{0, 2, 4\} = \mathbf{B}_{\delta,\sigma}[\![f(1)+f(1)]\!]$$

Durch den Übergang zu den deterministischen Abkömmlingen einer Umgebung werden in der Regel gewisse Mischungen ausgeschlossen, da jedem Vorkommnis eines Abbildungsbezeichners dieselbe Funktion zugewiesen wird. In einer nichtdeterministischen Umgebung können für unterschiedliche Vorkommnisse unterschiedliche Funktionen ausgewählt werden. Durch geeignete syntaktische Umbenennungen läßt sich jedoch jeder Ausdruck in eine Form überführen, für die Gleichheit gilt.

□

Sei $E[f_{11}, \ldots, f_{1n_1} \ldots f_{m1}, \ldots, f_{mn_m}]$ ein Ausdruck in dem die paarweise verschiedenen Bezeichner $f_{ij} \in$ FID jeweils genau einmal vorkommen. Die Bezeichner $f_1, \ldots, f_m \in$ FID seien ebenfalls paarweise verschieden und verschieden von den f_{ij}. Dann heißt

$$E[f_{11}, \ldots, f_{1n_1}, \ \ldots, \ f_{m1}, \ldots, f_{mn_m}]$$

die *Substitutions-Normalform*, kurz *S-Normalform*, von

$$E[f_1/f_{11}, \ldots, f_1/f_{1n_1}, \ \ldots, \ f_m/f_{m1}, \ldots, f_m/f_{mn_m}]$$

und wird mit $E^{\sim}$ bezeichnet. Die S-Normalform eines Ausdrucks E entsteht also dadurch, daß jedes Vorkommnis eines Abbildungsbezeichners durch eine disjunkte Kopie ersetzt wird. Die Normalform ist bis auf die Wahl der Kopienamen eindeutig bestimmt.
Ist δ eine Umgebung und $E^{\sim}$ die S-Normalform von E, dann ist

$$\delta[\delta(f_1)/f_{11},\ldots,\delta(f_1)/f_{1n_1}, \ldots, \delta(f_m)/f_{m1},\ldots,\delta(f_m)/f_{mn_m}]$$

die zugehörige *S-Normalform* von δ. Sie wird analog mit $\delta^{\sim}$ bezeichnet. Offensichtlich ist "~" idempotent: $E^{\sim} = E^{\sim\sim}$ und $\delta^{\sim} = \delta^{\sim\sim}$ und es gilt:

$$\mathbf{B}_{\delta,\sigma}[\![E]\!] = \mathbf{B}_{\delta^{\sim},\sigma}[\![E^{\sim}]\!]$$

Abkömmlings- und Umgebungsadditivität lassen daher in einer Gleichung zusammenfassen:

Korollar 3.11 (Kombinierte Additivität von B): Für beliebige Ausdrücke E, Umgebungen δ und Zustände σ gilt:

$$\mathbf{B}_{\delta,\sigma}[\![E]\!] = \bigcup_{\delta' \in DD(\tilde{\delta}), E' \in DD(\tilde{E})} \mathbf{B}_{\delta',\sigma}[\![E']\!]$$

Beweis:

$$\begin{aligned}
& \mathbf{B}_{\delta,\sigma}[\![E]\!] \\
= \quad & \{ \text{ siehe oben } \} \\
& \mathbf{B}_{\tilde{\delta},\sigma}[\![\tilde{E}]\!] \\
= \quad & \{ \text{ Umgebungsadditivität von } \mathbf{B}, \tilde{E} \text{ erfüllt die Voraussetzungen von Lemma 3.9 } \} \\
& \bigcup_{\delta' \in DD(\tilde{\delta})} \mathbf{B}_{\delta',\sigma}[\![E]\!] \\
= \quad & \{ \text{ Abkömmlingsadditivität von } \mathbf{B} \} \\
& \bigcup_{\delta' \in DD(\tilde{\delta})} \bigcup_{E' \in DD(\tilde{E})} \mathbf{B}_{\delta',\sigma}[\![E']\!]
\end{aligned}$$

Beachte: $|\mathbf{B}_{\delta',\sigma}[\![E']\!]| = 1$.

□

Dieses Korollar bildet die Basis für die Funktionssemantik von Ausdrücken, die wir nun definieren werden.

3.2.2 Funktionssemantik von Ausdrücken

Die funktionale Semantik **F** ordnet jedem Ausdruck eine Menge stetiger Funktionen zu. Sie stützt sich dabei auf das eben herausgearbeitete Additivitätsresultat 3.11, dessen Bedeutung sich im Zusammenhang mit folgendem Satz erschließt:

Satz 3.12 (Beziehung zwischen Ausdrücken und stetigen Fkt.): Sei E[x] ein deterministischer Ausdruck, δ eine eindeutige Umgebung und σ ein beliebiger Zustand. Dann gilt:

$$\lambda x.\mathbf{B}_{\delta,\sigma[x/x]}[\![E[x]]\!]$$

ist eine monotone und stetige Abbildung aus **MAP**.

Beweis: Die Grundfunktionen &, ft, rt, isempty und if sind stetig. Eine eindeutige Umgebung ordnet jedem $f \in FID$ genau eine stetige Funktion zu. Die Komposition stetiger Funktionen ist wieder stetig.

□

Ein deterministischer Ausdruck entspricht also in einer eindeutigen Umgebung einer stetigen Funktion. Die Funktionalität dieser Abbildung f hängt vom Typ von *x* und E[*x*] ab. Insbesondere ist f∈ **AGT**, falls E ein Stromausdruck ist und f∈**FCT**, falls E ein reiner Objektausdruck ist. Aufgrund der Additivität von **B** ist es nun möglich, beliebige Ausdrücke in beliebigen Umgebungen funktional zu interpretieren, nämlich durch *Funktionsmengen*. Jedes f aus einer solchen Menge ist durch einen deterministischen Ausdrucks- und einen eindeutigen Umgebungsabkömmling bestimmt. Darin besteht das Prinzip der semantischen Abbildung **F**.

$$\mathbf{F}\text{: ‹exp›} \to \text{ENV} \to \wp(\mathbf{MAP})\backslash\varnothing$$

ist wie folgt definiert:

$$\mathbf{F}_\delta[\![\text{E}[x]]\!] = \{\lambda x.\mathbf{B}_{\delta',\sigma[x/x]}[\![\text{E}']\!] \mid \text{E}' \in \text{DD}(\text{E}[x]\tilde{}),\ \delta' \in \text{DD}(\delta\tilde{}),\ \sigma \in \text{STATE}\}.$$

Beachte, daß $\mathbf{B}_{\delta',\sigma[x/x]}[\![\text{E}']\!]$ nicht von σ abhängt, da *x* konventionsgemäß der einzige Bezeichner aus ID ist, der in E[*x*] vorkommt. Treten mehrere Bezeichner auf, so liefert **F** entsprechend mehrstellige Funktionen.

Daß es günstig ist, Ausdrücke als implizite Charakterisierungen von Funktionsmengen zu deuten, zeigt schon der nächste Abschnitt. Dort wird die Semantik von Gleichungssystemen durch Rückgriff auf das bekannte Kleenesche Fixpunktargument (vgl. Satz 2.5) erklärt. Voraussetzung für dessen Anwendung, ist die Stetigkeit der Abbildungen auf den rechten Gleichungsseiten. Syntaktisch finden sich dort Ausdrücke, durch **F** wird die Verbindung zu stetigen Funktionen hergestellt. Wegen der Idempotenz des Normalformoperators (bzgl. **B**) gilt:

$$\mathbf{F}_\delta[\![\text{E}]\!] = \mathbf{F}_{\delta\tilde{}}[\![\text{E}\tilde{}]\!].$$

Weiterhin übertragen sich die zuvor für für **B** nachgewiesenen Eigenschaften. Insbesondere die *Umgebungsstetigkeit*, eine Eigenschaft, die für die Behandlung rekursiver Definitionen wichtig ist. Kompositionalitätseigenschaften von **F** werden in Abschnitt 3.3 gesondert behandelt.

Lemma 3.13 (Umgebungsmonotonie, -stetigkeit von F): Für beliebige Ausdrücke E und Umgebungen δ_1, δ_2 gilt:

$$\delta_1 \subseteq \delta_2 \Rightarrow \mathbf{F}_{\delta_1}[\![\text{E}]\!] \subseteq \mathbf{F}_{\delta_2}[\![\text{E}]\!].$$

Für eine Kette von Umgebungen $\delta_0 \subseteq \delta_1 \subseteq \ldots$ mit Supremum $\bigcup_i \delta_i$ gilt:

$$\mathbf{F}_{(\bigcup_i \delta_i)}[\![\text{E}]\!] = \bigcup_i \mathbf{F}_{\delta_i}[\![\text{E}]\!].$$

Beweis: a) Monotonie: Aus der Definition von ˜ und DD(δ) folgt: $\delta_1 \subseteq \delta_2 \Rightarrow \delta_1\tilde{} \subseteq \delta_2\tilde{} \Rightarrow \text{DD}(\delta_1\tilde{}) \subseteq \text{DD}(\delta_2\tilde{})$ und damit $\mathbf{F}_{\delta_1}[\![\text{E}]\!] \subseteq \mathbf{F}_{\delta_2}[\![\text{E}]\!]$ direkt aus der Definition von **F**.

b) Stetigkeit "⊆": O.B.d.A sei *x* der einzige Bezeichner in E. Sei $\text{f} \in \mathbf{F}_{(\bigcup_i \delta_i)}[\![\text{E}]\!]$. Dann gibt es ein $\text{E}' \in \text{DD}(\text{E}\tilde{})$ und ein $\delta' \in \text{DD}((\bigcup_i \delta_i)\tilde{})$, so daß

$$\text{f} = \lambda x.\mathbf{B}_{\delta',\sigma[x/x]}[\![\text{E}']\!]$$

mit σ beliebig. Aus der Definition von $\tilde{}$ und $\bigcup_i \delta_i$ folgt: $(\bigcup_i \delta_i)\tilde{} = \bigcup_i (\delta_i\tilde{})$. Also gilt: $\delta' \in DD(\bigcup_i (\delta_i\tilde{}))$ und daher existiert ein Kettenglied δ_k mit $\delta' \in DD(\delta_k\tilde{})$.
Also $f \in \mathbf{F}_{\delta_k}[\![E]\!] \subseteq \bigcup_i \mathbf{F}_{\delta_i}[\![E]\!]$.
"$\supseteq$": Folgt aus der Umgebungsmonotonie von **F**.
□

Den Zusammenhang zwischen **B** und **F** beleuchtet folgendes Lemma. Es zeigt, daß sich die Breite von E stets aus der zugehörigen Funktionsmenge ableiten läßt:

Lemma 3.14 (Zusammenhang zwischen B und F): Für beliebige Ausdrücke E, Umgebungen δ und Zustände σ gilt:

$$\mathbf{B}_{\delta,\sigma}[\![E[x]]\!] = \{ f(\sigma(x)) \mid f \in \mathbf{F}_\delta[\![E[x]]\!] \}.$$

Beweis: Folgt wegen Korollar 3.11 aus der Definition von **F**.
□

Zwei **F**-äquivalente Ausdrücke sind also auch **B**-äquivalent sind:

$$\mathbf{F}_\delta[\![E[x]]\!] = \mathbf{F}_\delta[\![E'[x]]\!] \quad \Rightarrow \forall\sigma: \mathbf{B}_{\delta,\sigma}[\![E[x]]\!] = \mathbf{B}_{\delta,\sigma}[\![E'[x]]\!].$$

Die Umkehrung gilt jedoch nicht (vgl. [Broy 90], S. 11 - 12).

3.2.3 Semantik von Gleichungssystemen

Sei GS[x] ein Gleichungssystem der Gestalt

$$s_1 \equiv S_1[x, s_1, \ldots, s_n],$$
$$\ldots,$$
$$s_n \equiv S_n[x, s_1, \ldots, s_n].$$

(Beachte, daß die S_i konventionsgemäß die Stelligkeit 1 haben, siehe Seite 28. Die s_i sind dann einfach Identifikatoren und keine Tupel wie im allgemeinen Fall.) $x \in$ ID sei ein Bezeichner, der nur auf rechten Gleichungsseiten auftritt. GS ist durch *x parametrisiert*. Nicht parametrisierte Gleichungssysteme heißen *geschlossen*.

Für jedes S_i ist $\mathbf{F}_\delta[\![S_i]\!]$ eine Menge n+1-stelliger stromverarbeitender Funktionen. Wählt man aus jeder Menge eine Funktion aus, so ergibt sich ein System semantischer Gleichungen auf deren rechten Seiten nur stetige Abbildungen vorkommen. Es besitzt eine kleinste Fixpunktlösung, die

gemäß Satz 2.6 stetig von *x* abhängt. Die funktionale Semantik eines Gleichungssystems läßt sich daher wie folgt definieren:

$$\mathbf{F}: \langle \text{eq\_sys} \rangle \rightarrow \text{ENV} \rightarrow \wp(\mathbf{AGT})\backslash\varnothing,$$

wobei:

$$\mathbf{F}_\delta[\![\text{GS}[x]]\!] = \{\ \lambda x.\text{fix} \begin{pmatrix} s_1 = f_1(x, s_1, \ldots, s_n) \\ \ldots \\ s_n = f_n(x, s_1, \ldots, s_n) \end{pmatrix} \mid\ f_i \in \mathbf{F}_\delta[\![S_i]\!]\ \}.$$

Jedes semantische Gleichungssystem dieser Form ist eine *semantische Instanz* von GS. Während die Stelligkeit der Eingabe eines $f \in \mathbf{F}_\delta[\![\text{GS}[x]]\!]$ durch die Anzahl der Parameter von GS bestimmt ist, entspricht die Stelligkeit der Ausgabe der Anzahl der rechtsseitig vorkommenden Strombezeichner. Die Anordnung der Komponenten wird durch die Aufschreibung im Gleichungssystem festgelegt.

Agenten und vollständige Programme können interne Ströme aufweisen, die lokal verwendet, aber nach außen nicht sichtbar seien sollen. Mit Hilfe der folgenden Definition werden sie verborgen:
Sei GS[*x*] das obige Gleichungssystem und $(o_1, \ldots, o_m)$, $m \leq n$, eine Permutation (einer Teilmenge) der Bezeichner $\{s_1, \ldots, s_n\}$, d.h. es existiere eine injektive Abbildung

$$\pi: \{1, \ldots, m\} \rightarrow \{1, \ldots, n\}$$

mit der Eigenschaft: $o_i = s_{\pi(i)}$. Dann sei:

$$\mathbf{F}_\delta[\![\text{GS}[x], (o_1, \ldots, o_m)]\!] = \{\ g \mid \exists f \in \mathbf{F}_\delta[\![\text{GS}[x]]\!]: \forall x:$$

$g(x) = (o_1, \ldots, o_m)$

$\Leftrightarrow$ (

$f(x) = (s_1, \ldots, s_n) \wedge \forall i, 1 \leq i \leq m: o_i = s_{\pi(i)}$) }.

Das n-stellige Tupel $f(x) = (s_1, \ldots, s_n)$ ist nach Definition die kleinste Fixpunklösung einer semantischen Instanz von GS. $g(x) = (o_1, \ldots, o_m)$ ist dann das m-stellige Tupel, das nur diejenigen Ströme aufführt, die diese Lösung den Bezeichnern $o_1, \ldots, o_m$ zuweist. Man beachte, daß $\mathbf{F}_\delta[\![\text{GS}[x], (o_1, \ldots, o_m)]\!]$ von der Reihenfolge der Gleichungen in GS unabhängig ist.

Aus der Definition von $\mathbf{F}_\delta[\![\text{GS}]\!]$ ergibt sich sofort die *Umgebungsstetigkeit* und *-additivität* von **F** auch für Gleichungssysteme.

3.2.4 Semantik von Funktionen

Die Verwendung nichtdeterministischer Ausdrücke bei der Funktionsdeklaration führt zu "mehrdeutigen" Funktionen. Semantisch werden diese durch Mengen strikter und damit stetiger Abbildungen über den flachen Grundbereichen erklärt. Der zuvor getriebene Aufwand zahlt sich nun in sofern aus, als daß **F** für dieses Sprachsegment sehr knapp beschreibbar ist. **F** hat die Funktionalität

$$\mathbf{F}\text{: ‹function›} \to \text{ENV} \to \wp(\mathbf{FCT}_S)\backslash\emptyset$$

und ist wie folgt definiert:

$$\mathbf{F}_\delta[\![\mathbf{funct}\ f \equiv \mathbf{u}\ x \to \mathbf{v}\text{:}\ \ \text{E}[x]\ \ \mathbf{end}\,]\!].$$

ist der (bzgl. $\subseteq$) *größte Fixpunkt* FIX(τ) des Funktionals τ: $\wp(\mathbf{FCT}_S)\backslash\emptyset \to \wp(\mathbf{FCT}_S)\backslash\emptyset$:

$$\tau.\text{M} = \text{strict}\ \mathbf{F}_{\delta[\text{M}/f]}[\![\text{E}]\!].$$

Offensichtlich macht diese Definition nur Sinn, wenn τ tatsächlich einen größten Fixpunkt besitzt. Um dies zu zeigen, weisen wir nach, daß τ den Bedingungen aus Satz 2.3 genügt.

Satz 3.15 (Wohldefiniertheit von τ): Es gilt:

i) τ ist stetig bzgl. $\subseteq$

ii) $\exists \text{M} \in \wp(\mathbf{FCT}_S)\backslash\emptyset\text{: M} \subseteq \tau.\text{M}$

Beweis: i) Folgt direkt aus der Umgebungsstetigkeit von **F**.
ii) Nach Definition gilt für ein beliebiges $f \in \mathbf{FCT}_S$:

$$\begin{aligned}\tau.\{f\} &= \text{strict}\ \mathbf{F}_{\delta[\{f\}/f]}[\![\text{E}]\!] \\ &= \text{strict}\ \{\lambda x.\mathbf{B}_{\delta',\sigma[x/x]}[\![\text{E}']\!] \mid \delta' \in \text{DD}(\delta[\{f\}/f]\tilde{}),\ \text{E}' \in \text{DD}(\text{E}\tilde{}),\ \sigma \in \text{STATE}\}\end{aligned}$$

Betrachte nun einen beliebigen Ausdruck $\text{E}' \in \text{DD}(\text{E}\tilde{})$ und eine beliebig Umgebung $\delta' \in \text{DD}(\delta\tilde{})$. f_1, …, f_n seien die disjunkten Kopien, die im Zuge des Übergangs von E zur S-Normalform E˜ für f substituiert wurden, und die auch in E' auftreten. Weil alle deterministischen AL-Konstrukte stetig sind und weil die Funktionskomposition und "strict" stetige Operatoren sind, ist auch folgendes Funktional ξ: $\mathbf{FCT}_S \to \mathbf{FCT}_S$ stetig:

$$\xi.\text{f} = \text{strict}\ \lambda x.\mathbf{B}_{\delta'[\{f\}/f_1,\ \ldots,\ \{f\}/f_n],\sigma[x/x]}[\![\text{E}']\!]$$

Es gilt: $\delta'[\{f\}/f_1, \ldots, \{f\}/f_k] \in \text{DD}(\delta[\{f\}/f]\tilde{})$ und daher: $\xi.\text{f} \in \tau.\{f\}$. Nach Kleene (vgl. Satz 2.2) hat ξ einen kleinsten Fixpunkt $\text{fix}(\xi) \in \mathbf{FCT}_S$ für den gilt:

$$\text{fix}(\xi) = \xi.\text{fix}(\xi) \in \tau.\{\text{fix}(\xi)\}$$

Also $\{\text{fix}(\xi)\} \subseteq \tau.\{\text{fix}(\xi)\}$.

□

Man beachte, daß die Semantik einer Funktionsdeklaration **fct** $f \equiv$ **u** $x \rightarrow$ **v:** E von der Umgebung δ abhängt, aber unabhängig von deren Wert an der Stelle f ist. Die Semantik eines Systems wechselseitig rekursiver Funktionsdeklarationen

$$\mathbf{F}_\delta[\![\quad \textbf{funct } f_1 \equiv \mathbf{u_1}\ x \rightarrow \mathbf{v_1}\textbf{:}\ \ E_1\ \ \textbf{end},\ \ldots,\ \textbf{funct } f_n \equiv \mathbf{u_n}\ x \rightarrow \mathbf{v_n}\textbf{:}\ \ E_n\ \ \textbf{end}\ ]\!]$$

ist das Tupel

$$(M_1, \ldots, M_n) \in (\wp(\mathbf{FCT}_S)\backslash\varnothing)^n,$$

der Funktionsmengen, das den größten Fixpunkt des entsprechend definierten, mehrstelligen Funktionals darstellt. Ein System von Funktionsdeklarationen heißt *geschlossen*, wenn es für jeden auftretenden Funktionsbezeichner eine Definition enthält. Die Semantik geschlossener Systeme ist von der Umgebung unabhängig.

Üblicherweise definiert man die Bedeutung rekursiver Funktionsdefinitionen (vgl. [Mosses 90]) durch Rückgriff auf den *kleinsten* Fixpunkt des zugehörigen Funktionals. Die obigen Definitionen stützt sich dagegen auf den *größten* Fixpunkt von τ. Welche Gründe und Konsequenzen dies hat, wird im Anschluß an den folgenden Abschnitt diskutiert.

3.2.5 Semantik von Agenten

Die Semantik von Agenten wird analog zu der von Funktionen erklärt. **F** ordnet jeder Agentendeklarationen eine Menge stromverarbeitender Funktionen zu. Der Rumpf eines Agenten besteht aus einem eventuell nichtdeterministischen Gleichungssystem. In Abschnitt 3.2.3 haben wir gesehen, daß Gleichungssysteme semantisch wie Ausdrücke behandelt werden können. Insbesondere können sie genauso durch Mengen stetiger Funktionen beschrieben werden. Für Agenten hat **F** hat die Funktionalität

$$\mathbf{F}: \langle\text{agent}\rangle \rightarrow \text{ENV} \rightarrow \wp(\mathbf{AGT}_S)\backslash\varnothing$$

und ist wie folgt definiert:

$$\mathbf{F}_\delta[\![\textbf{agent } f \equiv \mathbf{u}\ x, \textbf{chan } \mathbf{v}\ i \rightarrow \textbf{chan } \mathbf{w}\ o\textbf{:}\ \ GS[x,i]\ \ \textbf{end}\ ]\!]$$

ist der (bzgl. $\subseteq$) *größte Fixpunkt* des Funktionals τ: $\wp(\mathbf{AGT}_S)\backslash\varnothing \rightarrow \wp(\mathbf{AGT}_S)\backslash\varnothing$:

$$\tau.M = \text{strict } \mathbf{F}_{\delta[M/f]}[\![GS, o]\!].$$

Beachte, daß sich der Striktheitsoperator hier vereinbarungsgemäß (vgl. S. 29) nur auf den Objektparameter *x* bezieht. Weil **F** nicht nur für Ausdrücke, sondern auch für Gleichungssysteme umgebungsstetig ist, läßt sich die Existenz eines größten Fixpunktes genau wie oben beweisen: τ erfüllt die Bedingungen aus Satz 2.3.

Satz 3.16 (Wohldefiniertheit von τ): Es gilt:

i) τ ist stetig bzgl. $\subseteq$

ii) $\exists M \in \wp(\mathbf{AGT}_S)\backslash\emptyset: M \subseteq \tau.M$

Beweis: Analog zu Satz 3.15.

□

Systeme wechselseitig rekursiver Deklarationen werden ebenfalls wie oben behandelt. Für Beweiszwecke (siehe Abschnitt 5.3.1), ist es oft günstiger, die Semantik eines Agenten nicht als größten Fixpunkt des Funktionals τ aufzufassen, sondern *prädikativ* zu charakterisieren. Darüber hinaus werden Agenten auch in der zweiten Phase einer gemäß der FOCUS-Methodik ausgeführten Systementwicklung prädikativ beschrieben (siehe Abschnitt 1.1). Prädikative Charakterisierungen können durch Ausnutzen der obigen Definitionen aus den Fixpunktbeschreibungen abgeleitet werden[1]. Wir zeigen dies an einem einfachen Beispiel:

Beispiel 3.17 (Prädikative Charakterisierung von Agenten): Die Semantik des folgenden Agenten *f* ist nach Definition die größte Lösung N der Gleichung

$$M = \mathbf{F}_{\delta[M/f]}[\![\ \textbf{agent}\ f \equiv \textbf{chan nat}\ i \rightarrow \textbf{chan nat}\ o: \ o \equiv \textbf{ft.}i{+}1 \ \|\ \textbf{ft.}i{+}2\ \&\ f(\textbf{rt.}i)\ \textbf{end}\ ]\!]$$

Wir berechnen die rechte Seite:

$\mathbf{F}_{\delta[M/f]}[\![\textbf{agent}\ f \equiv \textbf{chan nat}\ i \rightarrow \textbf{chan nat}\ o:\ o \equiv \textbf{ft.}i{+}1 \ \|\ \textbf{ft.}i{+}2\ \&\ f(\textbf{rt.}i)\ \textbf{end}\ ]\!]$

$=$ { Definition von **F** für Agenten und Gleichungssysteme }

$\mathbf{F}_{\delta[M/f]}[\![\textbf{ft.}i{+}1 \ \|\ \textbf{ft.}i{+}2\ \&\ f(\textbf{rt.}i)\ ]\!]$

$=$ { Definition von **F** für Ausdrücke }

$\{\ \lambda i.\mathbf{B}_{\delta',\sigma[i/i]}[\![E']\!]\ \mid\ E' \in DD(\textbf{ft.}i{+}1 \ \|\ \textbf{ft.}i{+}2\ \&\ f(\textbf{rt.}i)),\ \delta' \in DD(\delta[M/f])\ \}$

$=$ { Definition von DD, *f* ist der einzige Funktionsbezeichner der im Rumpf vorkommt }

---

[1] Manche Autoren haben dies noch viel weitgehender ausgenutzt und die Semantik der von ihnen betrachteten Sprachen unter dem Schlagwort "Programs are Predicates" vollständig prädikativ beschrieben (vgl. [Hehner 84], [Hoare 85b], [Broy 87c], [Broy, Lengauer 91], [Olderog 91]).

$$\{\ \lambda i.\mathbf{B}_{\delta[\{f\}/f],\sigma[i/i]}[\![\mathbf{ft.}i+1\ \&\ f(\mathbf{rt.}i)]\!],\ \lambda i.\mathbf{B}_{\delta[\{f\}/f],\sigma[i/i]}[\![\mathbf{ft.}i+2\ \&\ f(\mathbf{rt.}i)]\!]\ |\ f\in M\ \}$$

$$=$$

{ Definition von **B** }

$$\{\ \lambda i.ft(i)+1\ \&\ f(rt(i)),\ \ \lambda i.ft(i)+2\ \&\ f(rt(i))\ \ |\ \ f\in M\ \}.$$

Also gilt: N ist die größte Menge, die folgendes Prädikat erfüllt

$$f\in N \quad\Leftrightarrow\quad \exists g\in N:\ \forall i\in \mathbf{Nat}^{\omega}:\ f(i) = ft(i)+1\ \&\ g(rt.i)\ \vee\ f(i) = ft(i)+2\ \&\ g(rt.i) \qquad \square$$

Aus der Bestimmung der Agenten- und Funktionssemantik als die *größten* Fixpunkte der zugehörigen Funktionale, ergeben sich einige Konsequenzen, die hier kurz diskutiert werden sollen. Man betrachte den Agenten *ntr* (*ntr* steht für nichtterminierende Rekursion):

agent *ntr* ≡ **chan v** *i* → **chan w** *o*: *o* ≡ *ntr*(*i*) **end**

Man zeigt leicht, daß die Semantik von *ntr* der gesamte Funktionenraum [$\mathbf{V}^{\omega} \to \mathbf{W}^{\omega}$] ist. *ntr* kann jedes beliebige Verhalten zeigen, es ist maximal nichtdeterministisch (vgl. die "Chaos"-Semantik in [Hoare 85b]). Operationell betrachtet, führt ein Aufruf von *ntr* unabhängig von der Eingabe zur Divergenz, was auf der denotationellen Ebene eigentlich durch die vollständig undefinierte Funktion $\lambda i.\bot$ repräsentiert wird. Daraus folgt, daß es eine Diskrepanz zwischen einer operationellen Semantik, die die intuitive Vorstellung von Programmabläufen präzisiert (vgl. [Dederichs 90]), und der angegebenen denotationellen Semantik gibt. Zwischen beiden besteht folgender Zusammenhang:
M_{op} sei die Menge der Stromfunktionen, die die operationelle Semantik einem Agenten zuordnen würde, und M_{de} sei die Menge, die die denotationelle Semantik diesem Agenten zuordnet. Dann gilt:

$$M_{op} \subseteq M_{de} \wedge \forall f_{de}\in M_{de}: \exists f_{op}\in M_{op}: f_{op} \sqsubseteq f_{de}$$

Die denotationelle Semantik ist damit eine *Implementierung* (vgl. Kapitel 2, S. 18) der operationellen Semantik mit der folgenden speziellen Eigenschaft: Wenn immer ein Agent *f* für einen Eingabestrom *i* von einer gewissen Stelle an divergiert, dann enthält die denotationelle Semantik von *f* alle Funktionen, die von dieser Stelle an beliebiges ausgeben. *ntr* divergiert für alle Argumente von Anfang an, die denotationelle Semantik enthält daher alle Funktionen.

Ich habe das Auseinanderklaffen zwischen operationeller und denotationeller Beschreibung beim Entwurf (der Semantik) von AL bewußt in Kauf genommen. Im Prinzip wäre es zwar möglich, für AL eine denotationelle Semantik anzugeben, die genau mit der operationellen "übereinstimmt". Der dazu nötige formale Apparat ist aber erheblich umfangreicher (vgl. [Broy 87b]). Die entstehende Semantik wäre entsprechend komplizierter. Da Transformationsregeln und der Beweis ihrer Korrektheit in dieser Arbeit ein wesentliches Gewicht haben, kommt es auf die Einfachheit der Semantik entscheidend an. Die getroffene Vereinfachung erscheint mir deshalb gerechtfertigt.

3.2.6 Semantik von Programmen

Ein vollständiges AL-Programm besteht aus einem geschlossenen System von Funktions- und Agentendeklarationen, sowie einem wohlgeformten Gleichungsteil.

$$\mathbf{F}: \langle \text{prg} \rangle \rightarrow \wp(\mathbf{AGT}_S)\backslash\emptyset$$

F ist für

program *prg* ≡ **chan v** *i* → **chan w** *o*:
funct $f_1 \equiv F_1$, …, **funct** $f_n \equiv F_n$,
agent $g_1 \equiv G_1$, …, **agent** $g_m \equiv G_m$,
GS
end

wie folgt definiert: δ_0 sei eine beliebige Anfangsumgebung, und es gelte:

$\mathbf{F}_{\delta_0}[\![\mathbf{fct}\ f_1 \equiv F_1, \ldots, \mathbf{fct}\ f_n \equiv F_n]\!] = (M_1 \ldots, M_n)$,
$\delta_1 = \delta_0[M_1/f_1,\ldots,M_n/f_n]$,
$\mathbf{F}_{\delta_1}[\![\mathbf{agent}\ g_1 \equiv A_1 \ldots, \mathbf{agent}\ g_m \equiv A_m\ ]\!] = (N_1, \ldots, N_m)$,
$\delta_2 = \delta_1[N_1/g_1,\ldots,N_m/g_m]$.

Die Semantik von *prg* ist dann diejenige Menge stromverarbeitender Funktionen, die dem Rumpfgleichungssystem GS in der Umgebung δ_2 zugewiesen wird. Formal:

$$\mathbf{F}[\![\mathbf{program}\ prg \equiv \ldots\ \mathbf{end}]\!] = \mathbf{F}_{\delta_2}[\![GS[i], o]\!].$$

Der Eingangsstrom *i* ist dabei der Parameter von GS. Alle auf den rechten Gleichungsseiten von GS vorkommenden Ströme werden bis auf den Ausgabestrom *o* verborgen.

3.3 Kompositionalitätsresultate

Die in den vorangegangenen Abschnitten definierte funktionale Semantik ist kompositional und monoton (bzgl. der ⊆-Ordnung auf Funktionsmengen). Grob gesagt gelten also folgende Eigenschaften: Seien X,Y zwei AL-Konstrukte aus der gleichen syntaktischen Kategorie und cn[.] sei ein passender Kontext:

$\mathbf{F}[\![X]\!] \subseteq \mathbf{F}[\![Y]\!] \Rightarrow \mathbf{F}[\![cn[X]]\!] \subseteq \mathbf{F}[\![cn[Y]]\!]$,
(Monotonie)

$\mathbf{F}[\![X]\!] = \mathbf{F}[\![Y]\!] \Rightarrow \mathbf{F}[\![cn[X]]\!] = \mathbf{F}[\![cn[Y]]\!]$.
(Kompositionalität)

Die Bedeutung von Monotonie und Kompositionalität für die transformationelle Programmentwicklung ist schon in der Einleitung zu Abschnitt 3.2 diskutiert worden. Entscheidend ist, daß sie die lokale Anwendung von Transformationsregeln ermöglichen.
Streng formal lassen sich die einzelnen Monotonie- und Kompositionalitätsaussagen nicht genau gemäß dieser einfachen Schemata formulieren. Grund dafür ist die Tatsache, daß Kontexte häufig aus rekursiven Funktions- und Agentendeklarationen gebildet werden. Um in diesen Fällen Mengeninklusion bzw. -gleichheit auf den rechten Implikationsseiten zu erreichen, müssen die (größten) Fixpunkte der beteiligten Funktionale entsprechend korrelliert sein. Die Prämissen müssen dann teilweise schärfer formuliert werden.
Im einzelnen ergeben sich folgende Resultate. Angegeben sind jeweils nur Monotonieaussagen. Kompositionalitätsaussagen sind dual dazu. Man erhält sie, wenn man überall das $\subseteq$-Symbol durch das =-Symbol ersetzt. Ihre Gültigkeit folgt aus Symmetriegründen.

Wir nennen einen Ausdruck (eine Funktion, einen Agenten, usw.) X einen $\mathbf{F}_\delta$-*Abkömmling* von Y falls gilt:

$$\mathbf{F}_\delta[\![X]\!] \subseteq \mathbf{F}_\delta[\![Y]\!].$$

Die Monotonie von **F** erhält die Abkömmlingsbeziehung bei Einbettung von X und Y in geeignete Kontexte.

Satz 3.18 (Substitution von Ausdrücken in Ausdruckskontexten): Für beliebige Ausdrücke $E[y]$, $E_1[x]$, $E_2[x]$ und Umgebungen δ gilt:

$$\mathbf{F}_\delta[\![E_1[x]]\!] \subseteq \mathbf{F}_\delta[\![E_2[x]]\!] \Rightarrow \mathbf{F}_\delta[\![E[E_1[x]]]\!] \subseteq \mathbf{F}_\delta[\![E[E_2[x]]]\!].$$

Beweis: Seien $E'[y] \in DD(E[y]\tilde{})$ und $E_{11}, \ldots, E_{1n} \in DD(E_1[x]\tilde{})$. Wenn y n-mal in $E'[y]$ vorkommt, dann entstehe $E'[y_1, \ldots, y_n]$ aus $E'[y]$, indem jedes Vorkommnis von y durch eine disjunkte Kopie y_i ersetzt wird. Entsprechend ist $E'[E_{11}/y_1, \ldots, E_{1n}/y_n]$ definiert. Für beliebiges $\delta \in ENV$ gilt:

$$\begin{aligned}
& \mathbf{F}_\delta[\![E[E_1[x]]]\!] \\
= \quad & \{ \text{ Definition von } \mathbf{F} \} \\
& \{\lambda x.\mathbf{B}_{\delta',\sigma[x/x]}[\![E'[E_{11}/y_1, \ldots, E_{1n}/y_n]]\!] \mid \ldots \\
\} & \\
= \quad & \{ \text{ Lemma 3.6 } \} \\
& \{\lambda x.\mathbf{B}_{\delta',\sigma[\mathbf{B}_{\delta,\sigma[x/x]}[\![E_{1i}]\!]/y_i]}[\![E'[y_1, \ldots, y_n]]\!] \mid \ldots \} \\
= \quad & \{ \text{ Zusammenhang zwischen } \mathbf{B} \text{ und } \mathbf{F}, \text{ Lemma 3.14 } \} \\
& \{\lambda x.\mathbf{B}_{\delta',\sigma[f_i(x)/y_i]}[\![E'[y_1, \ldots, y_n]]\!] \mid f_i \in \mathbf{F}_\delta[\![E_1[x]]\!], \ldots \\
\} &
\end{aligned}$$

Analog für $\mathbf{F}_\delta[\![E[E_2[x]]]\!]$.
Aus $\mathbf{F}_\delta[\![E_1[x]]\!] \subseteq \mathbf{F}_\delta[\![E_2[x]]\!]$ folgt dann $\mathbf{F}_\delta[\![E[E_1[x]]]\!] \subseteq \mathbf{F}_\delta[\![E[E_2[x]]]\!]$.
□

Korollar 3.19: Für beliebige Ausdrücke $E[y]$, $E_1[x]$ und Umgebungen δ gilt:

$$\{f \circ f_1 \mid f \in \mathbf{F}_\delta[\![E[y]]\!], f_1 \in \mathbf{F}_\delta[\![E_1[x]]\!]\} \subseteq \mathbf{F}_\delta[\![E[E_1[x]]]\!]$$

Falls $|\mathbf{F}_\delta[\![E_1[x]]\!]| = 1$, sind beide Mengen gleich.

Beweis: Wähle im obigen Beweis für alle y_i die gleiche Funktion $f_1 \in \mathbf{F}_\delta[\![E_1[x]]\!]$
□

Ist $E_1[x]$ ein δ-Abkömmling von $E_2[x]$, dann kann $E_2[x]$ in jedem *Ausdruckskontext* durch $E_1[x]$ ersetzt werden, wobei sich die Abkömmlingsbeziehung auf die erweiterten Ausdrücke überträgt. Für *Funktions-* und *Agentenkontexte* reicht dies nicht aus.

Satz 3.20 (Substitution von Ausdrücken in Funktionskontexten): Für (reine) Objektausdrücke $E_1[x]$, $E_2[x]$ und Umgebungen δ gilt:

$$(\forall M: \mathbf{F}_{\delta[M/f]}[\![E_1[x]]\!] \subseteq \mathbf{F}_{\delta[M/f]}[\![E_2[x]]\!])$$
$$\Rightarrow \quad \mathbf{F}_\delta[\![\textbf{funct } f \equiv \mathbf{u}\, x \rightarrow \mathbf{v}\text{: } E_1[x] \textbf{ end}]\!] \subseteq \mathbf{F}_\delta[\![\textbf{funct } f \equiv \mathbf{u}\, x \rightarrow \mathbf{v}\text{: } E_2[x] \textbf{ end}]\!],$$

Beweis: Nach Definition gilt: $\mathbf{F}_\delta[\![\textbf{funct } f \equiv \mathbf{u}\, x \rightarrow \mathbf{v}\text{: } E_i[x] \textbf{ end}]\!] = \mathrm{FIX}(\tau_i)$ wobei

$$\tau_i: \wp(\mathbf{FCT}_S)\backslash\emptyset \rightarrow \wp(\mathbf{FCT}_S)\backslash\emptyset$$

mit $\tau_i.M = \text{strict } \mathbf{F}_{\delta[M/f]}[\![E_i[x]]\!]$.
Aus der Prämisse des Satzes folgt dann: $\forall M: \tau_1.M \subseteq \tau_2.M$ und daher $\mathrm{FIX}(\tau_1) \subseteq \mathrm{FIX}(\tau_2)$. □

Wie in Abschnitt 3.2.4 erklärt, definiert der Rumpf einer Funktionsdeklaration ein mengenwertiges Funktional, dessen größter Fixpunkt die Semantik der Deklaration ist. Damit zwei Deklarationen in Abkömmlingsrelation stehen, muß der größte Fixpunkt des einen Funktionals im größten Fixpunkts des anderen enthalten sein. Die in der Prämisse des Satzes formulierte Bedingung ist dafür hinreichend. Intuitiv gesprochen kann E_1 dann für E_2 (als Rumpf der Deklaration für *f*) substituiert werden, wenn E_1 unabhängig vom Wert der Umgebung an der Stelle *f* ein $\mathbf{F}_\delta$-Abkömmling von E_2 ist.
Gleiches gilt für die Substitution von Ausdrücken in Gleichungssystemen die den Rumpf von Agentendeklarationen bilden.

Satz 3.21 (Substitution von Ausdrücken in Agentenkontexten): Für Stromausdrücke S_j^1, S_j^2 und Umgebungen δ gilt:

$$(\forall M: \mathbf{F}_{\delta[M/f]}[\![S_j^1[i, s_1, \ldots, s_n]]\!] \subseteq \mathbf{F}_{\delta[M/f]}[\![S_j^2[i, s_1, \ldots, s_n]]\!])$$
$$\Rightarrow \quad \mathbf{F}_\delta[\![\textbf{agent } f \equiv \mathbf{v}\, i \rightarrow \mathbf{w}\, o\text{: } GS^1 \textbf{ end}]\!] \subseteq \mathbf{F}_\delta[\![\textbf{agent } f \equiv \mathbf{v}\, i \rightarrow \mathbf{w}\, o\text{: } GS^2 \textbf{ end}]\!],$$

wobei $GS^k = (s_1 \equiv S_1, \ldots, s_j \equiv S_j^k, \ldots, s_n \equiv S_n)$.

Beweis: Nach Definition gilt: $\mathbf{F}_\delta[\![\textbf{agent } f \equiv \textbf{v } i \to \textbf{w } o\text{: GS}^k \textbf{ end}]\!] = \text{FIX}(\tau_k)$ wobei

$$\tau_k\text{: } \wp(\mathbf{AGT}_s)\backslash\varnothing \to \wp(\mathbf{AGT}_s)\backslash\varnothing$$

mit $\tau_k.M = \text{strict } \mathbf{F}_{\delta[M/f]}[\![\text{GS}^k[i], o]\!]$. Es gilt:

$$\mathbf{F}_{\delta[M/f]}[\![\text{GS}^k[i]]\!] = \{\ \lambda i.\text{fix}\begin{pmatrix} s_1 = f_1(i, s_1, \ldots, s_n) \\ \ldots \\ s_j = f_j^k(i, s_1, \ldots, s_n) \\ \ldots \\ s_n = f_n(i, s_1, \ldots, s_n) \end{pmatrix} \mid \ldots\ \},$$

wobei $f_1 \in \mathbf{F}_{\delta[M/f]}(S_1)$, …, $f_n \in \mathbf{F}_{\delta[M/f]}(S_n)$ und $f_j^k \in \mathbf{F}_{\delta[M/f]}[\![S_j^k]\!]$. Aus der Voraussetzung folgt dann:

$$\forall M\text{: } \mathbf{F}_{\delta[M/f]}[\![\text{GS}^1[i]]\!] \subseteq \mathbf{F}_{\delta[M/f]}[\![\text{GS}^2[i]]\!]$$

und damit $\forall M\text{: } \tau_1.M \subseteq \tau_2.M$. Daraus folgt wie oben $\text{FIX}(\tau_1) \subseteq \text{FIX}(\tau_2)$.

□

Bei der Berechnung von $\mathbf{F}_\delta[\![X]\!]$ wird durch den Umgebungsparameter δ Kontextinformation bereitgestellt. Die Bedeutung von X hängt von dieser Information (zumindest teilweise) ab. Wie die beiden obigen Sätze zeigen, darf die Kontextinformation aber nicht in jedem Fall ausgenutzt werden: Ist X ein geschlossenes System von Deklarationen für $f_1, \ldots, f_n$ und soll ein Rumpf E_i durch E_i' ersetzt werden, so muß man die Äquivalenz von E_i und E_i' unabhängig von den Umgebungswerten an den Stellen $f_1, \ldots, f_n$ nachweisen.

Vollständig ausgenutzt werden kann die in δ gespeicherte Kontextinformation, wenn im Gleichungsteil vollständiger Programme Substitutionen durchgeführt werden. Sei DK ein geschlossenes System von Deklarationen:

$$\textbf{funct } f_1 \equiv F_1, \ldots, \textbf{funct } f_n \equiv F_n,$$
$$\textbf{agent } g_1 \equiv G_1, \ldots, \textbf{agent } g_m \equiv G_m.$$

Eine Umgebung δ heißt *konsistent* mit DK, wenn gilt:

$$\mathbf{F}_\delta[\![DK]\!] = (\delta(f_1), \ldots, \delta(f_n), \delta(g_1), \ldots, \delta(g_m)).$$

Eine konsistente Umgebung ordnet jedem f_i bzw. g_i genau die Funktionsmenge zu, die DK festlegt. Beachte: Formal ist $\mathbf{F}_\delta[\![DK]\!]$ von δ unabhängig.

Satz 3.22 (Substitution von Ausdrücken in Programmkontexten): Sei DK ein geschlossenes System von Deklarationen und δ eine mit DK konsistente Umgebung. Dann gilt:

$$\mathbf{F}_\delta[\![S_j^1[i, s_1, \ldots, s_n]]\!] \subseteq \mathbf{F}_\delta[\![S_j^2[i, s_1, \ldots, s_n]]\!]$$
$$\Rightarrow \quad \mathbf{F}[\![\textbf{program } prg \equiv \mathbf{v}\ i \rightarrow \mathbf{w}\ o\text{: DK, } GS^1 \ \textbf{end}]\!]$$
$$\subseteq \mathbf{F}[\![\textbf{program } prg \equiv \mathbf{v}\ i \rightarrow \mathbf{w}\ o\text{: DK, } GS^2 \ \textbf{end}]\!],$$

wobei $GS^k = (s_1 \equiv S_1, \ldots, s_j \equiv S_j^k, \ldots, s_n \equiv S_n)$.

Beweis: Analog zu Satz 3.21

□

Die obigen Sätze betreffen die Substitution von Ausdrücken in verschiedenen Kontexten. Die folgende Aussage bezieht sich auf den Substitution von Deklarationen:

Satz 3.23 (Substitution von Deklarationen): Sei DK ein geschlossenes System von Deklarationen

funct $f_1 \equiv F_1, \ldots,$ **funct** $f_n \equiv F_n$,
agent $g_1 \equiv G_1, \ldots,$ **agent** $g_m \equiv G_m$,

und DK' das geschlossene System (k, l ≥ 0)

funct $f_1 \equiv F_1', \ldots,$ **funct** $f_{n+k} \equiv F_{n+k}'$,
agent $g_1 \equiv G_1', \ldots,$ **agent** $g_{m+l} \equiv G_{m+l}'$.

Wenn für je zwei Umgebungen δ, δ' mit der Eigenschaft:

δ ist konsistent mit DK ∧ δ' ist konsistent mit DK',

gilt:

$\forall h \in \{f_1, \ldots, f_n, g_1, \ldots, g_m\}: \delta(h) \subseteq \delta'(h)$.

Dann folgt:

$$\mathbf{F}[\![\textbf{program } prg \equiv \mathbf{v}\ i \rightarrow \mathbf{w}\ o\text{: DK, GS } \textbf{end}]\!]$$
$$\subseteq \mathbf{F}[\![\textbf{program } prg \equiv \mathbf{v}\ i \rightarrow \mathbf{w}\ o\text{: DK', GS } \textbf{end}]\!]$$

Beweis: Definitionsgemäß gilt

$$\mathbf{F}[\![\textbf{program } prg \equiv \mathbf{v}\ i \rightarrow \mathbf{w}\ o\text{: DK, GS } \textbf{end}]\!] = \mathbf{F}_\delta[\![GS[i], o]\!]$$
$$\mathbf{F}[\![\textbf{program } prg \equiv \mathbf{v}\ i \rightarrow \mathbf{w}\ o\text{: DK', GS } \textbf{end}]\!] = \mathbf{F}_{\delta'}[\![GS[i], o]\!]$$

wobei δ eine beliebige mit DK und δ' eine mit DK' konsistente Umgebung ist. Nach Voraussetzung gilt: $\delta(h) \subseteq \delta'(h)$ für $h \in \{f_1, \ldots, f_n, g_1, \ldots, g_m\}$. Da in GS nur diese Funktions- und Agentenbezeichner vorkommen (Kontextkorrektheit!), kann man o.B.d.A ein δ und ein δ' so wählen, daß gilt: $\delta \subseteq \delta'$. Dann folgt die Behauptung aus der Umgebungsmonotonie von **F**.

□

Aufgrund dieser Aussage ist möglich, Funktions- bzw. Agentendeklarationen im Kontext vollständiger Programme durch Abkömmlinge bzw. äquivalente Definitionen zu ersetzen. Dabei dürfen zusätzliche Hilfsfunktionen eingeführt werden. Die umgekehrte Aussage besteht trivialerweise darin, nicht benötigte Definitionen ersatzlos zu streichen. Dies trägt zur Vereinfachung von Programmen bei.

3.4 Nichtdeterminismus

Mit dem Auswahloperator $[]$ gibt es in AL ein explizit nichtdeterministisches Konstrukt, das die kontextfreie Auswahl unter einer endlichen Anzahl, gleichberechtigt nebeneinander stehender Möglichkeiten erlaubt (*"finite choice"*). Die Breitensemantik **B** beschreibt dies durch Mengenvereinigung:

$$\mathbf{B}[\![E_1 [] E_2]\!] = \mathbf{B}[\![E_1]\!] \cup \mathbf{B}[\![E_2]\!].$$

Operationell verbindet sich damit folgende Vorstellung: Steht der nichtdeterministische Ausdruck $E_1 [] E_2$ zur Auswertung an, so wird genau einer der Teilausdrücke ausgewählt und tatsächlich behandelt, der andere aber ignoriert. Die Entscheidung welcher Kandidat zum Zuge kommt, fällt ohne Rücksicht auf die Eigenschaften der E_i, insbesondere ohne Rücksicht auf ihr Terminierungsverhalten. Diese Spielart des Nichtdeterminismus heißt *erratisch*. Wählt man andere Auswertungsstrategien so ergeben sich andere Nichtdeterminismusformen (vgl. [Broy 85], [Hussmann 91]):

Die dem *dämonischen* Nichtdeterminismus entsprechende (operationelle) Strategie ($\ddagger$ symbolisiere den zugehörigen Operator) wertet alle Teilausdrücke vollständig aus und wählt dann irgendeines der berechneten Ergebnisse. Demzufolge terminiert die Auswertung von $E_1 \ddagger E_2$ dann und nur dann, wenn die Auswertung bieder E_i terminiert. $\ddagger$ läßt sich denotationell leicht beschreiben:

$$\mathbf{B}[\![E_1 \ddagger E_2]\!] = \backslash B \backslash LC \backslash \{(\backslash A \backslash AL(\{\bot\}$$
$$\text{falls } \bot \in \mathbf{B}[\![E_1]\!] \cup \mathbf{B}[\![E_2]\!]; \mathbf{B}[\![E_1]\!] \cup \mathbf{B}[\![E_2]\!] \qquad \text{sonst})).$$

Die Strategie des *angelischen* Nichtdeterminismus wertet die E_i parallel (oder "dovetail-artig" sequentiell) aus und liefert als Ergebnis den Wert desjenigen Teilausdrucks, dessen Auswertung als erstes beendet werden kann. Während der dämonische Nichtdeterminismus sich besonders "bösartig" verhält, ist der angelische Operator ∇ besonders gutartig, eben "engelsgleich": Die Auswertung von $E_1 \nabla E_2$ terminiert schon dann, wenn nur ein endgültig E_i ausgewertet werden kann. Die denotationelle Semantik von ∇ ist wie folgt definiert:

$$\mathbf{B}[\![E_1 \nabla E_2]\!] = \mathbf{B}[\![E_1]\!] \backslash \{\bot\} \cup \mathbf{B}[\![E_2]\!] \backslash \{\bot\} \cup \begin{cases} \{\bot\} & \text{falls } \bot \in \mathbf{B}[\![E_1]\!] \cap \mathbf{B}[\![E_2]\!] \\ \varnothing & \text{sonst} \end{cases}.$$

Der angelische Operator ist ein sehr mächtiges Konstrukt. Im Zusammenhang mit Rekursion induziert er *unbeschränkten Nichtdeterminismus* (*infinite choice*). Mit Hilfe von ∇ ist es in Broy´s AMPL beispielsweise möglich, eine Funktion

funct *any* ≡ **nat** *n* → **nat**: *n* ∇ *any*(*n*) **end**

zu definieren, die angewandt auf 0 eine beliebige natürliche Zahl, niemals jedoch $\perp$ liefert (vgl. [Broy 86]). AL bietet diese Möglichkeit nicht; die erratische Version von *any*

funct *any* ≡ **nat** *n* → **nat**: *n* [] *any*(*n*) **end**

schließt Divergenz nicht aus.

Trotz der knappen denotationellen Definition von ∇ ist die semantische Behandlung des angelischen Nichtdeterminismus problematisch. Der technische Grund hierfür ist in den (im Gegensatz zu []) fehlenden Distributionseigenschaften von ∇ zu sehen. Diese schlugen sich bei der Definition der Breitensemantik in Lemma 3.5 nieder und waren Voraussetzung für die Möglichkeit aus **B** die funktionale Semantik **F** abzuleiten. Aus dem Fehlen dieser Eigenschaften ergibt sich die entscheidende Konsequenz, daß die Semantik von Funktions- bzw. Agentendeklarationen, die auf ∇ abgestützt sind, *nicht* durch Mengen stetiger Stromfunktionen erklärt werden kann. Zur genaueren Analyse soll das wohl prominenteste Beispiel dienen:

Beispiel 3.24 (Mischen): In vielen Datenflußanwendungen (siehe Abschnitt 3.5) werden *Mischknoten* verwendet, die zwei oder mehr Eingabeströme zu einem Ausgabestrom zusammenmischen. Der einfachste Mischagent hat folgende AL-Darstellung:

agent *merge* ≡ **chan u** *a*, *b* → **chan u** *c*:
c ≡ **if isempty.***a* **then** *b* **else ft.***a* & *merge*(**rt.***a*, *b*) **fi**
[] **if isempty.***b* **then** *a* **else ft.***b* & *merge*(*a*, **rt.***b*) **fi**
end

Die Semantik von *merge* wird durch die folgende Menge stetiger Funktionen beschrieben:

MERGE = {merge | $\exists s \in \{0, 1\}^{\infty}$: $\forall a,b \in \mathbf{U}^{\omega}$: merge(a,b) = sched(a,b,s)},

wobei sched: $\mathbf{U}^{\omega} \times \mathbf{U}^{\omega} \times \mathbf{Nat}^{\omega} \rightarrow \mathbf{U}^{\omega}$ durch:

sched(a,b,ε) = ε,
sched(a,b,⊥) = ⊥,

sched(⊥,b,0&s) = ⊥,
sched(a,⊥,1&s) = ⊥,
sched(ε,b,0&s) = b,
sched(a,ε,1&s) = a,

sched(x&a,b,0&s) = x&sched(a,b,s), sched(a,x&b,1&s) = x&sched(a,b,s),

axiomatisiert ist. *merge* beschreibt einen Mischknoten, von dessen Arbeitsweise man sich folgende Vorstellung machen kann:
Bevor der Agent seine Eingaben bearbeitet, wählt er einen beliebigen, aber festen *Scheduler* $s \in \{0, 1\}^\infty$, dessen Vorgaben er von da an folgt. Eine 0 im Schedulerstrom zeigt an, daß das nächste Element der linken Eingabe verarbeitet werden soll, eine 1 verweist auf die rechte Eingabe. Für s ist jede beliebige unendliche 0-1-Sequenz zulässig, z.B. auch 0^∞, die den rechten Eingabestrom völlig ignoriert. "Wählt" *merge* diesen Scheduler, so verhält es sich "unfair". Formal heißt ein Scheduler s *(unendlich)-fair*, wenn er sowohl unendlich viele 0en als auch unendlich viele die 1en enthält. Faire Scheduler liefern *(unendlich-)faire Instanzen* des Mischknotens. Diese sind dadurch gekennzeichnet, daß für unendliche Ströme $a,b \in \mathbf{U}^\infty$ jedes Element der Eingabe auch als Ausgabe auftritt. Ein Agent, der nur unendlich-faire Instanzen besitzt ("infinity-fair merge", vgl. [Park 82], [Panangaden, Stark 88]), ist in AL nicht in voller Allgemeinheit darstellbar. Implementierungen (d.h. Abkömmlinge im Sinne von Abschnitt 3.3) lassen sich jedoch trivialerweise angeben, zum Beispiel:

agent *fmerge* ≡ **chan u** *a, b* → **chan u** *c*:
c ≡ **if isempty.***a* **then** *b*
else if isempty.*b* **then** *a*
else ft.*a* & **ft.***b* & *fmerge*(**rt.***a*, **rt.***b*)
▯ **ft.***b* & **ft.***a* & *fmerge*(**rt.***a*, **rt.***b*) **fi fi**
end

Da die Wahl des Schedulers s *unabhängig* von den Eingaben erfolgt, kann die Situation eintreten, daß s dem Agenten vorschreibt, einen Eingabestrom zu bearbeiten, obwohl dieser undefiniert (⊥) ist, während auf dem anderen noch Daten verfügbar sind. Wir bezeichnen einen Mischagenten, der diese Eigenschaft hat, als *strikt* und umgekehrt einen Agenten, der mit endlichen, partiellen Strömen so umgehen kann, daß diese Situationen nicht eintritt, als *nicht-strikt* oder *angelisch*. Der zu *merge* duale Agent modelliert das angelisches Mischen ("angelic merge").

agent *amerge* ≡ **chan u** *a, b* → **chan u** *c*:
c ≡ **if isempty.***a* **then** *b* **else ft.***a* & *amerge*(**rt.***a*, *b*) **fi**
∇ **if isempty.***b* **then** *a* **else ft.***b* & *amerge*(*a*, **rt.***b*) **fi**
end

Die oben angegebene Menge MERGE ist keine taugliche Semantik für *amerge*: Um die spezifischen Eigenschaften von ∇ richtig wiederzugeben, müßte man folgende Menge wählen:

AMERGE = {amerge | ∃merge ∈ MERGE: ∀ a,b ∈ $\mathbf{U}^\omega$:
amerge(a,b) = merge(a,b) ∧ #amerge(a,b) = #a + #b}

Das erste Konjunktionsglied drückt dabei die Mischeigenschaft des Agenten aus, das zweite seine Nichtstriktheit (#a bezeichne die Länge des Stroms a). Für ein beliebiges amerge$\in$ AMERGE läßt sich ableiten:

$$\text{amerge}(x\&\bot,\bot) = x\&\bot,$$
$$\text{amerge}(\bot,y\&\bot) = y\&\bot,$$
$$\text{amerge}(x\&\bot,y\&\bot) = x\&y\&\bot \vee \text{amerge}(x\&\bot,y\&\bot) = y\&x\&\bot.$$

Es gibt jedoch keine präfixmonotone Abbildung, die diesen Axiomen genügt. Also bestätigt sich die oben gemachte Aussage:

Die Semantik des angelischen Mischen ist durch eine Menge stetiger Stromfunktionen nicht beschreibbar.

Nicht-monotone Funktionen wollen wir als Semantikbeschreibung für ein algorithmisches Konstrukt aus naheliegenden Gründen ausschließen (vgl. Kapitel 2, S. 18). Wie ein Ausweg aus diesem Dilemma aussehen könnte, zeigt folgende Analyse:

In Anlehnung an die Definitionen aus Kapitel 2 heiße ein merge$\in$MERGE *partiell korrekt* oder *sicher*, wenn ein amerge$\in$ AMERGE existiert, so daß gilt: merge $\sqsubseteq$ amerge.
Gilt darüber hinaus für zwei Eingaben a,b$\in \mathbf{U}^\omega$: merge(a,b) = amerge(a,b), dann heiße merge *total korrekt* oder *lebendig* für a,b$\in \mathbf{U}^\omega$.
Aus den Definitionen von MERGE und AMERGE folgt nun:

(1) Jedes merge$\in$ MERGE ist partiell korrekt.

(2) Zu allen Eingaben a, b gibt es ein total korrektes merge$\in$ MERGE.

Hierin liegt der Schlüssel zur Lösung: Das Verhalten des angelischen Mischknotens wird korrekt wiedergegeben, wenn aus dem Reservoir aller monotonen Mischfunktionen, ein Repräsentant ausgewählt wird, der nicht nur sicher ist, sondern auch zu den aktuellen Eingaben paßt, also die dafür geforderten Lebendigkeitseigenschaften garantiert. Die Bedeutung von *amerge* kann daher durch eine Menge von Tupeln

$$(\text{merge},a,b) \in [\mathbf{U}^\omega \times \mathbf{U}^\omega \to \mathbf{U}^\omega] \times \mathbf{U}^\omega \times \mathbf{U}^\omega$$

angemessen repräsentiert werden (vgl dazu die "input choice specifications" in [Broy 90]):

$$\text{AMERGE'} = \{(\text{merge},a,b) \mid \text{merge} \in \text{MERGE} \wedge \#\text{merge}(a,b) = \#a + \#b\}$$

(merge,a,b)$\in$ AMERGE' ist äquivalent zu der Aussage: merge *ist eine für* a *und* b *zulässige Wahl, es erzeugt eine für diese Eingaben total korrekte Ausgabe*.

Diese verkomplizierte Semantikbeschreibung wird im folgenden keine weitere Rolle mehr spielen (s.u.). Sie ist hier nicht zuletzt deshalb angegeben, um zu zeigen, daß eine denotationelle Behandlung anderer Nichtdeterminismusformen im Prinzip möglich wäre.

Man beachte, daß der mit Hilfe von ∇ definierte Agent *amerge* zwar angelisch, aber nicht fair ist: Für $a \in \mathbf{U}^{\infty}$ und $b \in \mathbf{U}^{\omega}$ gibt es ein Tupel (merge,a,b)$\in$ AMERGE', für das gilt: merge(a,b) = a. Das klassische *nicht-strikte, faire Mischen*, das in gewisser Weise an der Spitze einer Hierarchie unterschiedlich ausdrucksstarker, nichtdeterministischer Agenten steht (vgl. [Russell 89]), ist auch mit ∇ nicht programmierbar[2]. □

Einem nichtdeterministischen Ausdruck E kann in der Regel kein eindeutiger Wert (bzw. $\perp$ im Fall der Divergenz), sondern nur eine Menge möglicher Auswertungsergebnisse zugeordnet werden. Unterschiedliche Vorkommen von E werden unabhängig behandelt und liefern potentiell unterschiedliche Resultate. Die Auswertung von

$$1 \sqcap 2 + 1 \sqcap 2$$

ergibt 2 oder 4 oder 3, je nachdem, ob in beiden Fällen gleiche oder verschiedene Werte ausgewählt werden. Eine herausragende Eigenschaft ("rein") applikativer Sprachen ist die sog. *Werttreue* (engl. *"referential transparency"*). Dieser Begriff kennzeichnet die Tatsache, daß der Zusammenhang zwischen Ausdruck und denotiertem Wert invariant ist; derselbe Ausdruck bezeichnet stets und an allen Stellen denselben Wert. Implizite, globale Zustandsparameter, die das Ergebnis von Berechnungen beeinflussen und selber über Seiteneffekte beeinflußt werden, sind in applikativen Sprachen ausgeschlossen. Auch AL ist seiteneffektfrei, der vorhandene Nichtdeterminismus führt aber offenbar zum Verlust der Werttreue. Damit verlieren eine ganze Reihe von vertrauten Regeln, die die Gleichheit von Ausdrücken betreffen, ihre Gültigkeit. Zum Beispiel sind die Ausdrücke

$$E = 1 \sqcap 2 + 1 \sqcap 2 \qquad \text{und} \qquad E' = 2 * 1 \sqcap 2$$

nicht äquivalent, den offensichtlich gilt: $\mathbf{B}[\![E]\!] = \{2, 3, 4\} \neq \{2, 4\} = \mathbf{B}[\![E']\!]$.
Darüber hinaus ergibt sich ein zusätzliches Phänomen, das mit den Parameterübergabemechanismen bei Funktionsaufrufen in Verbindung steht. Für werttreue Sprachen ist bekannt, daß die Auswertung eines Funktionsaufrufes $f(E_1, \ldots, E_n)$ gemäß *call-by-value* entweder nicht terminiert (also $\perp$ liefert) oder dasselbe Resultat liefert wie bei *call-by-name* Übergabe der Parameter. Im Falle von Nichtdeterminismus ist dies nicht mehr richtig. Sei die Funktion *double* wie folgt definiert

funct *double* ≡ **nat** $n \rightarrow$ **nat**: $x + x$ **end**

Dann liefert *double*(1 ⊓ 2) gemäß call-by-value ein Element aus {2, 4} und gemäß call-by-name ein Element aus {2, 3, 4}. Call-by-value übergibt tatsächlich nur einen Wert (*call-time-choice*) an die aufgerufene Funktion, call-by-name jedoch semantisch betrachtet die gesamte Breite des

---

[2] Mit "input choice specifications" kann die Semantik des nicht-strikten fairen Mischens angegeben werden: NFMERGE = {(merge,a,b) | $\exists s \in \{0, 1\}^{\infty}$: merge(a,b) = sched(a,b,s) $\wedge$ #0©s = #a $\wedge$ #1©s = #b}. 0©s steht dabei für die Filteroperationen, die aus s alle 1en herausfiltert. Analog für 1©s.

Ausdrucks (*run-time-choice*) (vgl. [Clinger 82], [Berghammer et al. 90])[3]. Bei call-by-value-Übergabe ist die übliche Auffaltungsregel (*"fold-unfold"*, β-Reduktion im Lambda-Kalkül, vgl. [Barendregt 90]) nicht mehr gültig. Eine Diskussion von Aufrufmechanismen findet sich in [Broy 86], S. 29-30.

Die Semantik eines nichtdeterministischen Agenten *f* wird durch eine Menge determinierter Instanzen (Funktionen) beschrieben, von denen jede ein mögliches Verhaltensmuster, d.h. die Reaktion von *f* auf alle denkbaren Eingaben, beschreibt. Alle Instanzen sind gleichberechtigt und als deterministische Implementierungen (Abkömmlinge) zulässig. Der Nichtdeterminismus ist damit streng lokal (*"internal choice*"): Unabhängig von der Umgebung in die ein Agent eingebettet ist, stehen ihm eine Reihe von Verhaltensweisen zur Verfügung unter denen er frei auswählen kann. Die Umgebung hat keine Möglichkeit, diese Wahl in irgendeiner Weise zu beeinflussen. Damit unterscheidet sich dieser Ansatz von CCS (vgl. [Milner 80]), CSP (vgl. [Hoare 85a]) und anderen synchronen Formalismen, die der Umgebung des Agenten das Recht einräumen, Auswahlentscheidungen durch Kommunikationsaktionen zu beeinflussen (vgl. [Hoare 85a], S. 101-111). Obwohl die Terminologie in meinen Augen etwas mißverständlich ist, ist hierfür auch der Begriff externer Nichtdeterminismus (*"external (general) choice"*) gebräuchlich.
Für das dynamische Verhalten eines AL-Agenten sind zwei operationelle Vorstellungen möglich:

- Während des Ablaufs entscheidet sich der Agent an jeder im Programmtext durch ▯ gekennzeichneten Auswahlstelle für eine der angebotenen Alternativen.
- Zu Beginn des Ablaufes werden alle Entscheidungen im voraus fixiert und von da ab nicht mehr verändert.

Das Konzept des internen Nichtdeterminismus gewährleistet, daß es extensional betrachtet keinen Unterschied macht, welcher Vorstellung man den Vorzug gibt. Operationelle Semantiken stützen sich üblicherweise auf die erste Variante (siehe z.B. [Plotkin 83]), während die denotationelle Beschreibung von AL auf der zweiten Auffassung beruht.

Insgesamt zeigt die obige Diskussion, daß Nichtdeterminismus in seinen verschiedenen Ausprägungen die Komplexität einer Sprache deutlich erhöht. Es stellt sich daher die Frage, ob er notwendig ist, oder ob nicht besser zugunsten konzeptueller Einfachheit darauf verzichtet werden sollte. Der Entwurf von AL repräsentiert (meines Erachtens) einen Kompromiß in dieser Frage: Der angelische Operator ∇ ist in AL nicht verfügbar. Dadurch wird eine kompliziertere Semantikbeschreibung ("input choice", s.o.) vermieden. Zudem würde die Verwendung von ∇ in AL die Einführung zeitsensitiver Konstrukte wie disjunktives Warten, nicht-blockierendes Lesen oder Kanalabfragen ("polling") auf der prozeduralen Ebene notwendig machen.

---

[3] In manchen Arbeiten (z.B. [Berghammer et al. 90]) werden call-by-name, call-by-value einerseits und call-time-choice, run-time-choice anderseits als orthogonal zueinander angesehen. Tatsächlich lassen sich für alle vier Kombinationsmöglichkeiten geeignete denotationelle Definitionen angeben. Clinger spricht im Falle von run-time-choice von einer *pluralen Semantik*, bei der Wertemengen, und im Falle von call-time-choice von einer *singulären Semantik*, bei der nur einzelne Werte übergeben werden.

Auf der anderen Seite kann der erratische Operator verwendet werden. Mit seiner Hilfe ist es möglich, Entwurfsentscheidungen in kontrollierter Weise offen zu halten und unnötige Festlegungen zu vermeiden. Dies ist besonderes wichtig, wenn man berücksichtigt, daß AL als Beschreibungsmittel in den Rahmen der FOCUS-Methodik eingepaßt ist. Wird ein System (oder eine Komponente davon) gemäß dieser Methodik entwickelt, so ist die erste AL-Version zugleich auch dessen erste Darstellung in einer formal definierten, algorithmischen Sprache mit fester Syntax. Dieses "abstrakte Programm" repräsentiert zumeist noch nicht die angestrebte, "konkrete" Endversion, bis zu der noch weitere Entwicklungsschritte notwendig sind. Die vorliegende Arbeit beschäftigt sich gerade damit. Die methodische Regel, unnötige Festlegungen zu vermeiden, gilt auch für diesen Abschnitt des Entwicklungsprozesses. Der weiteren Implementierung wird so nicht vorgegriffen, es bleiben Freiräume, die flexibel ausgenutzt werden können. Nichtdeterminismus ist eine Möglichkeit, dieser Anforderung gerecht zu werden.

3.5 AL-Programme und Agentennetze

AL-Programme lassen sich auf sehr direkte Art zu Netzen lose gekoppelter, asynchron miteinander kommunizierender Agenten in Beziehung. Solche Netze werden seit Beginn der 70er Jahre von zahlreichen Autoren untersucht (vgl. [Dennis 74], [Kahn 74], [Kahn, MacQueen 77], [Wadge, Ashcroft 85], [Broy 86, 87b], [Glasgow, MacEwen 89], u.v.a.). Sie dienen der funktionalen Beschreibung von Betriebssystemstrukturen (vgl. [Jones, Sinclair 89], [Turner 90]) und bilden die konzeptuelle Grundlage für ein eigenständiges Programmiermodell, das *Datenflußparadigma*, sowie für neuere Rechnerarchitekturen, die auf dieses Modell hin ausgerichtet sind. Ein Überblick über verschiedene Varianten des Modells findet sich in [Dennis 85] und (sehr kompakt) in [Sharp 87]. Ausgewählte Datenflußarchitekturen werden in [Herath et. al. 88] sowie in [Duncan 90] beschrieben.

Ein klassisches Datenflußprogramm – Dennis nennt es "data flow schema" – ist ein gerichteter Graph, bestehend aus untereinander verbundenen Akteuren. Über die Kanten fließen Datenobjekte, sogenannte "Token" oder "Datons", die von den Akteuren durch Fortschalten verarbeitet werden. Ein Akteur ist schaltbereit, falls auf allen seinen Eingangskanten Token vorhanden sind. Schaltet er, so werden diese Token abgezogen und neue, das Resultat der Berechnung repräsentierende Token auf die Ausgangskanten abgelegt. Dies ist dieselbe Schaltregel, die auch zur Erklärung des dynamischen Verhaltens von *Petrinetzen* verwendet wird (vgl. [Reisig 85]). Da in einem Zustand, d.h. bei gegebener Tokenverteilung, mehrere Akteure unabhängig von einander schaltfähig seien können, ergeben sich unabhängige Teilberechnungen, die parallel fortschreiten können. Neben den üblichen Reduktionstechniken ("string reduction", "graph reduction", vgl. [Peyton Jones 87]) ist das Datenflußmodell eine weitere Möglichkeit zur Implementierung *funktionaler Sprachen* (vgl. z.B. Kapitel 14 in [Field, Harrison 88]). Durch die Übersetzung von Funktionsausdrücken in Datenflußgraphen und deren Auswertung gemäß der skizzierten Regel wird die inhärent vorhandene Möglichkeit zur parallelen Auswertung funktionaler Programme ausgenutzt.

Wir sprechen in dieser Arbeit nicht von *Datenflußschemata*, sondern von *Agentennetzen*. Bei rein technischer Betrachtung gibt es zwischen beiden Begriffen kaum einen Unterschied. Methodisch und konzeptuell verbinden sich damit jedoch durchaus verschiedene Vorstellungen:

Spezielle Datenflußsprachen wie LUCID (vgl. [Wadge, Ashcroft 85]), VAL (vgl. [McGraw 82]) oder ID (vgl. [Hutner, Holzner 89]) sind auf angepaßte Architekturen ausgerichtet. Diese

ermöglichen zwar parallele Programmverarbeitung mit Hilfe mehrerer sog. "processing elements" (PE's), gelten jedoch im Sinne der z.B. in [Bal et al. 89] angegebenen Klassifikation nicht als verteiltes (Multicomputer-) System: Die PE's sind durch ein "Routing"-Netz eng miteinander verkoppelt und haben häufig auch Zugriff auf einen gemeinsamen Speicher.
Datenflußschemata beschreiben darüber hinaus eine sehr feinkörnige Parallelität. Die einzelnen Akteure sind in der Regel einfache Basisfunktionen deren komplexes Zusammenspiel umfangreichere Berechnungen realisiert.
Agentennetze modellieren verteilte Systeme, die aus einer Menge miteinander kommunizierender Einheiten bestehen, für die wir die Bezeichnung Agent verwenden (vgl. die Fußnote in Abschnitt 1.1). Der gesamte Entwicklungsprozeß zielt darauf ab, ein solches System zu realisieren. Und zwar nicht nur, um durch Parallelisierung Effizienzsteigerungen zu erreichen, sondern auch, weil manche Problemstellungen verteilte Realisierungen erzwingen. Dabei ist mit "verteilt" die Implementierung auf einer Rechnerkonfiguration gemeint, deren Komponenten durchaus auch räumlich getrennt seien können. Dies hat natürlich Auswirkungen auf die Granularität der Parallelität, die in der Regel gröber seien dürfte als bei klassischen Datenflußanwendungen. Ein Agent in einem Netz kann umfangreiche interne Berechnungen ausführen, ohne Nachrichten von seinen Partnern zu empfangen oder an diese zu senden. Möglicherweise wird er durch ein komplexes sequentielles Programm implementiert, das nur nur in großen Intervallen kommuniziert.

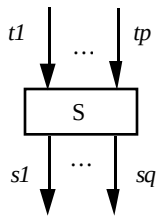
Ein Agentennetz ist ein *gerichteter Graph*. Seine Knoten repräsentieren die parallel arbeitenden Agenten, seine Kanten die Kommunikationslinien (Kanäle) zwischen den Agenten. Auf eine formale Graphdefinition soll an dieser Stelle verzichtet werden. Wir machen den Zusammenhang zwischen AL-Programmen und Agentennetzen exemplarisch klar. Sei

program *prg* ≡ **chan v** $i_1, \ldots, i_n \rightarrow$ **chan w** $o_1, \ldots, o_m$:
DK, GS
end

ein AL-Programm. Dann repräsentiert jede in GS vorkommende Gleichung

$s_1, \ldots, s_q = S[t_1, \ldots, t_p]$

einen Knoten des zugehörigen Netzes:

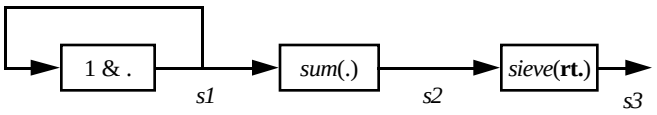


Figur 3.1: Netzdarstellung einer Gleichung

Die Ausgangskanten $s_1, \ldots, s_q$ werden durch die linksseitig vorkommenden Strombezeichner bestimmt, die Eingangskanten $t_1, \ldots, t_p$ durch die Strombezeichner, die in S vorkommen. Dabei steht jedes t_i für genau ein Vorkommnis eines Bezeichners *t*. Tritt *t* mehrfach auf definiert jedes Vorkommnis eine Eingangskante.
Ein Strombezeichner *s*, der auf der linken Seite der i-ten Gleichung und auf der rechten Seite der j-ten Gleichung vorkommt, definiert eine Kante von Knoten i zu Knoten j. Da rekursive

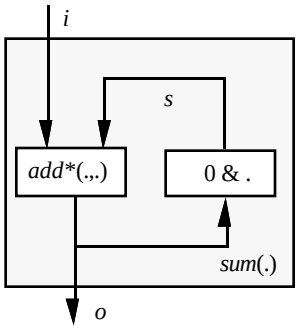
Stromgleichungen zulässig sind kann i gleich j sein. So entstehen direkte Rückkopplungsschleifen (Schlingen). Die in der Kopfleiste des Programms angegebenen Eingabeströme definieren die Eingangskanten des gesamten Netzes, die Ausgabeströme seine Ausgangskanten. Aufgrund der Kontextbedingungen für AL-Programme (vgl. Abschnitt 3.1) besitzt jede Kante eine eindeutige "Quelle": Entweder ist dies ein Knoten im Graphen oder die "Umgebung", falls es sich um eine Eingangskante handelt. Das "Ziel" einer Kante braucht dagegen nicht eindeutig sein: Derselbe Strombezeichner kann in mehreren Gleichungen verwendet werden. Die entsprechende Kante zerfasert dann in mehrere Äste, die an unterschiedlichen Knoten enden.

Beispiel 3.25 (Netzdarstellung des Erathostenes-Programms): Das Erathostenes Programm aus Beispiel 3.4 entspricht folgendem piplineartigen Agentennetz:



Figur 3.2: Netzdarstellung des Erathostenes Programms

Da auch Agenten durch Gleichungssystemen definiert werden, lassen sie sich ebenfalls durch Netze darstellen. Der Agent *sum* wird wie folgt repräsentiert:



Figur 3.3: Netzdarstellung des Agenten *sum*

□

Beispiel 3.26 (Iteriertes while): Das folgende Programm ist die Datenflußrealisierung einer **while**-Schleife, der iteriert von außen Daten zugeführt werden können.

program *while** ≡ **chan chan v** $i \rightarrow$ **chan v** o:

agent *inswitch* ≡ **chan bool** b, **chan v** $i_1, i_2 \rightarrow$ **chan v** o:

$o \equiv$ **if isempty.**b **then** ε

else if ft.$b \wedge$ **isempty.**i_1 **then** ε

else if ft.$b \wedge \neg$**isempty.**i_1 **then ft.**i_1 & *inswitch*(**rt.**b, **rt.**i_1, i_2)
else if $\neg$**ft.**$b \wedge$ **isempty.**i_2 **then** ε
else ft.i_2 & *inswitch*(**rt.**b, i_1, **rt.**i_2) **fi fi fi fi**

end,

agent *switch* ≡ **chan bool** b, **chan v** $i \rightarrow$ **chan v** o_1, o_2:
$o_1 \equiv$ *gate*(true, b, i),
$o_2 \equiv$ *gate*(false, b, i)
end,

agent *gate* ≡ **bool** bv, **chan bool** b, **chan v** $i \rightarrow$ **chan v** o:
$o \equiv$ **if isempty.**$b \vee$ **isempty.**i **then** ε
else if ft.$b = bv$ **then ft.**i & *gate*(bv, **rt.**b, **rt.**i)
else *gate*(bv, **rt.**b, **rt.**i) **fi fi**
end,

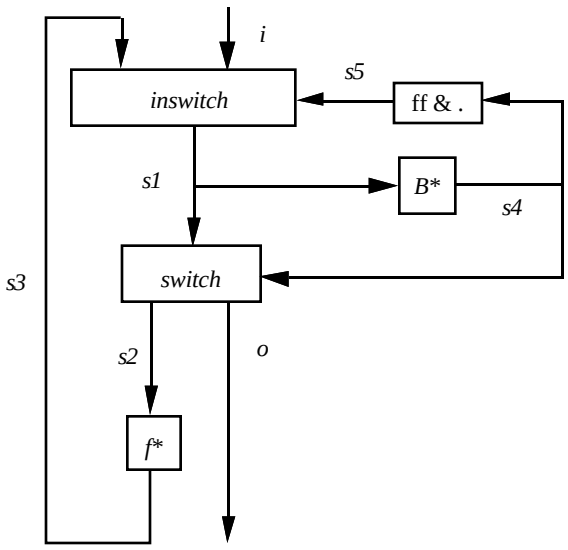
agent f^* ≡ **chan v** $i \rightarrow$ **chan v** o:
$o \equiv$ **if isempty.**i **then** ε **else** f(**ft.**i) & f^*(**rt.**i) **fi**
end,

agent B^* ≡ **chan v** $i \rightarrow$ **chan bool** o:
$o \equiv$ **if isempty.**i **then** ε **else** B(**ft.**i) & B^*(**rt.**i) **fi**
end,

$s_1 \equiv$ *inswitch*(s_5, s_3, i),
s_2, $o \equiv$ *switch*(s_4, s_1),
$s_3 \equiv f^*(s_2)$,
$s_4 \equiv B^*(s_1)$,
$s_5 \equiv$ false & s_4

end

Es definiert ein rückgekoppeltes Agentennetz:



Figur 3.4: Netzdarstellung des Programms *while**

und arbeitet wie folgt: Die Basisfunktion *f* wird solange auf *ein* Element des eingehenden Stroms *i* angewendet, wie die Bedingung *B* erfüllt ist. Dann gibt *switch* das Ergebnis der Rechnung auf *o* aus, während *inswitch* dual dazu den Zugang eines neuen Arguments erlaubt. □

4. Die prozedurale Sprache PL

Die in diesem Kapitel behandelte Sprache PL ist zuweisungsorientiert. Sie stützt sich auf getypte Programmvariable, deren Werte durch Zuweisungen verändert werden können. Wie in AL lassen sich Agenten definieren und parallel aufrufen. Die Kommunikation erfolgt dabei asynchron, mit Hilfe geeigneter Sende- und Empfangsprimitive, über unbeschränkte Kanäle.
PL ist die Zielsprache der angestrebten transformationellen Entwicklungen und daher sowohl von der syntaktischen Oberfläche, als auch von der konzeptuellen Ausgestaltung an das applikative AL angelehnt. Nichtsdestoweniger repräsentiert sie ein eigenständiges Ausdrucksmittel, das im Rahmen der in Kapitel 1 skizzierten Entwicklungsmethode FOCUS für implementierungsnahe Beschreibungen eingesetzt werden kann.

4.1 Syntax

Die Syntaxbeschreibung von PL folgt den gleichen Konventionen wie die Beschreibung von AL in Kapitel 3. Auch hier wird die Existenz paarweise disjunkter Identifikatormengen vorausgesetzt. Es sind dies die gleichen Mengen, die auch für AL benutzt wurden. Sie werden jedoch teilweise anders interpretiert:

FID: Identifikatoren für *Funktionen* und *Agenten*.

OID: Identifikatoren für *Variable*.

SID: Identifikatoren für *Kanäle*.

Um die Semantikdefinition für Anweisungen in Abschnitt 4.2.2 zu vereinfachen, nehmen wir an, daß die Menge der Kanalbezeichner SID in zwei disjunkte Teilmengen zerfällt:

IN: Identifikatoren für *Eingabekanäle*

OUT: Identifikatoren für *Ausgabekanäle*

In der anschließenden Grammatik seien: ‹prg_id›, ‹agt_id›, ‹fct_id› ∈ FID, ‹obj_id›, ‹var_id› ∈ OID, ‹chan_id› ∈ SID.

‹program› ::= **progam** ‹prg_id› ≡ ‹channel›* → ‹channel›$^+$:
{‹agent› | ‹function›}*
{‹eq_sys› | ‹stat›}

end

‹agent› ::= **agent** ‹agt_id› ≡ ‹object›*, ‹channel›* → ‹channel›+:
{‹eq_sys› | ‹stat›}
end

‹function› ::= **funct** ‹fct_id› ≡ ‹object›* → ‹sort›:
‹exp›
end

‹channel› ::= **chan** ‹sort› ‹chan_id›+

‹object› ::= ‹sort› ‹obj_id›+

‹eq_sys› ::= ‹equation›+

‹equation› ::= ‹chan_id›+ ≡ ‹agt_id›(‹exp›*, ‹chan_id›*)

‹stat› ::= **var** ‹sort› ‹var_id›+ {:= ‹exp›+} |
‹var_id› := ‹exp› | **skip** | **abort** |
‹chan_id› ! ‹exp› |
close.‹chan_id› | ‹stat› ; ‹stat› |
‹stat› ▯ ‹stat› |
if ‹exp› **then** ‹stat› **else** ‹stat› **fi** |
while ‹exp› **do** ‹stat› **od** |
loop ‹stat› **pool**

‹exp› ::= ⊥ | ‹primitive object› | ‹obj_id› |
isclosed.‹chan_id› | ‹exp› ▯ ‹exp› |
if ‹exp› **then** ‹exp› **else** ‹exp› **fi** |
{‹fct_id› | ‹primitive function›}(‹exp›*)

Die Struktur eines PL-Programms gleicht weitgehend der eines AL-Programms: In einer Kopfleiste wird zuerst der *Programmname* zusammen mit *Ein-* und *Ausgabekanälen* angeführt. Dann folgt ein *Definitionsteil*, in dem Funktionen und/oder Agenten erklärt werden, und schließlich der Programmrumpf, der entweder aus einem *Gleichungssystem* oder einer *Anweisung* besteht. Genau wie AL ist PL *typisiert*. Alle dort verfügbaren Sorten, primitiven Objekte und primitiven Funktionen sind auch in PL zulässig. Weiterhin sind die definierbaren Funktionen, d.h. die Elemente der syntaktischen Kategorien ‹function›, in AL und PL gleich. Der wesentliche Unterschied zwischen beiden Sprachen zeigt sich bei Betrachtung der Agentenkonzepte:

Der Rumpf eines PL-Agenten besteht, wie der eines vollständigen Programms, entweder aus aus einem *Gleichungssystem* oder aus einer (sequentiellen) *Anweisung*, der gegebenenfalls Variablendeklarationen vorangehen.
Anweisungen werden mit Hilfe der bekannten Konstruktoren – sequentielle Komposition (.;.), deterministische Auswahl (**if**.**then**.**else**.**fi**), nichtdeterministische Auswahl (.▯.) und Wiederholung (**while**.**do**.**od**) – aus elementaren Anweisungen zusammengesetzt. Dabei ist **skip** die leere Anweisung und **abort** die divergierende Anweisung. Zuweisungen *x* := E haben die übliche Semantik, jedoch ist zu beachten, daß Ausdrücke mehrdeutig seien können, so daß der Wert von *x* nach Ausführung der Zuweisung nicht immer eindeutig festliegt. **loop**.**pool** ist eine nichtterminierende Schleife und steht abkürzend für **while** true **do** . **od**.

Agenten kommunizieren mit ihrer Umgebung über *getypte, unidirektionale Kanäle*, auf die sie lesend und schreibend zugreifen. Seit Hoare's Arbeit über CSP aus dem Jahre 1978 (vgl. [Hoare 78]) sind Frage- und Ausrufezeichen als Notation hierfür gebräuchlich. Im Gegensatz zu der in CSP üblichen synchronen Interpretation werden die Zugriffsoperationen hier jedoch *asynchron* gedeutet:

Sei $i \in$ IN ein Eingabekanal und *x* eine Variable passenden Typs. Dann wird durch die Anweisung *i* ? *x* das erste auf *i* befindliche Element entfernt und *x* zugewiesen. Ist *i* zum Ausführungszeitpunkt "leer", so blockiert die Anweisung bis neue Daten verfügbar werden. Treffen die erwarteten Nachrichten niemals ein, so macht der Agent keine weiteren Fortschritte, er wartet unendlich lange (und vergeblich).

Sei $o \in$ OUT ein Ausgabekanal und E ein Ausdruck passenden Typs. Durch die Anweisung *o* ! E wird die Auswertung von E angestoßen und das Auswertungsergebnis e auf *o* ausgegeben. Enthält *o* in diesem Moment noch ungelesene Nachrichten, so wird e in FIFO-Manier hinter sie eingereiht. Wenn die Auswertung von E terminiert ($e \neq \perp$), kann die Schreiboperation in jedem Fall ausgeführt werden; der Kanal kann die Annahme einer Nachricht nicht verweigern. Damit bilden die Kommunikationskanäle in diesem Modell *unbeschränkte Puffer*, die es dem Sender erlauben, dem Empfänger beliebig weit zu enteilen. Sollen beide synchronisiert werden, so muß man dies durch Rückkopplung (*Flußkontrolle*) sicherstellen.
Durch das Kommando **close.***o* wird der Ausgabekanal *o* geschlossen und für weitere Übertragungen gesperrt. Alle weiteren Kommandos die sich auf *o* beziehen werden dann ignoriert, d.h. sie haben keine Wirkung auf *o*.

Ein Agent kann seine Eingangskanäle durch die eingebaute Operation **isclosed** überprüfen. **isclosed.***i* liefert "true", falls *i* geschlossen wurde und keine weiteren Nachrichten enhält und "false", falls noch ungelesene Nachrichten vorhanden sind. Ist *i* weder geschlossen, noch sind ungelesene Nachrichten verfügbar, so wird die Auswertung von **isclosed.***i* solange verzögert, bis der eine oder andere Fall eintritt. Dies kann wie bei der Leseanweisung zur ständigen Blockade führen. Der Versuch, von einem geschlossenen Kanal zu lesen, führt zur Divergenz.

Beispiel 4.1 (*add prozedural):** Die prozedurale Version des interaktiven Additions–agenten *add** aus Beispiel 3.2 hat folgende Gestalt:

agent *add** ≡ **chan nat** *i, j* → **chan nat** *o*:
    var nat *x, y*;
    while ¬**isclosed.***i* ∧ ¬**isclosed.***j* **do**
        i?*x*; *j*?*y*; *o*!*x*+*y*
    od;
    close.*o*
end

Der Schleifenanweisung ist die Deklaration der benutzten Variablen vorangestellt.
□

Alternativ zu Anweisungen lassen sich Agenten und Programme mit Hilfe von Gleichungssystemen definieren. Ein Gleichungssystem besteht aus einer durch Kommata getrennten Liste von Gleichungen der Form:

$$s_1, \ldots, s_n \equiv f(E, t_1, \ldots, t_m).$$

Jede Gleichung repräsentiert einen Agentenaufruf; sie erzeugt eine aktuelle Inkarnation, im obigen Fall von *f*. Die t_i sind dabei die aktuellen Eingabe- und die s_i die aktuellen Ausgabekanäle. E ist ein aktueller Objektparameter. Man beachte, daß in AL beliebige (Strom-)Ausdrücke auf den rechten Gleichungsseiten vorkommen können, in PL jedoch nur Agentenaufrufe.
Ein Gleichungssystem ist damit als *Parallelanweisung* anzusehen: n Gleichungen erzeugen n Agenten-Inkarnationen, die, operationell betrachtet, parallel arbeiten und durch die aktuellen Kanäle miteinander verbunden sind.

Beispiel 4.2 (*Erathostenes* prozedural): Auch das Erathostenes-Sieb ist in PL programmierbar:

agent *sieve* ≡ **chan nat** $i \rightarrow$ **chan nat** o:
s_1, s_2 ≡ *split*(i),
s_3 ≡ *filter*(s_2),
s_4 ≡ *sieve*(s_3),
o ≡ *join*(s_1, s_4)
end

sieve stützt sich auf folgende Agenten:

```
agent split ≡ chan nat i → chan nat o1, o2:
        var nat x;
        while ¬isclosed.i do i?x; o1!x; o2!x od;
        close.o1; close.o2
end,

agent join ≡ chan nat i1, i2 → chan nat o:
        var nat x;
        if isclosed.i1 then close.o
        else    i1?x; o!x;
                        while ¬isclosed.i2 do i2?x; o!x od;
                        close.o
        fi
end,

agent filter ≡ chan nat i → chan nat o:
        var nat x, y;
        if isclosed.i then close.o
        else    i?x;
                  while ¬isclosed.i   do
                                i?y; if y mod x = 0 then skip else o!y fi
                        od;
                   close.o
        fi
end
```

filter hat hier keinen Objektparameter. Der Agent liest die erste Nachricht von i und filtert dann alle Vielfachen von ihr aus den nachfolgenden Nachrichten heraus. *join* überträgt die erste auf i kommunizierte Nachricht und reicht dann alle Nachrichten von j weiter. *split* dupliziert die eingehenden Nachrichten und gibt sie auf zwei Kanälen aus. Beachte, daß *sieve* (funktions-) rekursiv ist. Die Korrektheit dieser Übersetzung läßt sich auf der Grundlage der Semantiken von AL und PL beweisen □

Die Semantik von PL-Agenten wird später wie in AL durch Mengen stromverarbeitender Funktionen beschrieben. Deshalb sind auch rekursive Definitionen wie die für *sieve* unproblematisch.

Wir nennen einen Agenten (bzw. ein vollständiges Programm) *sequentiell*, wenn sein Rumpf aus einer Anweisung besteht und *parallel* oder *hierarchisch* sonst. Parallele Agenten entsprechen in kanonischer Weise den Abschnitt 3.5 betrachteten Agentennetzen. Sequentielle Agenten bilden offensichtlich die Grundbausteine hierarchischer Strukturen. Sie können als *Prozeduren* angesehen werden, die zwar keinen globalen Zustand – alle Variablen sind in PL lokal – wohl aber den (aktuellen) Inhalt der sie verknüpfenden Kanäle verändern können. Durch gleichungsartiges Nebeneinanderstellen werden sie parallel komponiert und entsprechend parallel aufgerufen und ausgeführt.

PL-Programme müssen im wesentlichen denselben *Kontextbedingungen* genügen, deren Einhaltung auch von AL-Programmen gefordert wurde. Dies betrifft insbesondere Typisierungsaspekte. Für einen sequentiellen Agenten (A∈ ‹stat›)

agent $f \equiv$ **u** $x_1, \ldots, x_n$, **chan v** $i_1, \ldots, i_m \rightarrow$ **chan w** $o_1, \ldots, o_p$:
A
end

muß darüber hinaus gelten:

- Alle in A benutzten Variablen müssen in A deklariert sein. Globale Variablen sind verboten. Deklarationen stehen vor allen übrigen Anweisungen, Mehrfachdeklarationen sind verboten. Den Objektparametern x_j darf kein Wert zugewiesen werden.
- Auf die Eingabekanäle i_j darf nur lesend, auf die Ausgabekanäle o_j nur schreibend zugegriffen werden. Außer den i_j, o_j dürfen in A keine weiteren Kanäle verwendet werden.

Gleiches gilt für sequentielle Programme. Ein hierarchischer Agent (bzw. ein hierarchisches Programm) kann neben Ein- und Ausgabekanälen noch *interne Kanäle* als aktuelle Parameter bei Agentenaufrufen verwenden. Dabei gilt:

- Jeder Ausgabekanal wird *genau* einmal als aktueller Ausgabekanal und niemals als aktueller Eingabekanal verwendet. Das heißt, er tritt genau einmal auf der linken Seite einer Gleichung auf.
- Jeder Eingabekanal wird *höchstens* einmal als aktueller Eingabekanal und niemals als aktueller Ausgabekanal verwendet. Das heißt, er tritt höchstens einmal auf der rechten und niemals auf der linken Seite einer Gleichung auf.
- Jeder interne Kanal wird *genau* einmal als aktueller Ausgabekanal und *höchstens* einmal als aktueller Eingabekanal verwendet. Möglicherweise in ein- und demselben Aufruf.

Durch diese Einschränkungen (vgl. die schwächeren Anforderungen an die AL-Ströme auf S. 28) wird der Tatsache Rechnung getragen, daß Kanäle gerichtete Punkt-zu-Punkt Verbindungen realisieren: Sie verbinden genau einen Sender mit genau einem Empfänger. Eine implizite Aufspaltung durch Mehrfachverwendung wie in Stromgleichungssystem ist hier ausgeschlossen. Soll der gleiche Nachrichtenstrom mehreren Agenten zur Verfügung gestellt werden, müssen

entsprechende Verteilerknoten eingeführt werden (vgl. den Agenten *split* in Beispiel 4.2). Dadurch wird der konzeptuelle Unterschied zwischen Strömen und Kanälen deutlich. Das AL zugrunde liegende Stromkonzept ist in dieser Hinsicht abstrakter.
Für die Netzdarstellung (hierarchischer) PL-Agenten folgt aus den Kontextbedingungen, daß Kanten sowohl eindeutige Quellen als auch eindeutige Ziele haben. Sie werden also nicht mehr in Äste aufgespalten, die an mehreren Stellen enden.

4.2 Denotationelle Semantik

Die Semantik von PL stützt sich weitgehend auf die schon zuvor verwendeten denotationellen Konzepte. Dadurch reduziert sich der in diesem Abschnitt notwendige definitorische Aufwand, vor allem aber wird es möglich, in Kapitel 5 einen Korrektheitsbegriff für Transformationen von AL nach PL anzugeben.
Im folgenden definieren wir schrittweise die Semantik von Ausdrücken und Anweisungen, Funktionen, Agenten und vollständigen Programmen und bedienen uns dabei der bereits bekannten semantischen Abbildungen **B** und **F**. Dies ist ist möglich, weil wesentliche Sprachsegmente von PL mit denen von AL übereinstimmen. Einer besonderen Behandlung bedürfen die PL-typischen Anweisungen.
Während Funktionen, Agenten und vollständige Programme weiterhin funktional, d.h. durch Elemente aus $\wp(\mathbf{FCT}_S)\backslash\emptyset$ bzw. $\wp(\mathbf{AGT}_S)\backslash\emptyset$ beschrieben werden, beruht die Semantik von Anweisungen auf *Zustandstransformationen*. Zu diesem Zweck definieren wir eine weitere semantische Funktion **S**. Gemäß der synthetischen Konzeption denotationeller Beschreibungen (vgl. die Einleitung zu Abschnitt 3.2), muß die Semantik eines Agenten aus der Semantik seines Rumpfes abgeleitet werden. Da Agentenrümpfe aus Anweisungen bestehen können, ist es notwendig, stromverarbeitende Funktionen und Zustandstransformationen zueinander in Beziehung zu setzen.
Die folgenden Definitionen stützen sich auf die in Abschnitt 3.2 eingeführten Funktionsbereiche **MAP**, **FCT** und **AGT**, sowie auf Umgebungen:

$$\mathrm{ENV} = \mathrm{FID} \rightarrow \wp(\mathbf{FCT}_S)\backslash\emptyset \cup \wp(\mathbf{AGT}_S)\backslash\emptyset$$

und Zustände:

$$\mathrm{STATE} = \mathrm{ID} \rightarrow \mathbf{Dom}^{\omega} \cup \mathbf{Dom}^{\perp}$$

dabei gilt vereinbarungsgemäß: ID = SID ∪ OID und SID = IN ∪ OUT.

4.2.1 Semantik von Ausdrücken

Vergleicht man die syntaktischen Produktionen für AL-Ausdrücke (S. 22) mit denen für PL-Ausdrücke (S. 62), so stellt man zwei Dinge fest:

- Jeder wohlgeformte PL-Ausdruck ist auch ein wohlgeformter AL-Ausdruck, wenn man den Operator **isclosed** überall durch **isempty** ersetzt.
- Umgekehrt: Jeder reine Objektausdruck aus AL ist auch ein korrekter PL-Ausdruck[4].

Im Prinzip kann die für AL axiomatisch definierte Breitensemantik (vgl. Abschnitt 3.2.1)

$$\mathbf{B}: \langle \text{exp} \rangle \to \text{ENV} \to \text{STATE} \to \wp(\mathbf{Dom}^{\perp})\backslash\emptyset,$$

daher unverändert für PL übernommen werden. PL-Ausdrücke liefern jedoch stets Objekte und niemals Ströme, daher die etwas veränderte Funktionalität von **B** an dieser Stelle. Rein syntaktisch verwenden wir in PL das Schlüsselwort **isclosed** und in AL **isempty**. Dies ist ein Zugeständnis an die unterschiedliche Intuition der beiden Sprachkonzepte. Semantisch besteht kein Unterschied. Wir definieren:

$$\mathbf{B}_{\delta,\sigma}[\![\mathbf{isclosed.}o]\!] = \{ \text{isempty}(\sigma(o)) \}$$

Basierend auf **B** und analog zu Abschnitt 3.2.2 ist die funktionale Ausdruckssemantik erklärt:

$$\mathbf{F}: \langle \text{exp} \rangle \to \text{ENV} \to \wp(\mathbf{MAP})\backslash\emptyset$$

mit

$$\mathbf{F}_\delta[\![\text{E}[x]]\!] = \{ \lambda x.\mathbf{B}_{\delta',\sigma[x/x]}[\![\text{E}']\!] \mid \delta' \in \text{DD}(\delta\tilde{}), \text{E}' \in \text{DD}(\text{E}[x]\tilde{}), \sigma \in \text{STATE} \}.$$

Dabei sind $\text{E}[x]\tilde{}$ bzw. $\delta\tilde{}$ die *S-Normalformen* von $\text{E}[x]$ bzw. δ und $\text{DD}(\text{E}[x]\tilde{})$ bzw. $\text{DD}(\delta\tilde{})$ die Mengen ihrer *deterministischen Abkömmlinge*.

Für einen Ausdruck E, in dem (ausschließlich) die Bezeichner $x_1, \ldots, x_n$ vorkommen, ist jedes $f \in \mathbf{F}_\delta[\![\text{E}[x_1, \ldots, x_n]]\!]$ eine n-stellige Funktion. Um die Notation der Anweisungssemantik im nächsten Abschnitt zu erleichtern, soll folgende *Schreibkonvention* eingehalten werden:

Sei σ ein Zustand und $f \in \mathbf{F}_\delta[\![\text{E}[x_1, \ldots, x_n]]\!]$. Dann schreiben wir

$$f(\sigma) \qquad \text{anstelle von} \qquad f(\sigma(x_1), \ldots, \sigma(x_n)).$$

Beachte: Faßt man f in diesem Sinn als Funktion von Zuständen auf Werte auf, dann ist f stetig bzgl. der punktweisen Ordnung auf STATE.

---

[4] Zur Erinnerung: Ein AL-Ausdruck E hieß *reiner Objektausdruck*, wenn er von einer Objektsorte war und keine der Funktionen **ft**, **rt** und **isempty** in ihm vorkam (vgl. Abschnitt 3.1). Reine Objektausdrücke bildeten den Rumpf von Funktionsdefinitionen.

4.2.2 Semantik von Anweisungen

Semantisch entsprechen Anweisungen *Mengen von Zustandstransformationen*. Die Mengenstruktur ist dabei wie stets auf den möglichen Nichtdeterminismus zurückzuführen, hier sogar in zweifacher Hinsicht: Einerseits kann eine Anweisung $A \in \langle stat \rangle$ selber die Form $A_1 \,[]\, A_2$ haben, z.B. $x := 1 \,[]\, x := 2$, anderseits kann sie sich auf einen mehrdeutigen Ausdruck abstützen, z.B.: $x := 1[]2$. (Es wird sich zeigen, daß beide Versionen, die gleiche Bedeutung haben.)

Ein Zustand $\sigma \in$ STATE ordnet jeder Variablen x ein Element aus $\mathbf{Dom}^{\perp}$ und jedem Kanal c einen Strom aus $\mathbf{Dom}^{\omega}$ zu: $\sigma(x)$ repräsentiert den *aktuellen* Wert von x und $\sigma(c)$ den *aktuellen* Inhalt von c. Anweisungen verändern Zustände, indem sie Variablen Werte zuweisen und von Eingabekanälen lesen bzw. auf Ausgabekanäle schreiben. Ausgabekanäle werden in den anschließenden Definitionen anders behandelt als Eingabekanäle. Dies hat folgende Gründe (vgl. hierzu und zu den kommenden Aussagen [Broy, Lengauer 91]):

a) Semantisch werden auch divergierende Anweisungen wie **abort**, $x := \perp$ oder $o!\perp$ durch Zustandstransformationen modelliert. Was aber soll das Ergebnis einer solchen Transformation sein? Üblicherweise wählt man den ausgezeichneten Zustand $\lambda x.\perp$, der allen Zustandskomponenten den Wert "undefiniert" zuordnet (vgl. etwa [Gordon 79]). Für PL ist diese Vorgehensweise ungeeignet: Durch den Übergang zu $\lambda x.\perp$ wird alle bis zum Zeitpunkt der Divergenz erzeugte Information zerstört, einschließlich der bereits auf die Ausgabekanäle gesandten Nachrichten. Bereits ausgegebene Nachrichten können aber von einem Agenten nicht mehr beeinflußt werden. Sie zu "löschen", wäre inadequat.

b) Auf den "Inhalt" von Eingabekanälen wird am "vorderen" Ende zugegriffen, der "Inhalt" von Ausgabekanälen wird dagegen am "hinteren" Ende erweitert. Während Zugriffe am vorderen Ende unproblematisch sind, wirft die Erweiterung am hinteren Ende Monotonieprobleme auf. Betrachte dazu folgendes Beispiel: Es gilt $\langle\perp\rangle \sqsubseteq \langle 1\rangle\hat{}\langle\perp\rangle$. Erweitert man beide Ströme am hinteren Ende z.B. um 2, dann ergibt sich: $\langle 2\rangle\hat{}\langle\perp\rangle \not\sqsubseteq \langle 1\rangle\hat{}\langle 2\rangle\hat{}\langle\perp\rangle$.

Wir definieren daher einen alternativen Strombereich $\mathbf{Dom}^{\Omega}$ und eine alternative Ordnung $\sqsubseteq_{\Omega}$:

$$\mathbf{Dom}^{\Omega} = \mathbf{Dom}^{*} \cup \mathbf{Dom}^{*} \times \{\perp, @\} \cup \mathbf{Dom}^{\infty}.$$

$\perp$ steht weiterhin als Symbol für Divergenz. @ repräsentiert eine *Endemarke*, die das Ende eines Stroms explizit anzeigt. Für $s,t \in \mathbf{Dom}^{\Omega}$ gelte:

$$s \sqsubseteq_{\Omega} t \;\Leftrightarrow\; \exists s_1 \in \mathbf{Dom}^{\Omega}: t = s\hat{}s_1.$$

Im Gegensatz zum ursprünglichen Strombereich $\mathbf{Dom}^{\omega}$ ist hier nicht mehr $\langle\perp\rangle$ sondern ε das kleinste Element. Die Elemente aus $\mathbf{Dom}^{*}$ sind partiell (bzgl. $\sqsubseteq_{\Omega}$) und die aus $\mathbf{Dom}^{*} \times \{\perp, @\} \cup \mathbf{Dom}^{\infty}$ total. Wir passen nun den Zustandsbegriff an und ordnen Ausgabekanälen Ströme aus $\mathbf{Dom}^{\Omega}$ zu.

$$\text{STATE'} = \text{ID} \to \mathbf{Dom}^{\perp} \cup \mathbf{Dom}^{\omega} \cup \mathbf{Dom}^{\Omega}$$

Die Ordnung $\sqsubseteq$ auf STATE' ist wie üblich elementweise bestimmt: Für Variablen durch die flache Ordnung auf $\mathbf{Dom}^{\perp}$, für Eingabekanäle durch die bekannte Ordnung $\sqsubseteq$ auf $\mathbf{Dom}^{\omega}$ und für Ausgabekanäle durch $\sqsubseteq_{\Omega}$. Für ein $\sigma \in$ STATE' und ein $o \in$ OUT lassen sich folgende Situationen unterscheiden:

- $\sigma(o) = s \in \mathbf{Dom}^{*}$. Der sendende Agent hat bisher den Nachrichtenstrom s auf *o* gesandt. Er arbeitet noch und kann noch weitere Nachrichten auf diesen Kanal ausgeben.

- $\sigma(o) = s^{\frown}\langle @ \rangle$. Der Agent hat den Strom s versandt und dann den Kanal mit **close.***o* geschlossen. Weitere Übertragungen sind nicht mehr möglich.

- $\sigma(o) = s^{\frown}\langle \perp \rangle$. Der Agent hat den Strom s versandt und dann divergiert, ohne weitere Nachrichten zu generieren.

Man beachte, daß $\sigma(o)$ bei der Definition der Anweisungssemantik nicht die vollständige *Kommunikationsgeschichte* von *o*, sondern nur einen *temporären Zustand* repräsentiert. Durch die Unterscheidung zwischen der ersten und dritten Situation wird *intensionale Information* über den Agenten ausgedrückt. Dies ist gerechtfertigt, da an dieser Stelle sein internes Verhalten beschrieben wird. Äußere Kommunikationspartner können beide Situationen nicht unterscheiden. Im ursprünglichen Strombereich $\mathbf{Dom}^{\omega}$ werden sie daher identifiziert. Die Funktion

$$\text{cast}: \mathbf{Dom}^{\Omega} \to \mathbf{Dom}^{\omega}$$

löscht die intensionale Zusatzinformation:

$$\text{cast}(s) = \begin{cases} s & \text{falls } s \in \mathbf{Dom}^{*} \times \{\perp\} \cup \mathbf{Dom}^{\infty} \\ s^{\frown}\langle \perp \rangle & \text{falls } s \in \mathbf{Dom}^{*} \\ s_1 & \text{falls } s = s_1{}^{\frown}\langle @ \rangle \end{cases} .$$

cast ist monoton: $s \sqsubseteq_{\Omega} t \Rightarrow \text{cast}(s) \sqsubseteq \text{cast}(t)$ und für $s \in \mathbf{Dom}^{*}$ gilt: $\text{cast}(s) = \text{cast}(s^{\frown}\langle \perp \rangle)$. (Vergleiche die Situationen eins und drei oben).

Dual zu & sei $\bullet$ der Operator, mit dem sich ein Element $d \in \mathbf{Dom} \cup \{\perp, @\}$ ans Ende eines Strom $s \in \mathbf{Dom}^{\Omega}$ anfügen läßt:

$$s \bullet d = \begin{cases} s & \text{falls } s \in \mathbf{Dom}^{\infty} \cup \mathbf{Dom}^{*} \times \{\perp, @\} \\ s^{\frown}\langle d \rangle & \text{falls } s \in \mathbf{Dom}^{*} \end{cases} .$$

$\mathbf{Dom}^{\Omega}$ ist unter $\bullet$ abgeschlossen und es gilt:

$$s \sqsubseteq_{\Omega} s \bullet d.$$

Ein Zustand $\sigma \in$ STATE' heißt *(extensional) stabil*, wenn gilt:

$$\forall o \in \mathrm{OUT}: \sigma(o) \in \mathbf{Dom}^{\infty} \cup \mathbf{Dom}^{*} \times \{\bot, @\}$$

Die Veränderung eines stabilen Zustandes ist durch die Kommunikationspartner eines Agenten, d.h. von außen nicht beobachtbar, da sich der Zustand der Ausgabekanäle nicht mehr ändern kann. Wir schreiben ST.σ wenn σ ein stabiler Zustand ist.

Das auf Seite 69 unter a) beschriebene Problem der Behandlung divergierender Anweisungen, wird nun durch Übergang von einem Zustand σ in den Zustand $\sigma\downarrow$ gelöst. $\sigma\downarrow$ ist wie folgt definiert:

$$\mathrm{ST}.\sigma \quad \Rightarrow \quad \sigma\downarrow = \sigma,$$

$$\neg\mathrm{ST}.\sigma \quad \Rightarrow \quad \sigma\downarrow(x) = \begin{cases} \bot & \text{falls } x \in \mathrm{OID} \cup \mathrm{IN} \\ \sigma(x) \bullet \bot & \text{falls } x \in \mathrm{OUT} \end{cases}.$$

Wenn σ noch nicht stabil ist, werden durch den Übergang zu $\sigma\downarrow$ alle Variablen und Eingabekanäle gelöscht, der Inhalt der Ausgabekanäle bleibt aber erhalten. $\downarrow$ ist idempotent: $\sigma\downarrow\downarrow = \sigma\downarrow$ und monoton: $\sigma_1 \sqsubseteq \sigma_2 \Rightarrow \sigma_1\downarrow \sqsubseteq \sigma_2\downarrow$, und es gilt: $\sigma\downarrow \sqsubseteq \sigma$. Darüber hinaus ist $\sigma\downarrow$ in jedem Fall stabil: ST.$\sigma\downarrow$.

Die Aktualisierungsoperation für Zustände (vgl. Abschnitt 3.2) wird ebenfalls leicht variiert (sei e ein semantischer Wert, x, y seien Bezeichner):

$$\mathrm{ST}.\sigma \quad \Rightarrow \quad \sigma[e/x] = \sigma,$$

$$\neg\mathrm{ST}.\sigma \quad \Rightarrow \quad \sigma[e/x](y) = \begin{cases} e & \text{falls } P(e,x) \wedge x = y \\ \sigma(y) & \text{falls } P(e,x) \wedge x \neq y \\ \sigma\downarrow(y) & \text{falls } \neg P(e,x) \end{cases},$$

dabei steht P(e,x) abkürzend für:

$$P(e,x) = (x \in \mathrm{OID} \Rightarrow e \neq \bot) \wedge (x \in \mathrm{OUT} \Rightarrow e \notin \mathbf{Dom}^{*} \times \{\bot\}).$$

Nach dieser Definition können stabile Zustände nicht mehr aktualisiert werden. Der Versuch, einer Variablen den Wert $\bot$ bzw. einem Ausgabekanal einen Strom aus $\mathbf{Dom}^{*} \times \{\bot\}$ zuzuweisen, führt zum Übergang in einen divergenten Zustand.

Zustandstransformationen sind stetige Abbildungen zwischen Zuständen. Die Menge aller Zustandstransformationen ist wie folgt definiert:

$$\begin{aligned}\textbf{STATE-TRANS} = \{\ f \in [\text{STATE}' \to \text{STATE}'] \mid \forall \sigma \in \text{STATE}':\ & (\text{ST}.\sigma \Rightarrow f(\sigma) = \sigma) \wedge \\ & (\neg\text{ST}.\sigma \Rightarrow \forall i \in \text{IN}: \exists n \in \textbf{Nat} \cup \{\infty\}: \text{rt}^n(\sigma(i)) = f(\sigma)(i) \wedge \\ & \forall o \in \text{OUT}: \sigma(o) \sqsubseteq_\Omega f(\sigma)(o)\ )\ \}\end{aligned}$$

Dabei bezeichnet rt^n wie üblich die n-fache Anwendung von rt (vgl. Satz 2.1). Definiere $\text{rt}^\infty(\sigma(i)) = \bot$. Beachte: $\sigma(i), f(\sigma)(i) \in \textbf{Dom}^\omega$ und $\sigma(o), f(\sigma)(o) \in \textbf{Dom}^\Omega$.

Intuitiv ausgedrückt darf eine Zustandstransformation f (bzw. die zugehörige Anweisung A_f) nichts an Eingabekanäle i anfügen, sondern nur von dort entfernen und umgekehrt nichts von Ausgabekanälen o entfernen, sondern nur daran anfügen. Falls der Funktionswert $f(\sigma)(i)$ in der Gleichung $\text{rt}^n(\sigma(i)) = f(\sigma)(i)$ ungleich $\bot$ ist, bildet er offensichtlich ein *Postfix* von $\sigma(i)$. $f(\sigma)(i)$ repräsentiert die Nachrichten, die A_f nicht verbraucht hat und die deshalb auf i verbleiben. Das $\sigma(o)$ in der Formel $\sigma(o) \sqsubseteq_\Omega f(\sigma)(o)$ ist stets ein Präfix von $f(\sigma)(o)$. Es repräsentiert die Nachrichten, die vor Ausführung von A_f auf o vorhanden waren, und $f(\sigma)(o)$ repräsentiert die Nachrichten, die nach Ausführung von A_f auf o vorhanden sind.

Die Ordnung auf STATE-TRANS ist wie üblich punktweise definiert, d.h. für $f_1, f_2 \in$ STATE-TRANS gilt:

$$f_1 \sqsubseteq f_2 \Leftrightarrow \forall \sigma \in \text{STATE}': f_1(\sigma) \sqsubseteq f_2(\sigma)$$

STATE-TRANS bildet damit einen Bereich. Das kleinste Element dieses Bereichs ist die Funktion, die stabile Zustände unverändert läßt und die alle Variablen und Eingabekanäle nicht stabiler Zustände auf $\bot$ setzt und alle Ausgabekanäle unverändert läßt.

Nach diesen Vorbereitungen sind wir nun in der Lage, die semantische Funktion **S** festzulegen:

$$\textbf{S}: \langle\text{stat}\rangle \to \text{ENV} \to \wp([\textbf{STATE-TRANS}])\backslash\varnothing.$$

S ist axiomatisch definiert:

$$\textbf{S}_\delta[\![\textbf{skip}]\!] = \{\ \lambda\sigma.\sigma\ \},$$

$$\textbf{S}_\delta[\![\textbf{abort}]\!] = \{\ \lambda\sigma.\sigma\downarrow\ \},$$

$$\textbf{S}_\delta[\![x := \text{E}]\!] = \{\ \lambda\sigma.\sigma[h(\sigma)/x] \mid h \in \textbf{F}_\delta[\![\text{E}]\!]\ \},$$

$\mathbf{S}_\delta[\![i\ ?\ x]\!] = \{\ \lambda\sigma.\sigma[\mathrm{ft}(\sigma(i))/x,\ \mathrm{rt}(\sigma(i))/i]\ \},$

$\mathbf{S}_\delta[\![o\ !\ \mathrm{E}]\!] = \{\ \lambda\sigma.\mathrm{if}\ \mathrm{h}(\sigma) = \bot\ \mathrm{then}\ \sigma\downarrow\ \mathrm{else}\ \sigma[\sigma(o)\bullet\mathrm{h}(\sigma)/o]\ \mid\ \mathrm{h}\in\mathbf{F}_\delta[\![\mathrm{E}]\!]\ \},$

$\mathbf{S}_\delta[\![\mathbf{close}.o]\!] = \{\ \lambda\sigma.\sigma[\sigma(o)\bullet@/o]\ \},$

$\mathbf{S}_\delta[\![\mathrm{A}_1 [\!] \mathrm{A}_2]\!] = \mathbf{S}_\delta[\![\mathrm{A}_1]\!] \cup \mathbf{S}_\delta[\![\mathrm{A}_2]\!],$

$\mathbf{S}_\delta[\![\mathrm{A}_1\ ;\ \mathrm{A}_2]\!] = \{\ \mathrm{f}_2 \circ \mathrm{f}_1\ \mid\ \mathrm{f}_i\in\mathbf{S}_\delta[\![\mathrm{A}_i]\!]\ \},$

$\mathbf{S}_\delta[\![\mathbf{if}\ \mathrm{B}\ \mathbf{then}\ \mathrm{A}_1\ \mathbf{else}\ \mathrm{A}_2\ \mathbf{fi}]\!] = \{\ \mathrm{if}(\mathrm{h}, \mathrm{f}_1, \mathrm{f}_2)\ \mid\ \mathrm{h}\in\mathbf{F}_\delta[\![\mathrm{B}]\!], \mathrm{f}_i\in\mathbf{S}_\delta[\![\mathrm{A}_i]\!]\ \}$

$$\text{wobei}\quad \mathrm{if}(\mathrm{h}, \mathrm{f}_1, \mathrm{f}_2)(\sigma) = \begin{cases} \sigma\downarrow & \text{falls } \mathrm{h}(\sigma) = \bot \\ \mathrm{f}_1(\sigma) & \text{falls } \mathrm{h}(\sigma) = \mathrm{true} \\ \mathrm{f}_2(\sigma) & \text{falls } \mathrm{h}(\sigma) = \mathrm{false} \end{cases}$$

$\mathbf{S}_\delta[\![\mathbf{while}\ \mathrm{B}\ \mathbf{do}\ \mathrm{A}\ \mathbf{od}]\!] = \mathrm{FIX}(\tau)$,

wobei $\mathrm{FIX}(\tau)$ den größten Fixpunkt des mengenwertigen Funktionals

$$\tau:\ \wp(\mathbf{STATE\text{-}TRANS})\backslash\varnothing \to \wp(\mathbf{STATE\text{-}TRANS})\backslash\varnothing$$

mit ($\mathrm{id} = \lambda\sigma.\sigma$):

$$\tau.\mathrm{M} = \{\ \mathrm{if}(\mathrm{h}, \mathrm{f}_\mathrm{M} \circ \mathrm{f}_\mathrm{A}, \mathrm{id})\ \mid\ \mathrm{h}\in\mathbf{F}_\delta[\![\mathrm{B}]\!], \mathrm{f}_\mathrm{A}\in\mathbf{S}_\delta[\![\mathrm{A}]\!], \mathrm{f}_\mathrm{M}\in\mathrm{M}\ \}$$

bezeichnet,

$\mathbf{S}_\delta[\![\mathbf{loop}\ \mathrm{A}\ \mathbf{pool}]\!] = \mathbf{S}_\delta[\![\mathbf{while}\ \mathrm{true}\ \mathbf{do}\ \mathrm{A}\ \mathbf{od}]\!]$.

Deklarationen verändern den Zustand nur, wenn sie mit einer initialisierenden Zuweisung verbunden sind:

$\mathbf{S}_\delta[\![\mathbf{var\ u}\ x_1, \ldots, x_n]\!] = \{\ \lambda\sigma.\sigma\ \},$

$\mathbf{S}_\delta[\![\mathbf{var\ u}\ x_1, \ldots, x_n := \mathrm{E}_1, \ldots, \mathrm{E}_n]\!] = \{\ \lambda\sigma.\sigma[\mathrm{h}_i(\sigma)/x_i]\ |\ \mathrm{h}_i\in\mathbf{F}_\delta[\![\mathrm{E}_i]\!]\ \}.$

Bis auf das **while**-Axiom sind alle Axiome unproblematisch. Es läßt sich durch einfaches Nachrechnen zeigen, daß rechts nur (nicht-leere) Mengen von Zustandstransformationen zu finden sind. Für die Wohldefiniertheit von **S** ist damit das **while**-Axiom ausschlaggebend, also die Frage, ob τ tatsächlich einen größten Fixpunkt besitzt. Dies kann man mit Hilfe von Satz 2.3 positiv beantworten.

Satz 4.3 (Wohldefiniertheit des while-Axioms): Für τ gilt:

i) τ ist stetig bzlg. $\subseteq$,

ii) $\exists M \in \wp(\textbf{STATE-TRANS}) \setminus \emptyset : M \subseteq \tau.M$.

Beweis: i) folgt direkt aus der Definition von τ.
ii) Für den Nachweis von ii) nehmen wir o.B.d.A. an, daß A keine **while**-Schleife enthält.
Betrachte nun beliebige, aber feste $h \in \mathbf{F}_\delta[\![B]\!]$ und $f_A \in \mathbf{S}_\delta[\![A]\!]$.
ξ: **STATE-TRANS** $\rightarrow$ **STATE-TRANS** sei das dadurch festgelegte Funktional mit:

$$\xi.f = \text{if}(h, f \circ f_A, \text{id})$$

Es gilt definitionsgemäß für alle $f \in$ **STATE-TRANS** (*): $\xi.f \in \tau.\{f\}$.
Da h, f und f_A stetig sind und die Funktionskomposition $\circ$ und if stetige Funktionale sind, ist auch ξ stetig und besitzt daher einen kleinsten Fixpunkt fix(ξ). Für diesen gilt wegen (*):
$\text{fix}(\xi) = \xi.\text{fix}(\xi) \in \tau.\{\text{fix}(\xi)\}$. Also $\{\text{fix}(\xi)\} \subseteq \tau.\{\text{fix}(\xi)\}$
□

Für die denotationelle Semantik von AL haben wir im Abschnitt 3.2.5 festgestellt, daß sie das operationelle Verhalten eines Programms in einem bestimmten Sinn "robust" wiedergibt:
Nichtterminierende Rekursion wird dort nicht durch $\perp$ sondern durch "Chaos" beschrieben, d.h., vom Zeitpunkt der Divergenz an ist jede beliebige Ausgabe möglich. Die denotationelle Semantik von PL ist damit konsistent. Betrachte dazu die nichtterminierende Anweisung

while true **do skip od**

Es gilt: $\mathbf{S}_\delta[\![$**while** true **do skip od**$]\!]$ ist die größte Menge M mit der Eigenschaft

$$\begin{aligned} M &= \{ \text{if}(h, f_M \circ f_A, \text{id}) \mid h \in \mathbf{F}_\delta[\![\text{true}]\!], f_M \in M, f_A \in \mathbf{S}_\delta[\![\text{skip}]\!] \} \\ &= \{ \text{if}(h, f_M \circ f_A, \text{id}) \mid h = \lambda\sigma.\text{true}, f_M \in M, f_A = \lambda\sigma.\sigma \} \\ &= \{ f_M \mid f_M \in M \} \end{aligned}$$

Offensichtlich ist die Menge *aller* Zustandstransformationen die größte Lösung dieser Gleichung. Nichtterminierende Schleifen (mit wirkungslosem Rumpf!) werden also durch "Chaos" interpretiert, jeder beliebige Zustandsübergang ist möglich (siehe auch Abschnitt 4.3, S. 77)

4.2.3 Semantik von Funktionen, Agenten und Programmen

Die Semantik von Funktionen, Agenten und vollständigen Programmen wird wie für AL durch Mengen stetiger (stromverarbeitender) Funktionen erklärt. **F** hat dabei für die einzelnen syntaktischen Kategorien die schon bekannten Funktionalitäten:

$\mathbf{F}$: ‹function› $\to$ ENV $\to$ $\wp(\mathbf{FCT}_S)\backslash\emptyset$,

$\mathbf{F}$: ‹agent› $\to$ ENV $\to$ $\wp(\mathbf{AGT}_S)\backslash\emptyset$,

$\mathbf{F}$: ‹program› $\to$ $\wp(\mathbf{AGT}_S)\backslash\emptyset$.

Im ersten Fall können wir die semantischen Definition aus Abschnitt 3.2.4 unverändert übernehmen, da AL-Funktionen und PL-Funktionen syntaktisch vollständig übereinstimmen. Für die beiden anderen Fälle muß zwischen *hierarchischen* und *sequentiellen* Agenten bzw. Programmen unterschieden werden. Hierarchische Agenten (Programme) sind gleichungsdefiniert, ihr Rumpf entspricht einem AL-Gleichungssystem, auf dessen rechten Seiten nur Agentenaufrufe vorkommen. Gleichungssysteme werden in 3.2.3 und im Zusammenhang mit Agenten und Programmen in 3.2.5 bzw. 3.2.6 behandelt. Diese Definitionen übertragen sich kanonisch auf den Fall von PL-Gleichungen.

Um die Semantik von PL zu komplettieren, müssen daher nur noch sequentielle Agenten bzw. Programme behandelt werden. Wir führen dies hier für seq. Agenten aus. Sei

agent $f \equiv$ **u** x, **chan** **v** $i \to$ **chan** **w** o: A **end**

mit A$\in$ ‹stat› ein sequentieller PL-Agent. Dann ist die Semantik von f wie folgt definiert:

$\mathbf{F}_\delta[\![$ **agent** $f \equiv \ldots$ **end** $]\!] = \{ f \mid \exists f_A \in \mathbf{S}_\delta[\![A]\!]: \forall u \in \mathbf{U}^\perp, v \in \mathbf{V}^\omega:$

$$f(u,v) = \mathrm{cast}(f_A(\sigma_0[u/x, v/i])(o)) \},$$

dabei ist $\sigma_0 \in$ STATE' der (Anfangs-)Zustand für den gilt:

$$\forall x \in \mathrm{OID} \cup \mathrm{IN}: \sigma_0(x) = \perp \wedge \forall x \in \mathrm{OUT}: \sigma_0(x) = \varepsilon.$$

Insbesondere sind also alle Ausgabekanäle bei "Start" des Agenten leer. Der Eingangskanal i wird mit dem (vollständigen!) Eingabestrom v vorbesetzt, die Variable x mit dem aktuellen Objektparameter u. Beachte, daß f in seinem Objektparameter strikt ist! Dies folgt aus der Definition der Aktualisierungsoperation und der Tatsache, daß Zustandstransformationen stabile Zustände nicht mehr verändern:

$$f(\perp,v) = \mathrm{cast}(\, f_A(\sigma_0[\perp/x, v/i])(o)\,) = \mathrm{cast}(\, f_A(\sigma_0\downarrow)(o))\,) = \mathrm{cast}(\, \sigma_0\downarrow(o)\,) = \mathrm{cast}(\, \varepsilon\bullet\perp\,) = \perp$$

Jede durch A bestimmte Zustandstransformation f_A erzeugt bei Anwendung auf σ_0 eine Ausgabe auf dem Ausgabekanal o. Führt man mit Hilfe von "cast" eine Typanpassung durch (und "vergißt" dadurch die nicht-extensionale Information), so ergibt sich der Funktionswert der Stromfunktion, die das (bzw. ein) Verhalten des Agenten beschreibt. Weil f_A und "cast" stetig sind, ist auch f stetig.

4.3 AL und PL: Gemeinsamkeiten und Unterschiede

Zwischen AL und PL gibt es eine Reihe bemerkenswerter Gemeinsamkeiten, durchaus aber auch Unterschiede. Am stärksten fällt die *syntaktische Verwandtschaft* ins Auge:

AL- und PL-Programme sind nicht nur gleich strukturiert – an eine Folge von Funktions- und Agentendeklarationen schließt sich jeweils ein Hauptteil an –, sondern verwenden auch identische syntaktische Elemente. Zum Beispiel haben die Kopfzeilen von Programmen, Agenten und Funktionen in beiden Sprachen das gleiche Format. Rein prozedurale Elemente wie Variablen, Zuweisungen und Schleifen kommen natürlich nur in PL vor. Gleichungssysteme werden jedoch auf beiden Ebenen verwendet. Syntaktisch gilt dabei eine (echte) Inklusionsbeziehung: Jedes PL-Gleichungssystem ist auch ein wohlgeformtes AL-System, aber nicht umgekehrt. In dieser Einschränkung spiegeln sich auch die unterschiedlichen Sprachkonzeptionen:

- Die applikative Sprache AL ist *mathematisch* orientiert. Gleichungssysteme sind hier in einem sehr direkten Sinn zu *lösen*. Sie verwenden Unbekannte (Strombezeichner) und beschreiben implizit die Menge ihrer kleinsten Lösungen (Tupel von Strömen).
- Das prozedurale Sprache PL ist weniger abstrakt und stärker *operationell* ausgerichtet. Ein Gleichungssystem ist hier als *Parallelanweisung* zu verstehen: Intuitiv betrachtet wird es nicht gelöst, sondern (parallel) *ausgeführt*.

Im Rahmen der semantischen Behandlung verschwinden diese Unterschiede; darin liegt gerade einer der Vorzüge des gewählten funktionalen Ansatzes. Die denotationelle Beschreibung beider Sprachen macht den "Ausführungsgedanken" schon auf der applikativen Ebene, und den "Lösungsgedanken" auch auf der prozeduralen Ebene anwendbar. Tatsächlich bilden beide nur zwei Seiten derselben Medaille, die in PL vorzugsweise von dieser, in AL vorzugsweise von jener Seite betrachtet wird.

Semantische Gemeinsamkeiten ergeben sich vor allem aus der uniformen Verwendung der funktionalen Semantik **F**. Mit ihrer Hilfe wird die Bedeutung von Programmen und Agenten in beiden Sprachen durch Mengen stromverarbeitender Funktionen erklärt. Für AL ist das naheliegend, da die mathematische Sprachkonzeption den Funktionscharakter herausstreicht. Mit einem (sequentiellen) PL-Agenten verbindet sich dagegen eher die Vorstellung einer Prozedur, deren Aufruf eine Zustandsänderung bewirkt. Wie die Definitionen in den vorigen Abschnitten zeigen, können Zustandstransformationen und stromverarbeitende Funktionen so zu einander in Beziehung gesetzt werden, daß das Ein/Ausgabeverhalten prozeduraler Agenten präzise beschrieben wird. Wichtig ist in diesem Zusammenhang die gleichartige Behandlung von Nichtdeterminismus durch Unterspezifikation, insbesondere im Zusammenspiel mit Rekursion. Dies wird besonders deutlich, wenn man einen Grenzfall betrachtet: Der applikative Agent *ntr* aus Abschnitt 3.2.5

agent *ntr* ≡ **chan** **v** *i* → **chan** **w** *o*: *o* ≡ *ntr*(*i*) **end**

und der prozedurale Agent

agent *ntr* ≡ **chan** **v** *i* → **chan** **w** *o*: **loop skip pool end**

besitzen die gleiche Semantik, nämlich den gesamten Funktionenraum $[\mathbf{V}^{\omega} \rightarrow \mathbf{W}^{\omega}]$.

Als Resultat der syntaktischen und semantischen Verwandtschaft zwischen beiden Sprachen ergibt sich die Möglichkeit, sie in einen gemeinsamen Rahmen einzubetten. Konkret ist damit folgendes gemeint:
Es ist möglich AL- und PL-Programmfragmente zu mischen, die syntaktische Wohlgeformtheit gemischter Darstellungen zu analysieren, und ihnen eine präzise Semantik zuzuweisen. Gemischte Darstellungen haben folgende Form:

program *prg* ≡ … → … :

<< *Funktionsdefinitionen* >>

agent g_1 ≡ … → … : << *appl. Rumpf* >> **end**
…
agent g_m ≡ … → … : << *proz. Rumpf* >> **end**

<< *Gleichungssystem GS oder Anweisung A* >>

end

Ein gemischtes Programm kann also sowohl prozedural als auch applikativ definierte Agenten enthalten. Sein Hauptteil besteht entweder aus einem Gleichungssystem GS oder einer (seq.) Anweisung A. Die Semantik von GS bzw. A ist in beiden Fällen eindeutig bestimmbar. Dies ist eine Konsequenz des kompositionalen Aufbaues der denotationellen Semantik **F**: Die Bedeutung von GS bzw. A hängt von der *Bedeutung* der g_i ab, d.h. von den zugehörigen Mengen stromverarbeitender Funktionen, nicht jedoch von der *Darstellung* der g_i.

Zusammengenommen bilden AL und PL damit eine *Breitbandsprache*, die sowohl einen applikativen als auch einen prozeduralen Programmierstil unterstützt. Für die transformationelle Programmentwicklung bietet das Konzept einer Breitbandsprache wesentliche Vorteile. Viele Transformationsansätze stützen sich daher auf solche Sprachen (vgl. CIP-L in [CIP 85], PANNDAS in [Krieg-Brückner 90], ΦLANG in [Barstow 88] oder die MIX in [Olderog 91]). Entscheidend ist die Möglichkeit inkrementellen Vorgehens:
In einer Breitbandsprache können einzelne Programmteile lokal in einen anderen Darstellungsstil umgeformt werden, ohne das syntaktisch unzulässige und semantisch inkonsistente Programmversionen entstehen. In unserem Fall lassen sich applikative Agenten in prozedurale Agenten umsetzen. Die entstehenden Mischformen sind syntaktisch wohlgeformt (gemäß des obigen Schemas) und besitzen eine präzise Semantik. Der in Abschnitt 5.1 definierte Korrektheitsbegriff für Transformationen, der darauf abhebt, daß eine Transformationsregel in jedem beliebigen Kontext anwendbar sein muß, läßt sich damit auch auf AL/PL-Mischformen anwenden. Durch Regeln der Art

agent *g* ≡ … → … :
<< *appl. Rumpf* >>
end

\O(-
————————;↓)

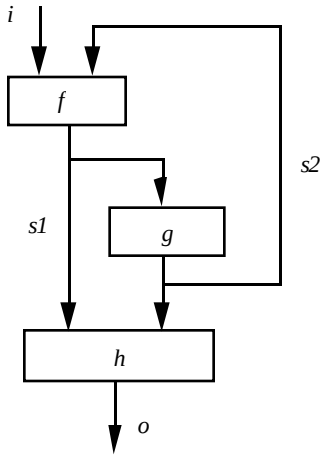
agent $g \equiv \ldots \rightarrow \ldots$:
<< *proz. Rumpf* >>
end,

können AL-Programme schrittweise in prozedurale Form überführt werden. Solche Regeln stehen im Mittelpunkt von Abschnitt 5.3.

Ein wichtiger konzeptueller *Unterschied* zwischen AL und PL der schon mehrfach angeklungen ist, soll nun noch einmal im Detail analysiert werden. Gemeint ist die Verwendung von *Strömen* durch AL und die Verwendung von *Kanälen* durch PL: Ströme werden in AL durch (Strom-)Ausdrücke definiert und in Gleichungssystemen verwendet. Betrachtet man ein gegebenes Gleichungssystem

$s_1 \equiv f(i, s_2)$,
$s_2 \equiv g(s_1)$,
$o \equiv h(s_1, s_2)$

und seine graphische Darstellung als Agentennetz in Figur 4.1. Dann gibt es für jede Kante genau eine Quelle (im Graphen selber oder für Eingangskanten die "Umgebung"), jedoch möglicherweise mehrere Ziele. Der zugehörige Strom tritt dann mehrfach auf rechten Gleichungsseiten auf.



Figur 4.1: Netzdarstellung eines Gleichungssystems

Auf der prozeduralen Ebene bilden Kanäle gerichtete Punkt-zu-Punkt-Verbindungen. Sie haben genau eine Quelle und genau ein Ziel. Aufspaltungen, die in AL durch einfache Mehrfachverwendung modelliert werden, erfordern hier die explizite Verwendung von Verteilerknoten ("*split*").

Darüber hinaus unterscheiden sich die Zugriffsoperationen auf Ströme und Kanäle grundsätzlich von einander. Ströme können mit Hilfe der eingebauten Basisfunktionen **ft**, **rt** und & manipuliert

werden. Der Zugriff auf Kanäle erfolgt mit der Leseoperation $i?x$ und der Schreiboperation $o!E$. Lesen ist dabei nur von Eingabe-, Schreiben nur auf Ausgabekanäle erlaubt.[1] $i?x$ hat einen doppelten Effekt. Es liefert in x das erste auf i befindliche Element und entfernt es gleichzeitig, ändert also den Zustand. Jedes Element kann nur einmal gelesen werden. Soll es mehrfach verwendet werden, so muß es der lesende Agent intern zwischenspeichern. Beim Zugriff auf Ströme besteht kein notwendiger Zusammenhang zwischen dem Lesen eines Elementes, d.h. der Applikation von **ft**., und dessen Entfernung, d.h. der Applikation von **rt**. Die Möglichkeit zur unabhängigen, "seiteneffektfreien" Anwendung von **ft**, **rt** und & erlaubt es einem AL-Agenten, seine Eingangsströme wie eine zusätzliche *Datenstruktur* zu benutzen.

Beispiel 4.4 (Interaktives Sortieren): Der folgende Agent *sort* repräsentiert eine interaktive Sortierkomponente. Er sortiert einen eingehenden Strom natürlicher Zahlen, mit 0 als Trennsymbol zwischen den zu sortierenden Teilsequenzen, auf folgende Weise:

$$7\,3\,1\,2\;\;0\;\;8\,4\,5\;\;0\;\;\ldots\;\;\rightarrow\;\;1\,2\,3\,7\;\;0\;\;4\,5\,8\;\;0\;\;\ldots$$

Erscheint eine 0 in der Eingabe, so wird die Teilsequenz aller bisher eingelesenen Werte (bzw. alle seit der letzten 0 eingelesenen Werte) aufsteigend sortiert ausgegeben.

agent *sort* ≡ **chan nat** $i \rightarrow$ **chan nat** o:
$o \equiv hs(i, \varepsilon)$
end

sort stützt sich auf *hs*, das wie folgt definiert ist:

agent *hs* ≡ **chan nat** $i, s \rightarrow$ **chan nat** o:
$o \equiv$ **if ft.** $i = 0$
then if isempty. s **then ft**.i & hs(**rt.**i, s) **else ft**.s & hs(i, **rt.**s) **fi**
else hs(**rt.**i, *insert*(**ft.** i, s))
fi
end

agent *insert* ≡ **nat** n, **chan nat** $i \rightarrow$ **chan nat** o:
$o \equiv$ **if isempty.** i
then n & ε
else if $n \leq$ **ft**.i **then** n & i **else ft.**i & *insert*(n, **rt.**i) **fi**
fi
end

---

[1] Durch Rückkopplung ist es es möglich, daß eine aktuelle Instanz eines Agenten einen Kanal sowohl als Eingabe- als auch als Ausgabekanal nutzt. Er schickt dann quasi Nachrichten an sich selbst. Dies ist in gewissem Sinn eine Entartung, da der rückgekoppelte Kanal die Aufgabe einer (lokalen) Variablen bzw. eines (lokalen) Pufferspeichers übernimmt.

hs nutzt seinen zweiten Eingabestrom *s* wie einen lokalen Speicher: Solange keine 0 auf *i* erscheint, werden die eingehenden Werte durch *insert* ordnungsverträglich in *s* eingefügt. Erscheint eine 0 wird *s* ausgegeben. □

Ein weiterer Unterschied zwischen AL und PL kristallisiert sich heraus, wenn man einige *Implementierungsüberlegungen* anstellt.

AL kann mit Hilfe der bekannten Implementierungstechniken für funktionale Sprachen implementiert werden. Hier bietet sich insbesondere Graphreduktion an (vgl. [Peyton Jones 87]). Tatsächlich wurde der AL-Vorläufer AMPL auf der Grundlage von Graphreduktion auf einer VAX unter VMS in PASCAL implementiert (vgl. [Nückel 88]) und inzwischen auf eine SUN-SPARC-Station unter UNIX portiert. Aus der Logik des methodischen Ansatzes, in den AL als Beschreibungsmittel eingebettet ist, und der auf die Entwicklung auch räumlich verteilter Systeme abzielt[1], ergibt sich die Notwendigkeit einer verteilten Implementierung. Für Graphreduktion bedeutet Verteilung die Partitionierung des Graphen, der dann von mehreren Prozessoren an mehreren Stellen gleichzeitig bearbeitet, d.h. reduziert wird. (vgl. [Peyton Jones 89]). Diese Partitionierung muß der durch die Gleichungsform eines AL-Programms nahegelegten Prozeßstruktur *nicht* folgen. Methodisch gesehen ist die Struktur Resultat einer Reihe von Entwurfsentscheidungen, deren Zweck das Erreichen einer bestimmte räumlichen Verteilung war. Es erscheint wenig sinnvoll, diese bei der Programmrealisierung zu ignorieren.

In PL ist die Prozeßstruktur noch stärker ausgeprägt: Die sequentiellen Agenten (bzw. ihre aktuellen, durch Aufruf erzeugten Instanzen) entsprechen direkt den sequentiellen Prozessen in einem verteilten Prozeßsystem. Sie können auf unterschiedlichen Prozessoren eines Multiprozessorsystems, aber auch auf vollständig getrennten Rechnern ablaufen. Die Kommunikation erfolgt über Nachrichten. Wie eine Implementierung von PL auf dem INTEL iPSC/2 bzw. iPSC/860 Hypercube auf Grundlage der Münchener Programmbibliothek MMK (vgl. [Bemmerl, Ludwig 90], [Bemmerl et al. 90a, 90b]) aussehen könnte, wird im folgenden kurz diskutiert:

Der MMK (Multiprocessor Multitasking Kernel) ist im Rahmen des TOPSYS-Projekts (Tools for Parallel Systems) am Institut für Informatik der TU München unter der Leitung von T. Bemmerl und A. Bode entstanden und wird gegenwärtig im Sonderforschungsbereich 342 ("Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen") fortentwickelt.
Der MMK stellt ein "transparentes multitasking Prozeßmodell" (vgl. [Bemmerl et al. 90b]) zur Verfügung, das es erlaubt, mehrere Prozesse, *Tasks* in MMK-Sprechweise, zu definieren, diese auf die Knoten des iPSC/2-Multiprozessors zu verteilen und parallel ablaufen zu lassen. Kommunikation und Synchronisation zwischen den Prozessen erfolgt über Postfächer ("mailboxes") und Semaphore, auf die mit Hilfe vordefinierter Operationen zugegriffen werden kann. Prozeßanzahl, Prozeßverteilung auf die Knoten des Hypercubes und die Kommunikationsstruktur sind dynamisch veränderbar. Das heißt, zur Laufzeit können neue Prozesse, durch den Systemaufruf `cretask`, und neue Postfächer, durch den Systemaufruf `crembox`, erzeugt werden. Durch die Wahl der Parameter bei diesen Aufrufen wird den Prozessen mitgeteilt, mit welchen Postfächern sie verbunden sind. Dadurch wird die Kommunikationsstruktur festgelegt. Darüber hinaus besteht die Möglichkeit, die Identifikatoren neu erzeugter Postfächer über bereits

---

[1] Man denke etwa an ein Liftsystem, bestehend aus fest installierter Steuerungskomponente und dem beweglichen Aufzugkorb (vgl. [Broy 88b]), oder an Protokolle (vgl. z.B. [Streicher 87]), in deren Natur es liegt, Sender und Empfänger räumlich zu trennen.

bestehende Postfächer zu verschicken und sie so anderen Prozessen bekannt zu machen. Ein Programm für den MMK besteht aus drei Teilen:

- Erstens: einem *Hostprogramm*, das auf einem Host (das ist in der Regel ein über TCP/IP angeschlossener Arbeitsplatzrechner) abläuft. Dieses Programm übernimmt die Initialisierung des eigentlichen parallelen Programms auf dem Hypercube und realisiert die Kommunikation mit externen Einheiten wie Terminals und Druckern.
- Zweitens: einer Reihe von *Knotenprogrammen*, die von den Knoten (Prozessoren) des Hypercubes ausgeführt werden. Es können mehrere Programme vorhanden sein, von denen jedes einzelne zudem mehrfach instanziiert seien kann. Knoten- und Hostprogramme werden in C und/oder FORTRAN geschrieben. Dabei benutzt der Programmierer zusätzliche, MMK-spezifische Systemaufrufe.
- Drittens: einer sog. *Abbildungsdatei* ("mapping file"), in der die initial, d.h. bei Programmstart, vorhandenen Prozesse und ihre Verknüpfung über Postfächer beschrieben werden. Darüber hinaus legt diese Datei die Abbildung der initialen Objekte (Prozesse und Postfächer) auf die Knoten des Hypercubes fest.

Um PL-Programme auf dem iPSC/2 auf der Basis des MMK zu implementieren, ist es notwendig, sie in die gerade beschriebenen Strukturen zu übersetzen. Konzeptuell bestehen folgende Entsprechungen:

- Kanäle entsprechen den Postfächern.
- (Deklarationen von) PL-Agenten entsprechen den Knotenprogrammen.
- Der Rumpf eines vollständigen PL-Programms (das Hauptprogramm) entspricht der Abbildungsdatei.

Auf Postfächer kann mit den Systemaufrufen `recmsg` lesend und `sndmsg` schreibend zugegriffen werden. Durch geeignete Parametrisierung läßt sich die Semantik der PL-Operatoren ? und ! nachbilden: Die PL-Anweisung *i*?*x* entspricht dabei folgendem MMK-Aufruf (im "C-Stil", vgl. [Bemmerl et al. 90b], S. 49):

```
x = (‹type_of_x› *) recmsg(i, UNLIMITED, &reply)
```

`i` ist hier der Name des Postfachs von dem die Nachricht gelesen werden soll und `reply` ist eine Integer-Variable, die in codierter Form Information über den Erfolg des Systemaufrufs enthält. Der Parameter `UNLIMITED` zeigt an, daß der Prozeß P, der diesen Aufruf absetzt, auf dessen vollständige Abarbeitung warten muß. Insbesondere wird P blockiert, wenn keine Nachricht im Postfach `i` vorhanden ist und zwar solange bis eine Nachricht eintrifft. Gelesene Nachrichten werden aus `i` entfernt. Dies ist genau die Semantik von ?.

`recmsg` liefert als Ergebnis einen Zeiger vom Typ `char` auf die Nachricht. Das vorgeschaltete `(‹type_of_x› *)` bewirkt eine Typanpassung. Beachte, daß `x` hier als Zeiger interpretiert wird.
Um asynchrone Kommunikation zu gewährleisten, muß darüber hinaus das Postfach `i` so dimensioniert werden, daß es (zumindest theoretisch) unbeschränkt viele Nachrichten aufnehmen kann. Dies ist möglich, indem man bei seiner Erzeugung ebenfalls den Parameter `UNLIMITED` verwendet. Die Anzahl der speicherbaren Nachrichten ist dann nur durch den verfügbaren Speicherplatz beschränkt (siehe unten).

Die PL-Anweisung *o*!E entspricht folgendem MMK-Aufruf (vgl. [Bemmerl et al. 90b], S. 53):

```
x = E;
sndmsg(o, &x, sizeof(‹type_of_x›), UNLIMITED,
&reply)
```

`o` ist der Name des Postfachs auf den die Nachricht geschrieben wird, `reply` hat die gleiche Funktion wie oben. `x` ist eine Variable vom selben Typ wie der Ausdruck `E`. `&x` liefert einen Zeiger auf diese Variable. `sndmsg` erwartet einen Zeiger auf die zu übermittelnde Nachricht auf der zweiten Parameterposition und eine Angabe über die "Größe" der Nachricht auf der dritten. Wiederum bedeutet `UNLIMITED`, daß der aufrufende Prozeß auf die Abarbeitung des Aufrufs warten muß.
Ein Sendebefehl kann verzögerungsfrei ausgeführt werden, wenn in dem angesprochenen Postfach noch Platz für die Aufnahme der Nachricht vorhanden ist. Obwohl die Kapazität von Postfächern (durch Wahl des Parameters `UNLIMITED` bei ihrer Erzeugung durch `crembox`) nur durch den verfügbaren Speicherplatz beschränkt ist, können sie natürlich trotzdem nur endlich viele Nachrichten speichern. Die idealisierte Annahme von PL, daß Kanäle wirklich unbeschränkte Kapazitäten besitzen, ist auf der realen Maschine nicht zu verwirklichen. Daraus resultiert eine Abweichung zwischen der Semantik eines PL-Programms und der zugehörigen MMK-Version: Aufgrund fehlenden Speicherplatzes in einem Postfach, in das ein MMK-Prozeß schreiben möchte, kann es passieren, daß er blockiert wird. In ungünstigen Fällen können dadurch zyklische Wartesituationen (Deadlocks) entstehen, die auf der PL Ebene nicht möglich sind. Es ist nicht ganz einfach, dieses Phänomen zu vermeiden:
Pragmatisch kann man versuchen, die Kapazitäten der Postfächer immer so zu dimensionieren, daß sich dadurch keine reale Beschränkungen ergeben. Theoretisch fundierter wäre ein Ansatz, der die Endlichkeit der Ressourcen schon auf der PL-Ebene berücksichtigt und zum Beispiel Mechanismen zur Flußkontrolle schon in die PL-Programme aufnimmt. Dadurch wird jedoch die logische Struktur der Programme beeinträchtigt. "Optimal" wäre daher eine "Compiler-Lösung", d.h. ein Compiler von PL nach MMK, der die zusätzlichen Strukturen zur Flußkontrolle (Nachrichten und/oder Kanäle) automatisch in das MMK-Programm einfügt und so den Programmierer von diesen stark implementierungsabhängigen Details entlastet. Hier sind noch viele Fragen offen. Auch bei "naiver" Übersetzung ist die MMK-Implementierung aber partiell korrekt in Bezug auf das PL-Programm.

Für sequentielle PL-Agenten ist die Beziehung zu MMK-Knotenprogrammen besonders eng. Bei der Übersetzung eines seq. PL-Agenten braucht man Zuweisungen, Schleifen, Alternativen usw. nur in der konkreten Syntax von C oder FORTRAN darstellen. Die Kommunikationsanweisungen ? und ! müssen dann wie oben beschrieben ersetzt werden.

Bei der Ausführung hierarchischer Agenten entstehen zur Laufzeit neue (Instanzen von) Agenten und entsprechend neue Kanäle. Dies ist in PL implizit repräsentiert und muß auf MMK-Ebene

ausprogrammiert werden. Dazu kann man die Systemaufrufe `crembox` und `cretask` heranziehen. `crembox` erzeugt ein neues Postfach und `cretask` erzeugt eine neue Instanz eines Knotenprogramms, d.h. eine neue Instanz eines Agenten. Auch rekursive PL-Agenten können so programmiert werden. Das MMK-Programm für den hierarchischen Agenten *sieve* aus Beispiel 4.2 sieht wie folgt aus:

```
TASK (sievecode, i, o)
{
        int reply;
        /* Nummern der Knoten, auf die Postfächer und Prozesse plaziert werden sollen */
        int node1, …, node8;
        /* Größe der Stacks für die Prozesse, die erzeugt werden */
        int stack1, …, stack4;
        /* Identifikatoren für Postfächer */
        MBOX_ID s1, …, s4;
        /* Identifikatoren für Prozesse */
        TASK_ID split, filter, sieve, join;

        /* Einstellen der Stackgrößen */
        stack1 = … ;
        …
        stack4 = … ;

        /* Verteilung der Prozesse und Postfächer auf die Knoten */
        node1 = … ;
        …
        node8 = … ;

        /* Erzeugung der Postfächer, die die internen Kanäle repräsentieren */
        s1 = crembox(node1, UNLIMITED, &reply);
        s2 = crembox(node2, UNLIMITED, &reply);
        s3 = crembox(node3, UNLIMITED, &reply);
        s4 = crembox(node4, UNLIMITED, &reply);

        /*   Erzeugung der Prozesse, d.h. der Agenten(instanzen), die auf den rechten

        **  Gleichungsseiten von Beispiel 4.2 aufgerufen werden */
        split = cretask(splitcode, node5, stack1, &reply, 3, i, s1,
s2);
        filter = cretask(filtercode, node6, stack2, &reply, 2, s2,
s3);
        sieve = cretask(sievecode, node7, stack3, &reply, 2, s3,
s4);
        join = cretask(joincode, node8, stack4, &reply, 3, s2, s3,
s4);

    }
```

Die `cretask`-Aufrufe erzeugen jeweils neue Instanzen derjenigen Knotenprogramme auf die im ersten Parameter des Aufrufs verwiesen wird. `cretask(sievecode, … )` erzeugt also eine neue Instanz des obigen Programms selber.

Man beachte, daß dies Programm aufgrund seiner rekursiven Struktur dazu führt, daß unendlich viele neue Prozesse entstehen. Dadurch stößt man relativ schnell an die Kapazitätsgrenzen der einzelnen Knoten, die allerdings in neueren Systemversionen nur doch durch den verfügbaren Speicherplatz gezogen werden. Für das eher theoretische Erathostenes-Beispiel ist die unendliche Prozeßanzahl wesentlich, man braucht ja für jede Primzahl einen eigenen Filter. Für realistische Anwendungen ist es sinnvoll, einen Terminierungsfall vorzusehen, bei dem die Prozeßerzeugung, eventuell in Abhängigkeit von den gelesenen Nachrichten (adaptive Lösung), abbricht.

Prozesse/Agenten, die ihre Aufgaben erfüllt haben oder von denen klar ist, daß die von ihnen produzierten Nachrichten nicht mehr benötigt werden, können mit dem Aufruf *deltask* gelöscht werden.

Das Gleichungssystem im Rumpf eines vollständigen (parallelen) PL-Programms legt die initiale Struktur des Agentennetzes fest, d.h. die bei Programmstart existierenden Agenteninstanzen und ihre Verknüpfung über Kanäle. Genau dies ist die Aufgabe der Abbildungsdatei auf MMK-Ebene. Zusätzlich bestimmt sie die Zuordnung der initialen Objekte zu den Knoten des Rechners.

Effizienzgründe machen es notwendig, bei der Auswertung der beschriebenen Programme ein ausgewogenes Mittelmaß zwischen Netzauffaltung (Prozeßerzeugung) und sequentieller Auswertung zu finden. Dies ist eng verwandt zu dem aus der funktionalen Programmierung bekannten Problem, "lazy evaluation" und "eager evaluation" angemessen zu kombinieren (vgl. [Peyton Jones 89]).

5. Transformationelle Programmentwicklung

Methodisches Ziel des in Kapitel 1 beschriebenen Gesamtansatzes, ist die systematische Entwicklung verteilter Systeme. Am Anfang steht dabei eine abstrakte Anforderungsspezifikation, am Ende ein konkretes Programm, das über mehrere Entwicklungsstufen aus der Spezifikation heraus entwickelt wurde und diese beweisbar korrekt realisiert. Während des Entwicklungsprozesses kommen unterschiedliche, aber auf einander abgestimmte Formalismen zum Einsatz. AL und PL sind in diesen Rahmen eingepaßt. AL ist die Sprache für *abstrakte Programme* (Phase 3). PL ist die Sprache für *konkrete Programme* (Phase 4).
In diesem Kapitel soll aufgezeigt werden, wie der Übergang von AL nach PL zu bewältigen ist. Der verfolgte Ansatz ist *deduktiv*: AL-Programme sollen durch iterierte Anwendung korrektheitserhaltender *Transformationsregeln* in prozedurale Form überführt werden.
In Abschnitt 5.1 gehen wir kurz auf die Grundlagen transformationeller Programmentwicklung ein, um dann in Abschnitt 5.2 zuerst Transformationsregeln für applikative Programme und schließlich in Abschnitt 5.3 Regeln für den Übergang von AL nach PL zu untersuchen.

5.1 Grundlagen

Der Einsatz transformationeller Entwicklungstechniken zielt im Allgemeinen darauf ab, aus einer (abstrakten) Spezifikation eine (konkrete) Implementierung abzuleiten. Voraussetzung für die Angabe von Transformationsregeln ist daher ein *Implementierungskonzept*:
Für nichtdeterministische, nicht-flache Sprachen wie AL und PL lassen sich eine ganze Reihe unterschiedlicher Implementierungsbegriffe angeben[5]. In dieser Arbeit wird allerdings nur einer behandelt.

Seien *prg*, *prg'* zwei Programme, dann wird ihre Semantik durch Mengen stetiger Stromfunktionen beschrieben:

$$\mathbf{F}[\![prg']\!], \mathbf{F}[\![prg]\!] \subseteq [(\mathbf{Dom}^{\omega})^n \rightarrow (\mathbf{Dom}^{\omega})^m].$$

Jede Funktion $f \in \mathbf{F}[\![prg]\!]$ repräsentiert ein mögliches Verhalten von *prg*.
Je mehr unterschiedliche Verhalten *prg* zeigen kann, desto weniger determiniert ist es. Ein deterministisches Programm zeigt genau ein Verhalten: $|\mathbf{F}[\![prg]\!]| = 1$.

---

[5] Mindestens 3*2 verschiedene Relationen erscheinen sinnvoll: Eine Implementierung kann *partiell*, *total* oder *robust korrekt* sein und zusätzlich (und orthogonal dazu) *vollständig* oder *ausschnitthaft*. Für Details siehe [Broy 85] oder [Berghammer et al. 90].

prg implementiert prg', wenn jedes Verhalten von *prg* auch ein Verhalten von *prg'* ist. Dies schießt nicht aus, daß *prg'* noch andere Verhalten zeigen kann, d.h. *prg'* kann nichtdeterministischer sein als *prg*. Formal:

$$\mathbf{F}[\![prg]\!] \subseteq \mathbf{F}[\![prg']\!].$$

prg und *prg'* sind *äquivalent* wenn sie vollständig identische Verhalten zeigen:

$$\mathbf{F}[\![prg]\!] = \mathbf{F}[\![prg']\!].$$

Diese Implementierungsrelation deckt sich damit mit dem in Abschnitt 3.3 eingeführten Abkömmlingsbegriff: Abkömmlings- und Implementierungsrelation sind identisch.

Eine *Transformationsregel* ist eine Abbildung zwischen Programmen bzw. Programmfragmenten. In dieser Arbeit werden solche Regeln wie folgt notiert:

.

I ist das *Eingabetableau* der Regel, O das *Ausgabetableau* und C eine Formel über Programmen bzw. Programmfragmenten, die mit Hilfe sprachspezifischer Attribute definiert wird (siehe unten). C heißt *Anwendungsbedingung*.
I und O sind von gleicher syntaktischer Bauart. Im Allgemeinen handelt es sich jedoch nicht direkt um Elemente aus den syntaktischen Bereichen ‹exp›, ‹function›, ‹agent› oder ‹program›, sondern um *Ausdrucks-*, *Funktions-*, *Agenten-* oder *Programmschemata*.
Ein Programmschema enthält neben den definierten Sprachkonstrukten noch sog. *Schemavariablen*. Eine Schemavariable ist ein Platzhalter für Ausdrücke, Sorten, Bezeichner usw..
Im Lichte einer algebraischen Auffassung von Programmiersprachen ist ein Programm ein Element aus $W(\Sigma)$, der Menge aller wohlgeformten Terme über der Sprachsignatur Σ. Diese Signatur ist hier konkret durch die Grammatiken in BNF-Form beschrieben. Ein Programmschema ist ein Element aus $W(\Sigma \cup X)$, wobei X eine Menge getypter Schemavariablen repräsentiert. Jedes Programm ist damit auch ein Programmschema, nämlich eines, das keine Schemavariablen enthält.
Eine *Instanz* (eines Programmschemas) ist eine Abbildung $\Theta: X \rightarrow W(\Sigma \cup X)$, die jeder Schemavariable ein passendes Programmschema zuordnet. Für ein Programmschema P bezeichnet $P\Theta$ das Schema, das entsteht, wenn alle Schemavariablen in P durch die von Θ vorgegebenen Werte ersetzt werden. Θ heißt *Grundinstanz*, wenn jeder Schemavariablen ein Programm, d.h. ein Element aus $W(\Sigma)$ zugeordnet wird. Details zu diesen Begriffen lassen sich in [Pepper 87] bzw. [Partsch 90] nachlesen.

Eine Anwendungsbedingung C ist eine Liste $C_1, \ldots, C_n$ von Implikationsformeln

$A_1, \ldots, A_m \Rightarrow B$ bzw. B.

Dabei sind die A_i, B atomare Formeln der Gestalt

$PRED[P_1, \ldots, P_n]$

und PRED ist ein sprachspezifisch definiertes Prädikat (Attribut). Die P_i sind Programmschemata.

Sei Θ eine Grundinstanz, C eine Anwendungsbedingung und $\delta \in ENV$ eine Umgebung. $C\Theta$ heißt *gültig bzgl.* δ, wenn $\delta \models C\Theta$ eine wahre Aussage ist. $\delta \models C\Theta$ ist dabei induktiv erklärt:

$$\delta \models (C_1, \ldots, C_n)\Theta \quad \Leftrightarrow \quad \delta \models C_1\Theta \wedge \ldots \wedge \delta \models C_n\Theta,$$
$$\delta \models (A_1, \ldots, A_m \Rightarrow B)\Theta \quad \Leftrightarrow \quad \delta \models A_1\Theta \wedge \ldots \wedge \delta \models A_m\Theta \Rightarrow \delta \models B\Theta$$

Aussagen über atomare Formeln $\delta \models A_i\Theta$ werden mit Hilfe der semantischen Funktion **F** erklärt. Wir verwenden dabei die folgenden Attribute (X, Y seien aus derselben syntaktischen Kategorie):

$\delta \models X \subseteq Y \quad \Leftrightarrow \quad \mathbf{F}_\delta[\![X]\!] \subseteq \mathbf{F}_\delta[\![Y]\!]$,

$\delta \models X = Y \quad \Leftrightarrow \quad \mathbf{F}_\delta[\![X]\!] = \mathbf{F}_\delta[\![Y]\!]$,

$\delta \models DET\ X \quad \Leftrightarrow \quad |\mathbf{F}_\delta[\![X]\!]| = 1$,

$\delta \models DEF\ X \quad \Leftrightarrow \quad \forall f \in \mathbf{F}_\delta[\![X]\!]$: f ist überall definiert (liefert niemals $\perp$).

Diese Attribute erfassen semantische Eigenschaften von X und Y. Zum Beispiel ist $X \subseteq Y$ der syntaktische Ausdruck dafür, daß das Programm(fragment) X das Programm(fragment) Y implementiert.
Neben den semantischen Attributen verwenden wir syntaktisch überprüfbare Attribute. Diese hängen nicht von δ ab (*x* sei ein beliebiger Bezeichner):

NOTOCCURS *x* IN X $\Leftrightarrow$ *x* kommt nicht in X vor

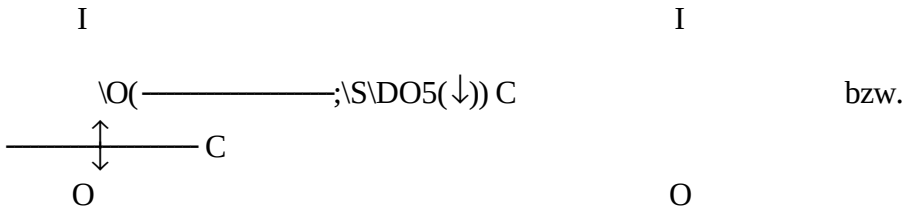
NEW *x* $\Leftrightarrow$ *x* ist ein frischer Bezeichner, er kommt nicht im Eingabetableau der Regel vor

In den nachfolgenden Regeln werden einige weitere Attribute verwandt. Sie sind weitgehend selbsterklärend.

Eine Anwendungsbedingung $C\Theta$ heißt *gültig*, notiert durch $\models C\Theta$, wenn gilt:

$\forall \delta: \delta \models C\Theta$

Darauf aufbauend läßt sich ein Korrektheitsbegriff für Transformationsregeln festlegen: Eine Transformationsregel

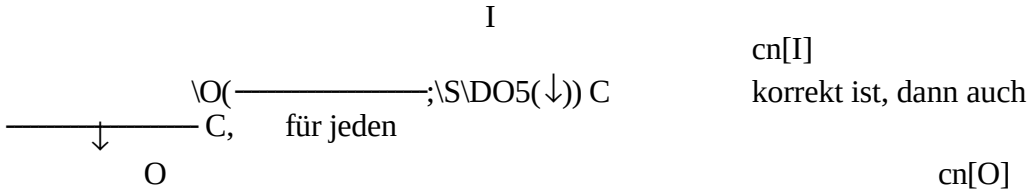


bzw.

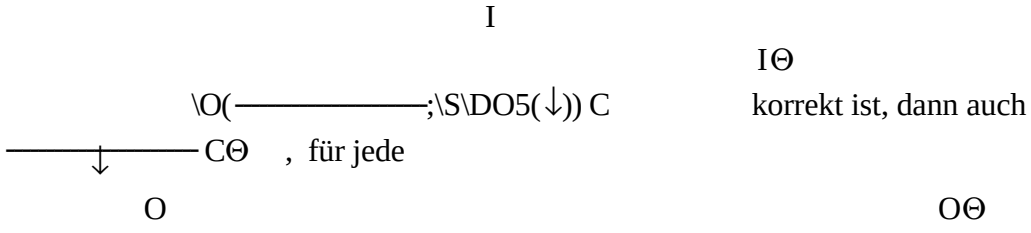
ist *korrekt*, wenn für alle Grundinstanzen Θ gilt:

$$\models C\Theta \Rightarrow \models (I\Theta \subseteq O\Theta) \qquad \text{bzw.} \qquad \models C\Theta \Rightarrow \models (I\Theta = O\Theta)$$

Beachte, daß aus dieser Definition folgt, daß Transformationsregeln *lokal* angewandt werden können:
$\models (I\Theta = O\Theta)$ besagt definitionsgemäß, daß $I\Theta$ in allen Umgebungen δ zu $O\Theta$ äquivalent ist. Umgebungen aber verkörpern Kontextinformation: Ein Programmkontext cn[.], d.h. ein vollständiges Programm mit einer ausgezeichneten Stelle, definiert semantisch betrachtet, eine Menge von Umgebungen δ für dieses Stelle. Nämlich die Menge aller $\delta \in$ ENV, die mit den Funktions- und Agentendeklarationen in cn[.] konsistent sind (vgl. Satz 3.23). Da $I\Theta$ in alle Umgebungen δ zu $O\Theta$ äquivalent ist, kann es auch in allen Kontexten cn[.] durch $O\Theta$ ersetzt werden. Formal gilt also die wichtige Aussage:



beliebigen Kontext cn[.]. Weiterhin liefert jede (Teil-)Instanzierung einer korrekten Regel wider eine korrekte Regel:



Instanz Θ.

Die *Anwendung* einer Regel auf ein gegebenes Programm *prg* erfordert drei Aktivitäten:

- Finde einen Programmteil *p* von *prg* und eine Grundinstanz Θ dergestalt, daß $I\Theta = p$.
- Instanziere die Anwendungsbedingung C mit dem gefundenen Θ und überprüfe, ob $C\Theta$ gültig ist. Falls ja,
- Ersetze $I\Theta = p$ in *prg* durch $O\Theta$.

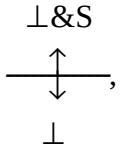
Das so entstehende neue Programm heiße *prg'*. Aufgrund des Korrektheitsbegriffs für Transformationen folgt, daß *prg'* das ursprüngliche Programm *prg* implementiert bzw. zu ihm äquivalent ist.

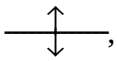
5.2 Transformation von AL-Programmen

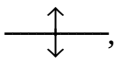
In diesem Abschnitt geht es um Transformationen, die AL-Programme bzw. AL-Programmfragmente untereinander in Beziehung setzen. Gemäß der in Kapitel 1 erläuterten Phasengliederung, ermöglichen sie transformationelle Entwicklungsschritte innerhalb der dritten Stufe des gesamten Entwicklungsprozesses. Solche Transformationen können der Vereinfachung und Optimierung *abstrakter* Programme dienen, etwa wenn Ausdrücke symbolisch ausgewertet oder redundante Gleichungen eliminiert werden. Methodisch besteht ihr Hauptzweck jedoch darin, den Übergang zu *konkreten*, d. h. prozeduralen Programmen vorzubereiten. Transformationen, die diesen Übergang tatsächlich leisten bzw. dezidiert daraufhin arbeiten, finden sich in Abschnitt 5.3. Im folgenden gehen wir kurz auf Transformationsregeln für Ausdrücke und dann ausführlicher auf Transformationsregeln für Gleichungssysteme ein. Die *Korrektheit* der Regeln wird formal nachgewiesen und zwar bezüglich der denotationellen Semantik von AL. Die Definitionen aus Kapitel 3 bilden daher die entscheidende Grundlage der Korrektheitsbeweise.

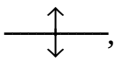
Ausdruckstransformationen gehören zum Standardrepertoire von Transformationskalkülen für applikative Sprachen (vgl. [Backus 78], [CIP 85], [Bird 89]). Aus der denotationellen Semantik von AL lassen sich solche Regeln direkt ableiten. Zum Beispiel sind die definierenden Axiome für die Stromoperatoren **ft**, **rt**, **isempty** und & (vgl. Kapitel 2) unmittelbar als Transformationsregeln verwendbar:

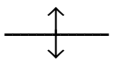
Transformation *streamop***:** Abgeleitete Regeln für Stromoperatoren

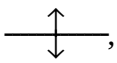


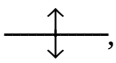
ft.E&S  , E

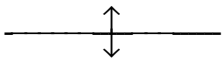
ft.ε  , ⊥

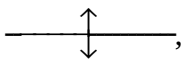
ft.⊥  , ⊥

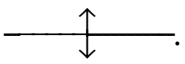
rt.E&S  DEF E, S

rt.ε  , ⊥

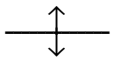
rt.⊥  , ⊥

isempty.E&S  DEF E, false

isempty.ε  , true

isempty.⊥  . ⊥

Korrektheitsbeweis: Der Korrektheitsbeweis ist für alle Regeln leicht zu erbringen.

Exemplarisch sei er für  (**rt.**E&S / S) DEF E ausgeführt.

Θ sei eine beliebige Grundinstanz. Wir identifizieren im folgenden EΘ mit E und SΘ mit S. O.b.d.A sei $x \in$ ID der einzige Bezeichner der in E&S vorkommt. Dann ist die Anwendungsbedingung DEF E äquivalent zu: $\forall\delta: \forall\sigma: \bot \notin \mathbf{B}_{\delta,\sigma}[\![E]\!]$ und es gilt für beliebiges $\delta \in$ ENV:

$$\begin{aligned}
& \mathbf{F}_\delta[\![\mathbf{rt.}E\&S]\!] \\
= \quad & \{\text{ Definition von } \mathbf{F} \} \\
& \{\lambda x.\mathbf{B}_{\delta',\sigma[x/x]}[\![\mathbf{rt.}E'\&S']\!] \mid E' \in DD(\tilde{E}),\ S' \in DD(\tilde{S}),\ \delta' \in DD(\tilde{\delta}),\ \sigma \in STATE\} \\
= \quad & \{\text{ Definition von } \mathbf{B} \} \\
& \{\lambda x.rt(\mathbf{B}_{\delta',\sigma[x/x]}[\![E']\!] \& \mathbf{B}_{\delta',\sigma[x/x]}[\![S']\!]) \mid \ldots \text{ " } \ldots\} \\
= \quad & \{\text{ Definition von rt und Applikationsbedingung } \} \\
& \{\lambda x.\mathbf{B}_{\delta',\sigma[x/x]}[\![S']\!] \mid \ldots \text{ " } \ldots\} \\
= \quad & \{\text{ Definition von } \mathbf{F} \} \\
& \mathbf{F}_\sigma[\![S]\!]
\end{aligned}$$

□

Die Semantikdefinition für AL stützt sich wesentlich auf die Eigenschaften des *Auswahloperators*. Diese sind hier ebenfalls als Transformationsregeln formalisiert: $\square$ ist kommutativ, assoziativ und idempotent. Er distributiert über alle anderen Konstrukte und ermöglicht die Abkömmlingsbildung. Durch gerichtete Anwendung dieser Regeln kann jeder Ausdruck so umgeformt werden, daß der Auswahloperator, wenn überhaupt, nur auf der äußersten Termebene auftritt:

Transformation *choice***:** Abgeleitete Regeln für $\square$

$$E_1 \square E_2 \updownarrow, \quad E_2 \square E_1$$

$$(E_1 \square E_2) \square E_3 \updownarrow, \quad E_1 \square (E_2 \square E_3)$$

$$E \square E \updownarrow, \quad E$$

$$E[E_1 \square E_2] \updownarrow, \quad E[E_1] \square E[E_2]$$

$$E_1 \square E_2 \downarrow, \quad E_1$$

$$E_1 \square E_2 \downarrow. \quad E_2$$

Korrektheitsbeweis: Analog zu oben. Die Aussage

$$\mathbf{F}_\delta[\![(E[E_1 \square E_2])]\!] = \mathbf{F}_\delta[\![(E[E_1] \square E[E_2])]\!]$$

muß durch Induktion über den Aufbau von E nachgewiesen werden.

□

Weitere Regeln, z.B. für **if.then.else.fi**, lassen sich analog ableiten und beweisen. Wir verzichten darauf und wenden uns nun den für AL typischen Gleichungssystemen zu.

Die *Transformation von Gleichungssystemen*, bildet die typische Entwicklungsaktivität innerhalb der dritten Phase der Entwurfsmethodik. Durch die Transformation von Gleichungen wird die Verteilungs- und Verbindungsstruktur des entworfenen (Software-)Systems verändert. Wie in Abschnitt 3.5 ausgeführt, entsprechen AL-Gleichungssysteme Netzen kommunizierender Agenten. Die Transformation von Gleichungssystemen entspricht daher der Transformation der zugehörigen Netze: Wir sprechen von *Netztransformationen*.

In einem AL-Programm treten Gleichungssysteme stets "verkapselt" auf: Sie bilden entweder den Rumpf eines Agenten oder den Gleichungsteil des Gesamtprogramms. Das hat zur Folge, daß jeder auftretende Strom(bezeichner) eindeutig als Eingabe-, Ausgabe- oder interner Strom gekennzeichnet werden kann. Darüber hinaus gelten einige Transformationsregeln, wie etwa das Vertauschen von Gleichungen oder das Weglassen unnötiger Gleichungen, die für "freie" Systeme keine Gültigkeit besäßen. Die folgenden Regeln behandeln daher stets Gleichungssysteme, die den Rumpf von Agenten bilden. Sie sind kanonisch auf vollständige Programme zu übertragen.

Transformation *eval***:** Substitution von Ausdrücken (Auswerten)

agent $f \equiv \text{IN} \rightarrow \text{OUT}$:
$s_1 \equiv S_1, \ldots, s_j \equiv S_j, \ldots, s_n \equiv S_n$
end

$\backslash O(- \; S_j' \subseteq S_j$
————————;↓)

agent $f \equiv \text{IN} \rightarrow \text{OUT}$:
$s_1 \equiv S_1, \ldots, s_j \equiv S_j', \ldots, s_n \equiv S_n$
end

Korrektheitsbeweis: Folgt aus der Monotonie von **F** (vgl. Satz 3.21).
□

Die Substitutionsregel ist die direkte (und eigentlich redundante) Umsetzung der Kompositionalitäts- und Monotonieresultate aus Abschnitt 3.3. Der für die Anwendung notwendige Nachweis, $S_j' \subseteq S_j$, kann natürlich mit Hilfe von Ausdruckstransformationen geführt werden. Die Regel besagt also nichts anderes, als das Ausdruckstransformationen im Rumpf von Agenten angewandt werden dürfen.

Der Rumpf eines Agenten besteht aus einem System rekursiver Stromgleichungen. Unter gewisse Einschränkungen kann man hier die üblichen gleichungsorientierten Umformungen ausführen.

Transformation *streamunfold***:** Auffalten von Stromdefinitionen

agent $f \equiv \text{IN} \rightarrow \text{OUT}$:
$s_1 \equiv S_1, \ldots, s_j \equiv S_j, \ldots, s_k \equiv S_k, \ldots, s_n \equiv S_n$
end

$\backslash O(- \; \text{DET } S_k$
————————;↓)

agent $f \equiv \text{IN} \rightarrow \text{OUT}$:
$s_1 \equiv S_1, \ldots, s_j \equiv S_j[S_k/s_k], \ldots, s_k \equiv S_k, \ldots, s_n \equiv S_n$
end

Transformation *streamfold***:** Falten von Stromdefinitionen

agent $f \equiv \text{IN} \rightarrow \text{OUT}$:

$s_1 \equiv S_1, \ldots, s_j \equiv S_j[S_k/s_k], \ldots, s_k \equiv S_k, \ldots, s_n \equiv S_n$

end

\O(-

———————;↓)

agent $g \equiv \text{IN} \rightarrow \text{OUT}$:

$s_1 \equiv S_1, \ldots, s_j \equiv S_j, \ldots, s_k \equiv S_k, \ldots, s_n \equiv S_n$

end

Korrektheitsbeweis: Um die Korrektheit beider Regeln nachzuweisen, ist es hinreichend, die Inklusionsbeziehungen zwischen den Rumpfgleichungssystemen zu zeigen. Wir nehmen an, daß i der einzige Eingabeparameter von f ist. Dann gilt für beliebige Umgebungen δ und Stromausdrücke S_i:

$$\mathbf{F}_\delta[\![s_1 \equiv S_1, \ldots, s_j \equiv S_j[S_k/s_k], \ldots, s_k \equiv S_k, \ldots, s_n \equiv S_n]\!]$$

$=$ { Definition von **F** }

$$\{\ \lambda i.\mathrm{fix}\begin{pmatrix} s_1 = f_1(i, s_1, \ldots, s_n) \\ \ldots \\ s_j = f_j^*(i, s_1, \ldots, s_n) \\ \ldots \\ s_k = f_k(i, s_1, \ldots, s_n) \\ \ldots \\ s_n = f_n(i, s_1, \ldots, s_n) \end{pmatrix} \mid f_i \in \mathbf{F}_\delta[\![S_i]\!], f_j^* \in \mathbf{F}_\delta[\![S_j[S_k/s_k]]\!]\ \}$$

$\subseteq$ { Korollar 3.19 }

$$\{\ \lambda i.\mathrm{fix}\begin{pmatrix} s_1 = f_1(i, s_1, \ldots, s_n) \\ \ldots \\ s_j = f_j(i, s_1, \ldots, f_k(i, s_1, \ldots, s_n), \ldots, s_n) \\ \ldots \\ s_k = f_k(i, s_1, \ldots, s_n) \\ \ldots \\ s_n = f_n(i, s_1, \ldots, s_n) \end{pmatrix} \mid f_i \in \mathbf{F}_\delta[\![S_i]\!]\ \}$$

$=$ { Gleichungslogik }

$$\{\ \lambda i.\mathrm{fix}\begin{pmatrix} s_1 = f_1(i, s_1, \ldots, s_n) \\ \ldots \\ s_j = f_j(i, s_1, \ldots, s_n) \\ \ldots \\ s_k = f_k(i, s_1, \ldots, s_n) \\ \ldots \\ s_n = f_n(i, s_1, \ldots, s_n) \end{pmatrix} \mid f_i \in \mathbf{F}_\delta[\![S_i]\!]\ \}$$

$=$ { Definition von **F** }

$$\mathbf{F}_\delta[\![s_1 \equiv S_1, \ldots, s_j \equiv S_j, \ldots, s_k \equiv S_k, \ldots, s_n \equiv S_n]\!]$$

Wenn DET S_K erfüllt ist, dann gilt gemäß Korollar 3.19 sogar Gleichheit und damit die Korrektheit der ersten Regel. □

Allen drei Transformationen ist gemeinsam, daß die Anzahl der Gleichungen unverändert bleibt. Bezogen auf die zugehörigen Netzdarstellungen (vgl. Abschnitt 3.5) heißt das, daß sich zwar die Verknüpfungsstruktur ändern kann, die Knotenzahl aber konstant bleibt. Die Knotenzahl eines Netzes verändert sich genau dann, wenn Gleichungen hinzugefügt oder weggelassen werden. Dazu dienen die folgenden Regeln. Das syntaktische Attribut INTERNAL *s* kennzeichnet den Strom *s* als intern.

Transformation *netreduct***:** Weglassen von Gleichungen (Netzreduktion)

agent $f \equiv \text{IN} \rightarrow \text{OUT}$:
$s_1 \equiv S_1, \ldots, s_j \equiv S_j, \ldots, s_n \equiv S_n$
end

$$\frac{\text{INTERNAL } s_j,\ \text{NOTOCCURS } s_j \text{ IN } S_1, \ldots, S_{j-1}, S_{j+1}, \ldots, S_n}{;\downarrow}$$

agent $f \equiv \text{IN} \rightarrow \text{OUT}$:
$s_1 \equiv S_1, \ldots, s_{j-1} \equiv S_{j-1}, s_{j+1} \equiv S_{j+1}, \ldots, s_n \equiv S_n$
end

Transformation *netexpand***:** Einführen von Gleichungen (Netzexpansion)

agent $f \equiv \text{IN} \rightarrow \text{OUT}$:
$s_1 \equiv S_1, \ldots, s_{j-1} \equiv S_{j-1}, s_{j+1} \equiv S_{j+1}, \ldots, s_n \equiv S_n$
end

$$\frac{\text{NEW } s_j}{;\downarrow}$$

agent $f \equiv \text{IN} \rightarrow \text{OUT}$:
$s_1 \equiv S_1, \ldots, s_j \equiv S_j, \ldots, s_n \equiv S_n$
end

Korrektheitsbeweis: Folgt direkt aus der Definition von $\mathbf{F}_\delta[\![\textbf{agent } f \equiv \ldots \textbf{ end}]\!]$: Die Verkapselung der Gleichungssysteme in den Rümpfen hat zur Folge, daß alle Lösungstupel auf die Ausgabeströme des Agenten projiziert werden. Unter den gegebenen Anwendungsbedingungen

trägt s_j zu deren Werten nichts bei.

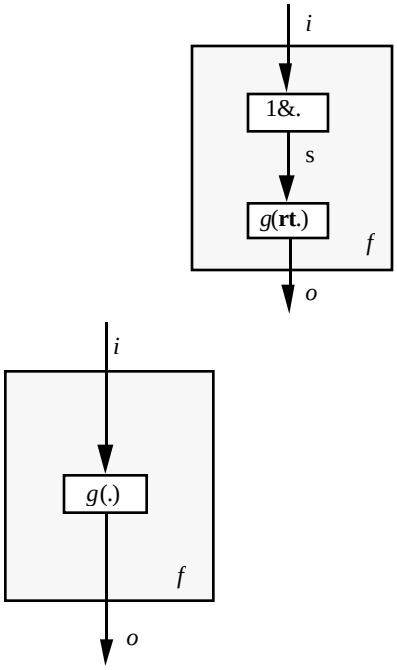
□

Dieses Regelpaar ermöglicht das Einführen bzw. Fortlassen redundanter Gleichungen. *Redundant* ist eine Gleichung dann, wenn sie einen (internen) Strom definiert, der zu keinem der Ausgabeströme einen mittel- oder unmittelbaren Beitrag liefert. Redundanz ist eine semantische Eigenschaft, die angegebene Regel stützt sich auf ein syntaktisches und deshalb hinreichendes Kriterium. Eine Erweiterung auf ganze (Sub-)Systeme redundanter Gleichungen läßt sich analog formulieren. Beide Regeln sind extrem einfach. Sie können sich aber mit den vorangegangenen Regeln zu komplexeren Transformationen verknüpft werden.

Beispiel 5.1 (Verknüpfung der Elementarregeln): a) Durch Verknüpfung der Regeln *streamunfold*, *eval* und *netreduct* lassen sich unnötige Gleichungen eliminieren:

$$s \equiv 1\&i, \quad o \equiv g(\mathbf{rt}.s) \xrightarrow{\textit{streamunfold}} s \equiv 1\&i, \quad o \equiv g(\mathbf{rt}.1\&i) \xrightarrow{\textit{eval}} o \equiv g(i)$$

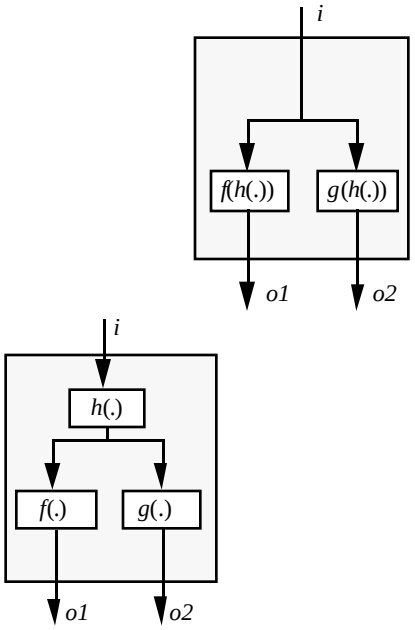
$$\xrightarrow{\textit{netreduct}} o \equiv g(i)$$



Figur 5.1: Gleichungselimination

b) Durch Expandieren und Falten lassen sich gemeinsame Teilausdrücke in neue Gleichungen "auslagern":

$o_1 \equiv f(h(i)), \quad \rightarrow \quad o_1 \equiv f(h(i)), \quad \rightarrow \quad o_1 \equiv f(s),$
$o_2 \equiv g(h(i)) \quad \textit{netexpand} \quad o_2 \equiv g(h(i)), \quad \textit{streamfold} \quad o_2 \equiv g(s),$
$s \equiv h(i) \qquad\qquad s \equiv h(i)$



Figur 5.2: Auslagern gemeinsamer Teilausdrücke

Beachte: Wenn *h* nichtdeterministisch ist, ist die expandierte Version eine (echte) Implementierung der ursrpünglichen.

c) Schließlich erlauben *netexpand*, *streamunfold* und *streamfold* das *Abwickeln* und umgekehrt *netreduct* und *streamfold* das *Zusammenfassen* von Rückkopplungsschleifen. Dabei ist jedoch zu beachten, daß das Abwickeln nur für det. Agenten erlaubt ist. Dies folgt aus der Anwendungsbedingung von *streamfold*. Seien also im folgenden *f* und *g* deterministische Agenten.

$s \equiv f(o), \quad \rightarrow \quad s \equiv f(g(s)), \quad \rightarrow$
$s \equiv f(g(f(g(s)))),$
$o \equiv g(s) \quad \textit{streamunfold} \quad o \equiv g(s) \quad \textit{streamunfold} \quad o \equiv g(s)$

$\rightarrow$ $s \equiv f(g(f(o))),$ $\rightarrow$ $s \equiv f(g(f(o))),$

streamfold $o \equiv g(s)$

netexpand $o \equiv g(s),$

$x \equiv f(o),$

$y \equiv g(f(o))$

$\rightarrow$ $s \equiv f(y),$

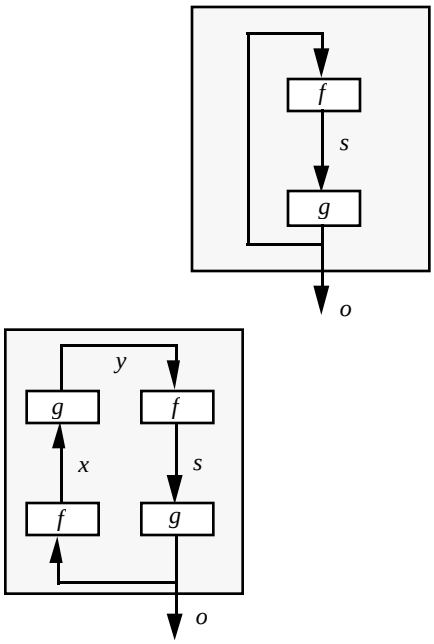
$\rightarrow$ $s \equiv f(y),$

streamfold $o \equiv g(s),$

streamfold $o \equiv g(s),$

$x \equiv f(o),$ $x \equiv f(o),$

$y \equiv g(f(o))$ $y \equiv g(x)$



Figur 5.3: Abwickeln von Rückkopplungsschleifen

Falls *f* und *g* nichtdeterministisch wären, so wäre die Semantik des ursprünglichen Netzes von der des abgewickelten verschieden: Die ursprüngliche Version wäre eine Implementierung der abgewickelten.

Im Zusammenhang mit dieser Anwendung sollte man sich die hinter Agentennetzen liegende Konzeption ins Gedächnis rufen. Jeder Agent repräsentiert eine eigenständige Einheit, die parallel zu den übrigen arbeitet und mit diesen kommuniziert. Durch das Abrollen einer Rückkopplungsschleife wird die Anzahl der parallel arbeitenden Instanzen vergrößert. Sofern genügend Rechenkapazität, sprich Prozessoren, vorhanden ist, kann das Abwickeln einen Effizienzgewinn ermöglichen. Allerdings muß man berücksichtigen, daß die zunehmende Parallelisierung auch einen erhöhten Kommunikationsaufwand bedingt. Da Kommunikation (zwischen verschiedenen Prozessoren) in der Regel teurer ist als interne Berechnungen (auf einem Prozessor), entsteht ein Konflikt, bei dessen Lösung man genau abwägen muß, welchen Nettoeffekt eine Parallelisierung hätte.

Offensichtlich kann eine Rückkopplungsschleife durch die Regeln beliebig weit abgewickelt werden (potentiell zu einem unendlichen Netz). Alle Darstellungen haben (im deterministischen Fall) dieselbe Semantik.

□

Das Falten und Auffalten von Stromgleichungen haben wir bereits behandelt. Die Idee, einen Bezeichner durch seine Definition ("unfold") und umgekehrt, eine Definition durch den zugehörigen Bezeichner ("fold") zu ersetzen, ist von Burstall und Darlington (vgl. [Burstall, Darlington 77]) als Transformationstechnik für applikative Sprachen eingeführt worden. Insbesondere ist das *Fold/Unfold-Schema* auch auf Funktionsaufrufe anwendbar. Bei der Formulierung der Regeln, sind Funktionsstriktheiten und die Handhabung von Nichtdeterminismus besonders zu beachten.

Transformation *non-recursive agentunfold*: Auffalten nicht-rekursiver Agentenaufrufe

agent $f \equiv$ **chan v** $s_0 \rightarrow$ **chan w** s_p:
$s_1 \equiv S_1, \ldots, s_j \equiv g(E,S), \ldots, s_n \equiv S_n$
end
agent $g \equiv$ **u** x, **chan v** $t_0 \rightarrow$ **chan w** t_q:
$t_1 \equiv T_1, \ldots, t_m \equiv T_m$
end

DET E, DEF E, DET S
\O(- NOTOCCURS $t_1, \ldots, t_{q-1}, t_{q+1}, \ldots, t_m$ IN **agent** $f \ldots$ **end**
————————;↓)

agent $f \equiv$ **chan v** $s_0 \rightarrow$ **chan w** s_p:
$s_1 \equiv S_1, \ldots, s_{j-1} \equiv S_{j-1}$,
$(t_1 \equiv T_1, \ldots, t_m \equiv T_m)[s_j/t_q, E/x, S/t_0]$,
$s_{j+1} \equiv S_{j+1}, \ldots, s_n \equiv S_n$,
end
agent $g \equiv$ **u** x, **chan v** $t_0 \rightarrow$ **chan w** t_q:
$t_1 \equiv T_1, \ldots, t_m \equiv T_m$
end

Korrektheitsbeweis: Sei $\delta \in$ ENV beliebig. Es ist zu zeigen:

$$\mathbf{F}_\delta[\![\textbf{agent } f \equiv \ldots : GS_2 \textbf{ end}, \textbf{agent } g \equiv \ldots : GS \textbf{ end}]\!] = FIX(\tau_2)$$
$$\subseteq \quad FIX(\tau_1) = \mathbf{F}_\delta[\![\textbf{agent } f \equiv \ldots : GS_1 \textbf{ end}, \textbf{agent } g \equiv \ldots : GS \textbf{ end}]\!]$$

Dabei bezeichne GS_1 das ursprüngliche Gleichungssystem im Rumpf von f und GS_2 das entfaltete. GS ist das Gleichungssystem im Rumpf von g. $FIX(\tau_1)$, $FIX(\tau_2)$ seien die größten Fixpunkte folgender Funktionale (i = 1,2):

$$\tau_i.(F,G) = (\text{strict } \mathbf{F}_{\delta[F/f,G/g]}[\![GS_i[s_0], s_p]\!], \text{strict } \mathbf{F}_{\delta[F/f,G/g]}[\![GS[x,t_0], t_q]\!])$$
(*)

Wir zeigen: $FIX(\tau_2) \subseteq \tau_1.FIX(\tau_2)$. Daraus folgt dann mit den Argumenten aus Satz 2.3) $FIX(\tau_2) \subseteq FIX(\tau_1)$. Sei also $FIX(\tau_2) = (F,G)$. Dann gilt für $\delta' = \delta[F/f,G/g]$ (Fixpunkt!):

$$(F,G) = (\text{strict } \mathbf{F}_{\delta'}[\![GS_2[s_0], s_p]\!], \text{strict } \mathbf{F}_{\delta'}[\![GS[x,t_0], t_q]\!])$$
(**)

Betrachte $\mathbf{F}_{\delta'}[\![GS_2[s_0],\ s_p]\!]$. Wegen DET E, DET S gilt der Gleichheitsfall aus Korollar 3.19 und deshalb folgt aus der Definition von **F**:

$$\mathbf{F}_{\delta'}[\![GS_2[s_0],\ s_p]\!] = \{\lambda s_0.\mathrm{pr}_p.\mathrm{fix}\begin{pmatrix}X\\Y\\Z\end{pmatrix} \mid f_i \in \mathbf{F}_{\delta'}[\![S_i]\!], h_i \in \mathbf{F}_{\delta'}[\![T_i]\!], f_S \in \mathbf{F}_{\delta'}[\![S]\!], f_E \in \mathbf{F}_{\delta'}[\![E]\!]\ \}.$$

Dabei ist pr_p die Projektion auf s_p und X, Y, Z sind wie folgt definiert:

$$X = \begin{pmatrix} s_1 = f_1(s_0, s_1, \ldots, s_n) \\ \ldots \\ s_{j-1} = f_{j-1}(s_0, s_1, \ldots, s_n) \end{pmatrix},$$

$$Y = \begin{pmatrix} t_1 = h_1(f_E(s_0,s_1,\ldots,s_n), f_S(s_0,s_1,\ldots,s_n), t_1, \ldots, t_{q-1}, s_j, t_{q+1}, \ldots, t_m) \\ \ldots \\ s_j = h_q(f_E(s_0,s_1,\ldots,s_n), f_S(s_0,s_1,\ldots,s_n), t_1, \ldots, t_{q-1}, s_j, t_{q+1}, \ldots, t_m) \\ \ldots \\ t_m = h_m(f_E(s_0,s_1,\ldots,s_n), f_S(s_0,s_1,\ldots,s_n), t_1, \ldots, t_{q-1}, s_j, t_{q+1}, \ldots, t_m) \end{pmatrix},$$

$$Z = \begin{pmatrix} s_{j+1} = f_{j+1}(s_0, s_1, \ldots, s_n) \\ \ldots \\ s_n = f_n(s_0, s_1, \ldots, s_n) \end{pmatrix}.$$

Analog gilt für $\mathbf{F}_{\delta'}[\![GS[x,t_0],\ t_q]\!]$:

$$\mathbf{F}_{\delta'}[\![GS[x,t_0],\ t_q]\!] = \{\lambda(x,t_0).\mathrm{pr}_q.\mathrm{fix}\ Y' \mid h_i \in \mathbf{F}_{\delta'}[\![T_i]\!]\},$$

wobei pr_q die Projektion auf t_q und Y' wie folgt definiert ist:

$$Y' = \begin{pmatrix} t_1 = h_1(x, t_0, t_1, \ldots, t_m) \\ \ldots \\ t_q = h_q(x, t_0, t_1, \ldots, t_m) \\ \ldots \\ t_m = h_m(x, t_0, t_1, \ldots, t_m) \end{pmatrix}.$$

Daraus folgt, daß zu jedem Y ein $g \in \mathbf{F}_{\delta'}[\![GS[x,t_0],\ t_q]\!]$ existiert mit:

$$\lambda s_0.\mathrm{pr}_p.\mathrm{fix}\begin{pmatrix}X\\Y\\Z\end{pmatrix} = \lambda s_0.\mathrm{pr}_p.\mathrm{fix}\begin{pmatrix}X\\s_j = g(f_E(s_0,s_1,\ldots,s_n), f_S(s_0,s_1,\ldots,s_n))\\Z\end{pmatrix}.$$

Wegen DEF E gilt sogar:

$$\lambda s_0.\mathrm{pr}_p.\mathrm{fix}\begin{pmatrix}X\\Y\\Z\end{pmatrix} = \lambda s_0.\mathrm{pr}_p.\mathrm{fix}\begin{pmatrix}X\\s_j = g'(f_E(\ldots), f_S(\ldots))\\Z\end{pmatrix},$$

wobei g' = strict g. Wegen (*) gilt: $g' \in G$. Daraus folgt:

$$\mathbf{F}_{\delta'}[\![GS_2[s_0], s_p]\!]$$

$\subseteq$ { gerade gezeigt }

$$\{ \lambda s_0.pr_p.fix\begin{pmatrix} X \\ s_j = g'(f_E(\ldots), f_S(\ldots)) \\ Z \end{pmatrix} \mid g' \in G, f_S \in \mathbf{F}_{\delta'}[\![S]\!], f_E \in \mathbf{F}_{\delta'}[\![E]\!] \}$$

$=$ { Definition von **F** }

$$\mathbf{F}_{\delta'}[\![GS_1[s_0], s_p]\!].$$

Durch einfaches Rechnen zeigt man nun:

$$\mathbf{F}_{\delta'}[\![GS_2[s_0], s_p]\!] \subseteq \mathbf{F}_{\delta'}[\![GS_1[s_0], s_p]\!]$$

$\Rightarrow$ { Definition von strict: strict A = {strict a | a$\in$ A} }

$$\text{strict } \mathbf{F}_{\delta'}[\![GS_2[s_0], s_p]\!] \subseteq \text{ strict } \mathbf{F}_{\delta'}[\![GS_1[s_0], s_p]\!]$$

$\Rightarrow$ { Definition von $\subseteq$ auf Tupeln }

$$(\text{ strict } \mathbf{F}_{\delta'}[\![GS_2[s_0], s_p]\!], \text{strict } \mathbf{F}_{\delta'}[\![GS[x,t_0], t_q]\!]) \subseteq (\text{ strict } \mathbf{F}_{\delta'}[\![GS_1[s_0], s_p]\!], \text{strict } \mathbf{F}_{\delta'}[\![GS[x,t_0], t_q]\!])$$

$\Rightarrow$ { (*) und (**) }

$(F,G) \subseteq \tau_1.(F,G)$.

□

AL-Agenten sind in ihren Objektparametern strikt, in ihren Stromparametern aber nicht. Deshalb die unterschiedliche Behandlung von E und S in obiger Regel. Faltet man einen Agentenaufruf auf, so ist es im allgemeinen nötig, eine einzelne Gleichung durch ein ganzes Gleichungssystem – den Rumpf des aufgerufenen Agenten – zu ersetzen. Eventuelle Namenskonflikte werden durch vorherige Umbenennung bereinigt. Besitzt der aufgerufene Agent keine Objektparameter auf, so kann in jedem Fall aufgefaltet werden. Probleme, die durch einen nichtdeterministischen Stromparameter S entstehen, lassen sich ausräumen, indem man eine neue Gleichung $s \equiv S$ einführt (Netzexpansion) und an der Aufrufstelle S durch *s* substituiert. *s* ist deterministisch. Symmetrisch zum Auffalten ist das Falten, d.h. der Ersatz eines (Teil-)Gleichungssystems durch einen Agentenaufruf.

Transformation *non-recursive agentfold***:** Falten nicht-rekursiver Agentenaufrufe

agent $f \equiv$ **chan v** $s_0 \rightarrow$ **chan w** s_p:
$s_1 \equiv S_1, \ldots, s_{j-1} \equiv S_{j-1},$
$(t_1 \equiv T_1, \ldots, t_m \equiv T_m)[s_j/t_q, E/x, S/t_0],$
$s_{j+1} \equiv S_{j+1}, \ldots, s_n \equiv S_n,$
end
agent $g \equiv$ **u** x, **chan v** $t_0 \rightarrow$ **chan w** t_q:
$t_1 \equiv T_1, \ldots, t_m \equiv T_m$
end

\O(- DEF E
————————;↓)

agent $f \equiv$ **chan v** $s_0 \rightarrow$ **chan w** s_p:
$s_1 \equiv S_1, \ldots, s_j \equiv g(E,S), \ldots, s_n \equiv S_n$
end
agent $g \equiv$ **u** x, **chan v** $t_0 \rightarrow$ **chan w** t_q:
$t_1 \equiv T_1, \ldots, t_m \equiv T_m$
end

Korrektheitsbeweis: Sei $\delta \in$ ENV beliebig. Symmetrisch zur "Unfold"-Regel ist zu zeigen:

$$\mathbf{F}_\delta[\![\textbf{agent } f \equiv \ldots : GS_1 \textbf{ end}, \textbf{agent } g \equiv \ldots : GS \textbf{ end}]\!] = FIX(\tau_1)$$
$$\subseteq \quad FIX(\tau_2) = \mathbf{F}_\delta[\![\textbf{agent } f \equiv \ldots : GS_2 \textbf{ end}, \textbf{agent } g \equiv \ldots : GS \textbf{ end}]\!]$$

GS_1, GS_2, GS, sowie τ_1, τ_2 seien dabei wie oben definiert. Wiederum zeigen wir: $FIX(\tau_1) \subseteq \tau_2.FIX(\tau_1)$. Sei $FIX(\tau_1) = (F,G)$. Dann gilt wie oben mit $\delta' = \delta[F/f, G/g]$:

$$(F,G) = (\text{ strict } \mathbf{F}_{\delta'}[\![GS_1[s_0], s_p]\!], \text{ strict } \mathbf{F}_{\delta'}[\![GS[x,t_0], t_q]\!])$$

und definitionsgemäß für $\mathbf{F}_{\delta'}[\![GS[x,t_0], t_q]\!]$:

$$\mathbf{F}_{\delta'}[\![GS[x,t_0], t_q]\!] = \{\lambda(x,t_0).pr_q.fix \begin{pmatrix} t_1 = h_1(x, t_0, t_1, \ldots, t_m) \\ \ldots \\ t_q = h_q(x, t_0, t_1, \ldots, t_m) \\ \ldots \\ t_m = h_m(x, t_0, t_1, \ldots, t_m) \end{pmatrix} | \, h_i \in \mathbf{F}_{\delta'}[\![T_i]\!]\}$$

und für $\mathbf{F}_{\delta'}[\![GS_1[s_0], s_p]\!]$:

$$\mathbf{F}_{\delta'}[\![GS_1[s_0],\ s_p]\!] = \{\lambda s_0.pr_p.fix\begin{pmatrix}X\\ s_j = g'(f_E(s_0,s_1,\ldots,s_n),\ f_S(s_0,s_1,\ \ldots,s_n))\\ Z\end{pmatrix}|$$

$f_i \in \mathbf{F}_{\delta'}[\![S_i]\!]$, $g' \in G$, $f_S \in \mathbf{F}_{\delta'}[\![S]\!]$, $f_E \in \mathbf{F}_{\delta'}[\![E]\!]\}$,

wobei wie oben:

$$X = \begin{pmatrix}s_1 = f_1(s_0,\ s_1,\ \ldots,\ s_n)\\ \ldots\\ s_{j-1} = f_{j-1}(s_0,\ s_1,\ \ldots,\ s_n)\end{pmatrix},$$

$$Z = \begin{pmatrix}s_{j+1} = f_{j+1}(s_0,\ s_1,\ \ldots,\ s_n)\\ \ldots\\ s_n = f_n(s_0,\ s_1,\ \ldots,\ s_n)\end{pmatrix}.$$

Zu jedem $g' \in G$ gibt es ein $g \in \mathbf{F}_{\delta'}[\![GS[x,t_0],\ t_q]\!]$ mit g' = strict g. Weiterhin existiert[1] zu jedem solchen $g \in \mathbf{F}_{\delta'}[\![GS[x,t_0],\ t_q]\!]$ ein Y:

$$Y = \begin{pmatrix}t_1 = h_1(f_E(s_0,s_1,\ldots,s_n),\ f_S(s_0,s_1,\ \ldots,s_n),\ t_1,\ \ldots,\ t_{q-1},\ s_j,\ t_{q+1},\ \ldots,\ t_m)\\ \ldots\\ s_j = h_q(f_E(s_0,s_1,\ldots,s_n),\ f_S(s_0,s_1,\ \ldots,s_n),\ t_1,\ \ldots,\ t_{q-1},\ s_j,\ t_{q+1},\ \ldots,\ t_m)\\ \ldots\\ t_m = h_m(f_E(s_0,s_1,\ldots,s_n),\ f_S(s_0,s_1,\ \ldots,s_n),\ t_1,\ \ldots,\ t_{q-1},\ s_j,\ t_{q+1},\ \ldots,\ t_m)\end{pmatrix}$$

$h_i \in \mathbf{F}_{\delta'}[\![T_i]\!]$, so daß

$$\lambda s_0.pr_p.fix\begin{pmatrix}X\\ s_j = g(f_E(\ldots),\ f_S(\ldots))\\ Z\end{pmatrix} = \lambda s_0.pr_p.fix\begin{pmatrix}X\\ Y\\ Z\end{pmatrix}.$$

Wegen DEF E gilt wieder:

$$\lambda s_0.pr_p.fix\begin{pmatrix}X\\ s_j = g'(f_E(\ldots),\ f_S(\ldots))\\ Z\end{pmatrix} = \lambda s_0.pr_p.fix\begin{pmatrix}X\\ Y\\ Z\end{pmatrix}.$$

Also folgt:

$$\mathbf{F}_{\delta'}[\![GS_1[s_0],\ s_p]\!]$$

$\subseteq$ { gerade gezeigt }

---

[1] Beachte: Beim Falten eines rekursiven Aufrufes muß dies nicht so sein. Deshalb ist das Falten rekursiver Aufrufe ***keine*** korrekte Transformationsregel!

$$\{\ \lambda s_0.pr_p.fix\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \mid\ f_i \in \mathbf{F}_{\delta'}[\![S_i]\!],\ h_i \in \mathbf{F}_{\delta'}[\![T_i]\!],\ f_S \in \mathbf{F}_{\delta'}[\![S]\!],\ f_E \in \mathbf{F}_{\delta'}[\![E]\!]$$

}

$\subseteq$ { Def. von **F** und Korollar 3.19. Weil DET E, DET S nicht gelten, gilt nicht = ! }

$\mathbf{F}_{\delta'}[\![GS_2[s_0],\ s_p]\!]$.

Analog zu oben ergibt sich aus $\mathbf{F}_{\delta'}[\![GS_1[s_0],\ s_p]\!] \subseteq \mathbf{F}_{\delta'}[\![GS_2[s_0],\ s_p]\!]$:

$$FIX(\tau_1) \subseteq \tau_2.FIX(\tau_1)$$

und daraus mit den Argumenten aus Satz 2.3: $FIX(\tau_1) \subseteq FIX(\tau_2)$.

□

Die beiden vorigen Regeln beziehen sich auf das Falten und Auffalten *nicht-rekursiver* Aufrufe. *Rekursive* Aufrufe müssen gesondert behandelt werden. Hier gilt, daß nur das Auffalten eine korrekte Transformationsregel darstellt, das Falten jedoch *nicht* (vgl. [Kott 80]).

Transformation *recursive agentunfold***:** Auffalten rekursiver Aufrufe

agent $f \equiv$ **u** x, **chan u** $s_0 \rightarrow$ **chan v** s_p:
$s_1 \equiv S_1, \ldots, s_j \equiv f(E,S), \ldots, s_n \equiv S_n$
end

DEF E, DET E, DET S,
\O(- NEW $t_1, \ldots, t_{p-1}, t_{p+1}, \ldots, t_n$
————————;↓)

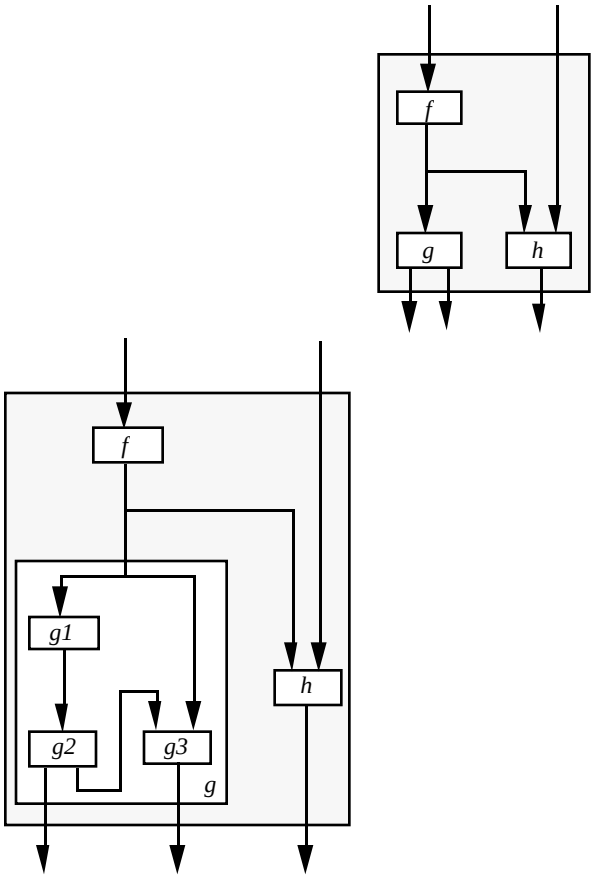
agent $f \equiv$ **u** x, **chan u** $s_0 \rightarrow$ **chan v** s_p:
$s_1 \equiv S_1, \ldots, s_{j-1} \equiv S_{j-1},$
$(s_1 \equiv S_1, \ldots, s_n \equiv S_n)[t_1/s_1, \ldots, s_j/s_p, \ldots, t_n/s_n, E/x, S/s_0]$
$s_{j+1} \equiv S_{j+1}, \ldots, s_n \equiv S_n,$
end

Korrektheitsbeweis: Wie oben.

□

Man macht sich leicht klar, daß die inverse Regelanwendung, d.h. das Falten rekursiver Aufrufe, zur Veränderung des Fixpunktes führen kann. Operationell gesprochen kann "Fold" zur Divergenz führen. Häufig will man trotzdem "Fold"-Schritte ausführen. Deren Korrektheit muß dann aber gesondert, d.h. von Fall zu Fall, nachgewiesen werden

Bezogen auf die Netzgestalt impliziert das Auffalten eines Aufrufes die Auflösung einer Hierarchiestufe. Die interne Struktur einer Systemkomponente, die anfangs verborgen war, wird wird nun sichtbar:



Figur 5.4: Auffalten von Agentenaufrufen

Invers dazu ist das Falten von Aufrufen. Auf diese Weise wird eine zusätzliche Hierarchiestufe eingeführt und Strukturinformation verborgen. Wenn g im obigen Beispiel rekursiv ist, z.B. $g \cong g_2$, dann kann das Netz offensichtlich beliebig häufig entfaltet werden. Es entsteht ein beliebig großes, potentiell unendliches Netz.

5.3 Übergang von AL nach PL

Mit Hilfe der Transformationsregeln des vorigen Abschnitts können AL-Programme manipuliert werden. Die Darstellung bleibt aber applikativ. Regeln, die den Übergang von applikativen zu prozeduralen Darstellungen tatsächlich leisten, werden in diesem Abschnitt vorgestellt. Dabei kann man auf unterschiedliche Weise vorgehen, entsprechend ist der Abschnitt weiter unterteilt:

Im ersten Unterabschnitt 5.3.1 wird die spezielle Klasse der *stromrepetitiven* Agenten untersucht, deren Vertreter schematisch in sequentielle PL-Agenten transformiert werden können. Wir geben hier einige Transformationsregeln, sowie einen *Metakalkül* an, mit dessen Hilfe Transformationsregeln für diese Agentenklasse hergeleitet werden können.

Der zweite Unterabschnitt 5.3.2 stützt sich auf die Tatsache, daß PL-Agentennetze als spezielle AL-Netze auffaßbar sind. Man kann daher einen AL-Agenten in prozedurale Form überführen, indem man die *Netztransformationen* aus Abschnitt 5.2 zielgerichtet anwendet. Die angegebene Regelmenge sehr allgemein, d.h. man kann mit den Regeln aus 5.3.2 (in Verbindung mit den Regeln aus 5.3.1) eine große Klasse von AL-Agenten in PL-Agenten transformieren. Allerdings wird dabei die rekursive Struktur der applikativen Ebene voll auf die prozedurale Ebene übertragen.

Der dritte Unterabschnitt 5.3.3 beschäftigt sich schließlich mit Agenten, die nicht in die in 5.3.1 untersuchte Klasse fallen, also nicht stromrepetitiv sind. Hier geht es vor allem darum, *Funktionsrekursion*, die zu unendlichen Netzen führt, durch *Stromrekursion*, d.h. durch rückgekoppelte Netze abzulösen. In gewissem Sinn soll dadurch der Nachteil der Regeln aus Abschnitt 5.3.2 behoben werden.

Die meisten der angeführten Transformationsregeln werden formal als korrekt bewiesen (eine Ausnahme bilden die Regeln des Metakalküls aus Abschnitt 5.3.1, deren Korrektheit durch informelle Argumentation plausibel gemacht wird). Da die Regeln den Übergang von AL nach PL behandeln, treten in ihnen Fragmente beider Sprachen auf. Die Korrektheitsbeweise stützen sich daher auf die Semantiken beider Sprachen. Hier zahlt sich deren Uniformität besonders aus.

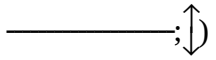
5.3.1 Transformation stromrepetitiver Agenten

Die Basis eines verteilten (hierarchischen) PL-Programms sind Agenten, deren Rumpf aus sequentiellen Anweisungen besteht. Sie heißen deshalb *sequentiell*. Jedes PL-Programm wird aus diesen Grundbausteinen zusammengesetzt. Wenn ein applikatives Programm in prozedurale Form überführt werden soll, so müssen notwendigerweise einige Programmteile, sprich AL-Agenten, auf diese sequentiellen Prozeduren abgebildet werden.
Es zeigt sich, daß eine bestimmte Klasse von AL-Agenten in enger Beziehung zu den sequentiellen PL-Agenten steht und schematisch transformiert werden kann. Man betrachte dazu folgende, sehr einfache Transformationsregel:

Transformation *recursion-to-iteration I***:**

agent $f \equiv$ **chan v** $i \rightarrow$ **chan w** o:
$o \equiv$ **if isempty.**i **then** ε **else** H[**ft.**i] & f(**rt.**i) **fi**
end

\O(-



agent $f \equiv$ **chan v** $i \rightarrow$ **chan w** o:
var v x;
while ¬**isclosed.**i **do**
$i?x$; o!H[x]
od;
close.o
end

Korrektheitsbeweis: Sei δ eine beliebige Umgebung. Dann ist zu zeigen:

$$\mathbf{F}_\delta[\![\textbf{agent } f \equiv \text{ << appl. >> } \textbf{end}]\!] = \mathbf{F}_\delta[\![\textbf{agent } f \equiv \text{ << proz. >> } \textbf{end}]\!].$$

Aus der Definition von **F** für applikative Agenten folgt, daß $\mathbf{F}_\delta[\![\textbf{agent } f \equiv \text{ << appl. >> } \textbf{end}]\!]$ die *größte* Menge $M \subseteq [\mathbf{V}^\omega \rightarrow \mathbf{W}^\omega]$ ist, für die gilt:

$$f \in M \quad \Leftrightarrow \quad \exists h \in \mathbf{F}_\delta[\![H[x]]\!],\ g \in M:\ f(\bot) = \bot \wedge f(\varepsilon) = \varepsilon \wedge$$

$$\forall v_0 \in \mathbf{V},\ v \in \mathbf{V}^\omega:\ f(v_0 \& v) = h(v_0) \& g(v).$$

Wir zeigen, daß $\mathbf{F}_\delta[\![\textbf{agent } f \equiv \text{ << proz. >> } \textbf{end}]\!]$ genauso charakterisiert ist.
Dazu betrachten wir die Semantik der Anweisung im Rumpf des prozeduralen *f*. Es gilt nach Definition von **S** für die einzelnen Bestandteile der Gesamtanweisung:

$$\mathbf{S}_\delta[\![i?x;\ o!F[x]]\!] = \{\ \lambda\sigma.\sigma[ft(\sigma(i))/x,\ rt(\sigma(i))/i,\ \sigma(o) \bullet h(ft(\sigma(i)))/o] \mid h \in \mathbf{F}_\delta[\![H[x]]\!]\ \}.$$

Beachte: Für $\sigma(i) = \bot$ oder $\sigma(i) = \varepsilon$ oder $h(ft(\sigma(i))) = \bot$ folgt: $\sigma[\ldots] = \sigma\downarrow$. Weiterhin:

$$\mathbf{S}_\delta[\![\textbf{while } \neg\textbf{isclosed.}i \textbf{ do } i?x;\ o!H[x] \textbf{ od};]\!]$$

ist die *größte* Menge $N \subseteq$ **STATE-TRANS**, für die gilt:

$$a \in N \quad \Leftrightarrow \quad \exists h \in \mathbf{F}_\delta[\![H[x]]\!],\ a' \in N:\ \forall \sigma \in \text{STATE}':\ \forall v_0 \in \mathbf{V},\ v \in \mathbf{V}^\omega:$$

$$a(\sigma) = \backslash B \backslash LC \backslash \{(\backslash A \backslash AL(\sigma\downarrow$$
falls $\sigma(i) = \bot;\sigma$
falls $\sigma(i) = \varepsilon;a'(\sigma[v_0/x,\ v/i,$
$\sigma(o) \bullet h(v_0)/o])$ falls $\sigma(i) = v_0 \& v)).$

Daraus folgt:

$\mathbf{S}_\delta[\![\mathbf{var\ v}\ x; \mathbf{while}\ \ldots\ \mathbf{od}; \mathbf{close.}o]\!]$

ist die *größte* Menge N', für die gilt:

$$a \in N' \Leftrightarrow \exists h \in \mathbf{F}_\delta[\![H[x]]\!],\ b \in N' : \forall \sigma \in \mathrm{STATE}' :\ \forall v_0 \in \mathbf{V},\ v \in \mathbf{V}^\omega : \tag{1}$$

$$a(\sigma) = \backslash B \backslash LC \backslash \{(\backslash A \backslash AL(\sigma\downarrow \text{ falls } \sigma(i) = \perp; \sigma[\sigma(o) \bullet @] \text{ falls } \sigma(i) = \varepsilon; b(\sigma[v_0/x,\ v/i,\ \sigma(o) \bullet h(v_0)/o]) \text{ falls } \sigma(i) = v_0 \& v)).$$

Sei nun $\mathbf{F}_\delta[\![\mathbf{agent}\ f \equiv\ <<\ \text{proz.}\ >>\ \mathbf{end}]\!] = M'$. Dann gilt definitionsgemäß:

$f \in M'$

$\Leftrightarrow$ { Definition von $\mathbf{F}_\delta[\![\mathbf{agent}\ f \equiv\ <<\ \text{proz}\ >>\ \mathbf{end}]\!]$ }

$$\exists a \in \mathbf{S}_\delta[\![\mathbf{var\ u}\ x\ ;\ \ldots\ ;\ \mathbf{close.}o]\!] : \forall v \in \mathbf{V}^\omega : f(v) = \mathrm{cast}(\ a(\sigma_0[v/i])(o)\ ) \tag{2}$$

$\Leftrightarrow$ { Definition von N', Def. von $\sigma[\ldots]$ und **STATE-TRANS** }

$$\exists h \in \mathbf{F}_\delta[\![H[x]]\!],\ b \in N' : \forall v_0 \in \mathbf{V},\ v \in \mathbf{V}^\omega : \tag{3}$$

$$f(\perp) = \mathrm{cast}(\ a(\sigma_0[\perp/i])(o)\ ) = \mathrm{cast}(\ \sigma_0\downarrow(o)\ ) = \mathrm{cast}(\varepsilon \bullet \perp) = \perp \wedge$$

$$f(\varepsilon) = \mathrm{cast}(\ a(\sigma_0[\varepsilon/i])(o)\ ) = \mathrm{cast}(\ \sigma_0[\sigma_0(o) \bullet @/o](o)\ ) = \mathrm{cast}(\varepsilon \bullet @) = \varepsilon \wedge$$

$$f(v_0 \& v) = \mathrm{cast}(\ a(\sigma_0[v_0 \& v/i])(o)\ ) = \mathrm{cast}(\ b(\sigma_0[v_0/x,\ v/i,\ \sigma_0(o) \bullet h(v_0)/o])(o)\ ) = \mathrm{cast}(\ b(\sigma_0[v_0/x,\ v/i,\ \varepsilon \bullet h(v_0)/o])(o)\ ) = h(v_0)\ \&\ \mathrm{cast}(\ b(\sigma_0[v/i])(o)\ )$$

$\Leftrightarrow$ { wegen (2) und (3) }

$$\exists h \in \mathbf{F}_\delta[\![H[x]]\!],\ g \in M' : \forall v_0 \in \mathbf{V},\ v \in \mathbf{V}^\omega : f(\perp) = \perp \wedge f(\varepsilon) = \varepsilon \wedge f(v_0 \& v) = h(v_0)\ \&\ g(v).$$

Weil N' die größte Menge ist, die (1) erfüllt, ist M' die größte Menge, die die jetzt hergeleitete Eigenschaft erfüllt. Also gilt: M = M'

□

Man beachte, daß in diesem Beweis sowohl die Semantik von AL als auch die Semantik von PL verwendet wird.
Der Agent *f* definiert eine "Map"-Funktion (im Sinne von Bird, vgl. [Bird 89]). Er wendet eine Basisfunktion, repräsentiert durch den Ausdruck H, punktweise auf die Elemente seiner Eingabe an. Es lassen sich viele Variationen dieser Regel angeben, die sich von der obigen Version an folgenden Punkten unterscheiden:

- Andere Reaktion im "nicht-rekursiven" Zweig, z.B. Ausgabe weiterer Nachrichten statt sofortiger Terminierung.
- Ausgabe mehrerer Nachrichten als Reaktion auf eine Eingabe oder anders herum, Ausgabe einer Nachricht erst bei mehreren Eingaben.
- Behandlung mehrerer Eingabekanäle, sowie zusätzlicher Objektparameter.

Explizit soll hier nur ein weiteres Schema angegeben werden, daß sich insbesondere auf den letzten Punkt bezieht.

Transformation *recursion-to-iteration II*: Behandlung von Objektparametern

agent $f \equiv$ **u** p, **chan v** $i \rightarrow$ **chan w** o:
$o \equiv$ **if** B[p, **ft.**i] **then** G[p, **ft.**i] & ε
else H[p, **ft.**i] & f(K[p, **ft.**i], **rt.**i) **fi**
end

\O(- OCCURS **ft.**i IN B
——————;⇕)

agent $f \equiv$ **u** p, **chan v** $i \rightarrow$ **chan w** o:
var u $x := p$; **var v** y;
$i?y$;
while ¬B[x, y] **do**
o!H[x, y]; x := K[x, y]; $i?y$
od;
o!G[x, y]; **close.**o
end

Korrektheitsbeweis: Der Beweis verläuft genau wie oben, ist aber technisch etwas aufwendiger. Sei δ eine beliebige Umgebung. Dann gilt:

$\mathbf{F}_\delta$⟦**agent** $f \equiv$ << appl. >> **end**⟧

ist die *größte* Menge $M \subseteq [\mathbf{U}^\perp \times \mathbf{V}^\omega \rightarrow \mathbf{W}^\omega]$, für die gilt:

$f \in M \Leftrightarrow \exists f' \in M$:
$\exists b \in \mathbf{F}_\delta$⟦B[$x$,$y$]⟧, $h \in \mathbf{F}_\delta$⟦H[x,y]⟧, $k \in \mathbf{F}_\delta$⟦K[x,y]⟧, $g \in \mathbf{F}_\delta$⟦G[x,y]⟧:
$\forall u \in \mathbf{U}$, $v_0 \in \mathbf{V}$, $v \in \mathbf{V}^\omega$:
$f(\perp,v) = f(u,\perp) = f(u,\varepsilon) = \perp \wedge$
$f(u,v_0\&v) =$ \B\LC\{(\A\AL(⊥
falls $b(u,v_0) = \perp$;‹$g(u,v_0)$›
falls $b(u,v_0) =$ true ;$h(u,v_0)$ & $f'(k(u,v_0),v)$ falls $b(u,v_0) =$ false)).

Weiterhin gilt für die Schleife im Rumpf des prozeduralen f:

$\mathbf{S}_\delta$⟦**while** ¬B[x, y] **do** o!H[x, y]; x := K[x, y]; $i?y$ **od**⟧

ist die *größte* Menge N $\subseteq$ **STATE-TRANS**, für die gilt:

$$a \in N \Leftrightarrow \exists b \in \mathbf{F}_\delta[\![B[x,y]]\!],\ h \in \mathbf{F}_\delta[\![H[x,y]]\!],\ k \in \mathbf{F}_\delta[\![K[x,y]]\!],\ a' \in N: \quad (1)$$

$$\forall \sigma \in \mathrm{STATE}': a(\sigma) = \backslash B \backslash LC \backslash \{(\backslash A \backslash AL(\sigma\downarrow$$

falls $b(\sigma(x),\sigma(y)) = \perp; \sigma$ falls $b(\sigma(x),\sigma(y)) = \mathrm{true}; a'(\sigma')$

falls $b(\sigma(x),\sigma(y)) = \mathrm{false})),$

wobei $\sigma' = \sigma[\sigma(o) \bullet h(\sigma(x),\sigma(y))/o,\ k(\sigma(x),\sigma(y))/x,\ \mathrm{ft}(\sigma(i))/y,\ \mathrm{rt}(\sigma(i))/i]$. Daraus folgt:

$$\mathbf{S}_\delta[\![\textbf{var u } x := p; \ldots ; \textbf{close.} o]\!]$$

$$= \{ a_3 \circ a_2 \circ a_1 \mid a_1 \in \mathbf{S}_\delta[\![\textbf{var u } x := p; \textbf{ var v } y; \ i?y;]\!], \ a_2 \in N, \ a_3 \in \mathbf{S}_\delta[\![o!G[x, y]; \textbf{ close.} o]\!] \}$$

$$= \{ a_3 \circ a_2 \circ a_1 \mid a_1 = \lambda\sigma.\sigma[\sigma(p)/x, \mathrm{ft}(\sigma(i))/y, \mathrm{rt}(\sigma(i))/i], \ a_2 \in N, \ a_3 = \lambda\sigma.\sigma[\sigma(o) \bullet g(\sigma(x),\sigma(y)) \bullet @/o], \ g \in \mathbf{F}_\delta[\![G[x,y]]\!] \}.$$

Sei nun $\mathbf{F}_\delta[\![\textbf{agent } f \equiv$ << proz. >> **end**$]\!] = M'$. Dann gilt für M':

$f \in M'$

$\Leftrightarrow$ { Definition von $\mathbf{F}_\delta[\![\textbf{agent } f \equiv$ << prozedural >> **end**$]\!]$ }

(2)

$\exists a \in \mathbf{S}_\delta[\![\textbf{var u } x := p; \ldots ; \textbf{close.} o]\!]:$

$\forall u \in \mathbf{U}^\perp,\ v \in \mathbf{V}^\omega: f(u,v) = \mathrm{cast}(\ a(\sigma_0[u/x, v/i])(o)\ )$

$\Leftrightarrow$ { Definition von $\mathbf{S}_\delta[\![\textbf{var u } x := p; \ldots ; \textbf{close.} o]\!]$ }

(3)

$\exists a_2 \in N,\ a_1,\ a_3$ wie oben definiert:

$\forall u \in \mathbf{U}^\perp,\ v \in \mathbf{V}^\omega: f(u,v) = \mathrm{cast}(\ a_3(a_2(a_1(\sigma_0[u/x, v/i])))(o)\ )$

$\Leftrightarrow$ { Def. von N, Def. von a_1 und a_3 oben, Def. von $\sigma[\ldots]$ und **STATE-TRANS** }

$\exists a_2, a_2' \in N,\ a_1,\ a_3$ wie oben definiert:

$\exists b \in \mathbf{F}_\delta[\![B[x,y]]\!],\ h \in \mathbf{F}_\delta[\![H[x,y]]\!],\ k \in \mathbf{F}_\delta[\![K[x,y]]\!],\ g \in \mathbf{F}_\delta[\![G[x,y]]\!]:$

$\forall u \in \mathbf{U},\ v_0 \in \mathbf{V},\ v \in \mathbf{V}^\omega:$

$$f(u,\perp) = \mathrm{cast}(\ a_3(a_2(a_1(\sigma_0[u/x, \perp/i])))(o)\ )$$

$= \text{cast}(\ a_3(a_2(\sigma_0[u/x, \bot/y, \bot/i]))(o)\ )$

$= \text{cast}(\ \sigma_0{\downarrow}(o)\ ) =$

$\text{cast}(\varepsilon\bullet\bot) = \bot \qquad \wedge$

$f(u,\varepsilon)$ = *analog zum vorherigen Fall* $= \bot$

$\wedge$

$f(u,v_0\&v) = \text{cast}(\ a_3(a_2(a_1(\sigma_0[u/x, v_0\&v/i])))(o)\ )$

$= \text{cast}(\ a_3(a_2(\sigma_0[u/x, v_0/y, v/i]))(o)\ )$

$= \B\LC\{(\A\AL(\text{cast}(\ \sigma_1(o)\ )$ falls $b(u,v_0) = \bot;\text{cast}(\ \sigma_2(o)\ )$ falls $b(u,v_0) = \text{true};\text{cast}(\ a_3(a_2'(\sigma_3(o)))\ )$ falls $b(u,v_0) = \text{false})),$

wobei $\sigma_1 = \sigma_0{\downarrow}$,

$\sigma_2 = \sigma_0[u/x,\ v_0/y,\ v/i,\ \langle g(u,v_0)\rangle\bullet@/o]$,

$\sigma_3 = \sigma_0[\langle h(u,v_0)\rangle/o,\ k(u,v_0)/x,\ ft(v)/y,\ rt(v)/i\,]$.

$= \B\LC\{(\A\AL(\bot$ falls $b(u,v_0) = \bot;\langle g(u,v_0)\rangle$ falls $b(u,v_0) = \text{true} ;h(u,v_0)\&\text{cast}(\ a_3(a_2'(\sigma_4(o)))\ )$ falls $b(u,v_0) = \text{false})),$

wobei $\sigma_4 = \sigma_0[k(u,v_0)/x,\ ft(v)/y,\ rt(v)/i\,]$.

$= \B\LC\{(\A\AL(\bot$ falls $b(u,v_0) = \bot;\langle g(u,v_0)\rangle$ falls $b(u,v_0) = \text{true} ;h(u,v_0)\&\text{cast}(\ a_3(a_2'(a_1(\sigma_5)))(o)\ )$ falls $b(u,v_0) = \text{false})),$

wobei $\sigma_5 = \sigma_0[k(u,v_0)/x,\ v/i]$.

$\Leftrightarrow$ { wegen (2) und (3) }

$\exists f'\in M'$:

$\exists b\in \mathbf{F}_\delta[\![B[x,y]]\!],\ h\in \mathbf{F}_\delta[\![H[x,y]]\!],\ k\in \mathbf{F}_\delta[\![K[x,y]]\!],\ g\in \mathbf{F}_\delta[\![G[x,y]]\!]$:

$\forall u\in \mathbf{U},\ v_0\in \mathbf{V},\ v\in \mathbf{V}^\omega$:

$$f(\perp,v) = f(u,\perp) = f(u,\varepsilon) = \perp \wedge$$

$$f(u,v_0\&v) = \B\LC\{(\A\AL(\perp$$

falls $b(u,v_0) = \perp$;‹$g(u,v_0)$›

falls $b(u,v_0)$ = true ;$h(u,v_0)\&f'(k(u,v_0),v)$ falls $b(u,v_0)$ = false))

Weil N die größte Menge ist, für die Eigenschaft (1) gilt, ist M' die größte Menge, für die die jetzt abgeleitete rekursive Charakterisierung gilt. Also: M' = M

□

Beim Übergang zur prozeduralen Version wird die Rolle des Objektparameters für die Modellierung *interner Zustände* besonders deutlich: Jeder Objektparameter der applikativen Ebene entspricht einer lokalen Variable, die mit dem Aufrufwert vorbesetzt und dann vor jedem Schleifendurchlauf geeignet aktualisiert wird. Durch Einbettung kann ein fester Startzustand festgelegt werden. Eingebettete Agenten lassen sich mit (einer Variante) der folgenden Regel transformieren:

Transformation *recursion-to-iteration III***:** Behandlung eingebetteter Agenten

agent f'**≡ chan v** i **→ chan w** o:
$o \equiv f(\mathrm{E}, i)$
end
agent f**≡ u** u**, chan v** i **→ chan w** o:
var u x:= u; V;
A
end

\O(- E *ist ein reiner Objektausdruck*

—————;⇕)

agent f'**≡ chan v** i **→ chan w** o:
var u x : = E; V;
A
end
agent f**≡ u** u**, chan v** i **→ chan w** o:
var u x:= u; V;
A
end

Korrektheitsbeweis: Sei δ eine beliebige Umgebung. Es ist zu zeigen:

$\mathbf{F}_\delta[\![$**agent** $f'\equiv$ << appl. >> **end**, **agent** $f\equiv$ … **end** $]\!]$ = (M',M)
=
$\mathbf{F}_\delta[\![$**agent** $f'\equiv$ << proz. >> **end**, **agent** $f\equiv$ … **end** $]\!]$ = (N',N)

Weil f ein sequentieller Agent ist, kommt f' im Rumpf von f nicht vor. Deshalb gilt:

M = N = $\mathbf{F}_\delta[\![$**agent** $f\equiv$ … **end** $]\!]$

und

M' = $\mathbf{F}_{\delta[M/f]}[\![$**agent** $f'\equiv$ << appl. >> **end** $]\!]$
N' = $\mathbf{F}_{\delta[M/f]}[\![$**agent** $f'\equiv$ << proz. >> **end** $]\!]$

Es bleibt zu zeigen: M' = N'. Nach Definition gilt für M'

M'
= { Definition von $\mathbf{F}_{\delta[M/f]}[\![$**agent** $f'\equiv$ << appl. >> **end** $]\!]$ }
$\mathbf{F}_{\delta[M/f]}[\![f(\mathrm{E}, i)]\!]$
= { Definition von $\mathbf{F}_{\delta[M/f]}[\![f(\mathrm{E}, i)]\!]$ }

$$\{ \lambda v.f(u,v) \mid f \in M, u \in \mathbf{F}_{\delta[M/f]}[\![E]\!] \}$$

$=$ { Anwendungsbedingung: f kommt in E nicht vor }

$$\{ \lambda v.f(u,v) \mid f \in M, u \in \mathbf{F}_{\delta}[\![E]\!] \}$$

Für N' gilt:

$$N'$$

$=$ { Definition von $\mathbf{F}_{\delta[M/f]}[\![$**agent** $f' \equiv$ << proz.>> **end**$]\!]$ }

$$\{ f' \mid \exists f_A \in \mathbf{S}_{\delta[M/f]}[\![\textbf{var u } x := E; V; A]\!]: \forall v \in \mathbf{V}^{\omega}: f'(v) = cast(f_A(\sigma_0[v/i])(o)) \}$$

$=$ { Anwendungsbedingung: f kommt im Rumpf des prozeduralen f' nicht vor }

$$\{ f' \mid \exists f_A \in \mathbf{S}_{\delta}[\![\textbf{var u } x := E; V; A]\!]: \forall v \in \mathbf{V}^{\omega}: f'(v) = cast(f_A(\sigma_0[v/i])(o)) \}$$

$=$ { Definition von $\mathbf{S}_{\delta}[\![$**var u** $x := E; V; A]\!]$ }

$$\{ f' \mid \exists u \in \mathbf{F}_{\delta}[\![E]\!]: \exists g_A \in \mathbf{S}_{\delta}[\![V; A]\!]: \forall v \in \mathbf{V}^{\omega}: f'(v) = cast(g_A(\sigma_0[u/x, v/i])(o)) \}$$

$=$ { Definition von $\mathbf{F}_{\delta}[\![$**agent** $f \equiv \ldots$ **end**$]\!] = N = M$ }

$$\{ f' \mid \exists u \in \mathbf{F}_{\delta}[\![E]\!]: \exists f \in M: \forall v \in \mathbf{V}^{\omega}: f'(v) = f(u,v) \}$$

$=$ { klar }

$$\{ \lambda v.f(u,v) \mid f \in M, u \in \mathbf{F}_{\delta}[\![E]\!] \}$$

Also M' = N'.

□

Die Klasse der AL-Agenten, auf die sich derartige Regeln anwenden lassen, ist durch den auftretenden Rekursionstyp (und die Verwendung der applikativen Zugriffsoperationen auf Ströme **ft**, **rt** und **isempty**) charakterisiert. Eine Unterscheidung zwischen verschiedenen Rekursionsformen – *repetitiv*, *linear-rekursiv*, *kaskadenartig*, usw. – ist aus der sequentiellen Programmierung und insbesondere aus der Theorie der rekursiven Programmschemata wohlbekannt (vgl. [Manna 74], [Bauer, Wössner 81]). Auch dort sind abgestimmte Transformationsregeln entwickelt worden (vgl. [Manna 74], Kap. 4, [Bauer, Wössner 81], Kap. 4, sowie [Walker, Strong 73]). Der Übergang von funktionalen zu prozeduralen Darstellungen war für repetitive (engl. *"tail recursive"*) Rechenvorschriften besonders einfach. Es ist daher nicht verwunderlich, wenn der Rekursionstyp der hier betrachteten Agentenklasse dem sequentiellen repetitiven Schema ähnelt:

Ein AL-Agent

agent $f \equiv$ **chan v** $i \rightarrow$ **chan w** o:
    $o \equiv E$
end

heißt *stromrepetitiv*, wenn gilt:

- In E kommen nur rekursive Aufrufe von *f* und sonst keine anderen Aufrufe stromverarbeitender Agenten vor.
- Wenn E einen Teilausdruck der Art **ft.**E', **rt.**E', **isempty.**E' oder *f*(E') enthält, dann hat E' die Gestalt $\mathbf{rt}^n\mathbf{.}i$. Darüber hinaus kommt der Ausgabestrom *o* nicht in E vor.

Dabei ist $\mathbf{rt}^n\mathbf{.}i$ ($n \geq 0$) eine abkürzende Schreibweise für **rt.rt.** … **rt.***i* (n-mal). Entsprechend steht $\mathbf{rt}^0\mathbf{.}i$ für *i*. Diese Schreibweise entspricht der Funktionsiteration auf der semantischen Ebene (vgl. Satz 2.2).

Beim sequentiellen repetitiven Schema findet sich der rekursive Aufruf der definierten Rechenvorschrift aufschreibungstechnisch ganz "außen". Operationell (z.B. bei Abarbeitung auf der Kellermaschine) wird er als letzte Aktion vor dem Abschluß der Auswertung (Rücksprung), ausgeführt. Rein formal gilt dies für stromrepetitive Agenten nicht:

Man betrachte das applikative *f* aus der Regel *recursion-to-iteration I* auf Seite 106. *f* ist offensichtlich stromrepetitiv. Der rekursive Aufruf ist hier jedoch in einen Konstruktorausdruck eingebunden – F[**ft.***i*] & *f*(**rt.***i*) oder deutlicher in Präfixschreibweise &(F[**ft.***i*], *f*(**rt.***i*)) –, steht also nicht "ganz außen". Folgt daraus, daß "hängende Operationen" entstehen, die nach Abarbeitung des rekursiven Aufrufs erledigt werden müssen? Nein, das ist nicht der Fall!
Der Stromkonstruktor & ist in seinem zweiten Element nicht-strikt. Die Konstruktionsoperation & kann daher vorgezogen und der rekursive Aufruf verzögert werden (*lazy evaluation*). Hängende Operationen werden so vermieden. Diese Form des repetitiven Schemas tritt in Sprachen mit verzögerter Auswertung häufig auf. Der Begriff *"tail recursive modulo cons"* ist dafür gebräuchlich.

Die erste Bedingung der obigen Definition erlaubt es, stromrepetitive Agenten auf sequentielle PL-Agenten (**while**-Prozeduren) abzubilden. Die zweite Bedingung erzwingt einen disziplinierten Umgang mit den Ein- und Ausgabeströmen, der mit dem Kanalkonzept von PL verträglich ist. Im wesentlichen besagt sie, daß auch AL-Agenten nur lesend auf ihre Eingabe- und schreibend auf ihre Ausgabeströme zugreifen dürfen. Der Agent

agent $f \equiv$ **chan v** $i \rightarrow$ **chan w** o:
$o \equiv$ F[**ft.***i*] & *f*(**ft.rt.***i* & **ft.***i* & $\mathbf{rt}^2\mathbf{.}i$)
end

ist aus diesem Grund nicht stromrepetitiv. Man kann den rekursiven Aufruf als "schreibenden" Zugriff auf *i* deuten: Von *i* wird kein Element entfernt, vielmehr werden die ersten beiden Elemente vertauscht.

Es ist kennzeichnend für die PL-Agenten in den Transformationsregeln *recursion-to-iteration I, II*, daß bei jedem Schleifendurchlauf genau ein Element von der Eingabe gelesen und verarbeitet wird. Daher reicht eine Variable für die Zwischenspeicherung aus. Im Allgemeinen werden mehrere Werte benötigt, die darüber hinaus während mehrerer Schleifendurchläufe zur Verfügung stehen müssen. Das macht geeignete Speicherstrukturen notwendig. Wir benutzen dazu im folgenden eine Variable vom Typ **queue** mit den zugehörigen Operationen "eq", "head", "tail", "stock" usw. (vgl. Beispiel 3.3 in Abschnitt 3.1).

Anschließend soll nun ein Ansatz vorgestellt werden, mit dessen Hilfe jeder stromrepetitive AL-Agent in einen sequentiellen PL-Agenten umgeformt werden kann. Dadurch wird eine Art *Metakalkül* für Transformationsregeln definiert: Zu jedem stromrepetitiven AL-Agenten kann mit seiner Hilfe ein äquivalenter PL-Agent, mithin eine korrekte Transformationsregel, abgeleitet werden. Die abgeleiteten Transformationsregeln haben folgendes Aussehen

agent $f \equiv$ **chan v** $i \rightarrow$ **chan w** o:
    $o \equiv$ E
end

\O(-
———————;⇕)

agent $f \equiv$ **chan v** $i \rightarrow$ **chan w** o:
    var bool *stop* := false;
    var v x; **var queue v** q := eq;
    var int *ap, rd* := 0, 0;
    while ¬*stop* **do**
        ‹Statement›
    od;
    close.o
end

Dabei ist ‹Statement› diejenige (eindeutig bestimmte) Anweisung, die mit den Regeln des Kalküls aus der Startkonfiguration **skip** ›› $o \equiv$ E abgeleitet wird. Allgemein ist eine *Konfiguration* K entweder eine PL-Anweisung oder K hat die Gestalt

A ›› $o \equiv$ E bzw. cn[A ›› $o \equiv$ E]

wobei A∈ ‹stat› eine PL-Anweisung, cn[.] ein Anweisungskontext und E∈ ‹exp› ein AL-Ausdruck ist, der die syntaktischen Nebenbedingungen für stromrepetitive Agenten erfüllt.
Der Kalkül besteht aus einer Menge von Regeln, durch die Konfigurationen in einander überführt werden. Die Idee liegt darin, die applikative Definition für o, d.h. den Gleichungsteil aus A ›› $o \equiv$ E, schrittweise in den Anweisungsteil zu verlagern, bis die Gleichung schließlich ganz verschwindet. In den Konfigurationen treten die Variablen und Bezeichner auf, die im obigen Schema vereinbart wurden. Sie haben folgende Bedeutung:

- *stop* steuert den Abbruch der Schleife, d.h. die Termination des Agenten.
- q dient dazu, die Werte zwischenzuspeichern, die von der Eingabe i gelesen werden, um für die nächsten Berechnungsschritte verfügbar zu sein. Zu Beginn ist q leer (Initialisierung mit "eq")

- *ap* (für *append*) und *rd* (für *reduce*) steuern das Wachsen und Schrumpfen der Speicherschlange. In jedem Schleifendurchlauf müssen (in der Regel) neue Werte von *i* gelesen werden (mglw. in Abhängigkeit von früheren Werten). Sie werden ans Ende von *q* angefügt. Analog können nicht mehr benötigte Werte aus der Schlange entfernt werden und zwar in FIFO-Manier am Anfang von *q*. Zugriffe auf die Eingabe sind nur notwendig, wenn Bedingungen überprüft und/oder Ausgaben erzeugt werden.

Die Anweisung im Rumpf der Schleife **while** ¬*stop* **do** ‹Statement› **od** beschreibt in prozeduraler Form, die Wirkung auf den Ausgabekanal *o*, die sich auf der applikativen Ebene (operationell betrachtet) bei Abarbeitung eines (rekursiven) Aufrufes von *f* ergibt. Jeder Schleifendurchlauf entspricht einem (rekursiven) Aufruf auf der applikativen Ebene. Eine Konfiguration A ›› *o* ≡ E beschreibt die Wirkung auf *o* (und *i*) teilweise prozedural (durch A), teilweise applikativ (durch *o* ≡ E). Die Pfeilspitzen ›› können als "und dann" gelesen werden. Das heißt man erhält die Gesamtwirkung, indem man erst die Wirkung der Anweisung "und dann" die Wirkung der Gleichung berücksichtigt.

Auf Grundlage dieser Vorstellung kann man sich die *Korrektheit* der nachfolgenden Regeln informell klar machen:

Eine (Meta-)Regel $K_1 \rightarrow K_2$ ist *korrekt*, wenn die Wirkung der Konfiguration K_1 auf den Ausgabestrom/kanal *o*, gleich der Wirkung der Konfiguration K_2 auf *o* ist.
Zusammen mit den einzelnen Regeln werden jeweils die entsprechenden Korrektheitsargumente skizziert. Auf einen *formalen* Korrektheitsbeweis für den Kalkül soll aber aus Platzgründen verzichtet werden. Um ihn durchzuführen, müßte eine Semantik für die Konfigurationen entwickelt werden (was technisch sicher aufwendig wäre), die mit den Funktionsmengensemantiken für AL und PL verträglich wäre. Es wäre dann zu zeigen, das die Regeln des Kalküls bezüglich dieser Semantik Äquivalenzumformungen darstellen.

Die ersten drei Regeln sind simpel: Wenn die Anweisung A den Ausgabestrom o auf *o* erzeugt, dann repräsentiert o auch die Gesamtwirkung der Konfiguration A ›› *o* ≡ ε. Es ist leicht einzusehen, daß die Anweisung A; *stop* := true die gleiche Wirkung hat: Wenn A den Ausgabestrom o erzeugt hat, wird die äußere **while**-Schleife wegen *stop* := true beendet und der Kanal durch die nachfolgende Anweisung **close.***o* geschlossen (vgl. obiges Schema). Gemäß PL-Semantik führt dies zum Versenden der Endemarke @, die durch Übergang zur funktionalen Beschreibung (siehe die Funktion "cast" in Abschnitt 4.2.2) wieder "gelöscht" wird. Die Wirkung von A; *stop* := true ist daher ebenfalls o. Die erste Regel des Metakalküls lautet:

1. A ›› *o* ≡ ε → A; *stop* := true.

Wenn A die Ausgabe o erzeugt, dann repräsentiert o^⊥ die Wirkung der Konfiguration A ›› *o* ≡ ⊥. Die Anweisung A; **abort** hat die gleiche Wirkung: Wegen **abort** "beendet" der Agent seine Arbeit, ohne weitere Schritte auszuführen, insbesondere ohne die Ausgabe abzuschliessen.
Gemäß PL-Semantik führt dies zum Anhängen von ⊥ an den bisher auf *o* erzeugten Ausgabestrom o (siehe wiederum die Funktion "cast" in Abschnitt 4.2.2). Die Regel lautet:

2. A ›› *o* ≡ ⊥ → A; **abort**.

Wenn A die Ausgabe o erzeugt, dann repräsentiert o^i die Wirkung der Konfiguration $A \rightarrow\!\rightarrow o \equiv i$, wobei i der Strom auf *i* ist. Auf der prozeduralen Ebene führt die Anweisung **while** ¬**isclosed.***i* **do** *i*?*x*; *o*!*x* **od**; *stop* := true dazu, daß der Strom i von *i* nach *o* kopiert wird. Die Regel lautet daher:

3. $A \rightarrow\!\rightarrow o \equiv i \quad \rightarrow \quad$ A; **while** ¬**isclosed.***i* **do** *i*?*x*; *o*!*x* **od**; *stop* := true

Man kann sich leicht klar machen, daß die Wirkung der Anweisung **while** … **od**; *stop* := true für alle mögliche "Varianten" von i korrekt arbeitet: Wenn i unendlich ist, terminiert die Schleife nicht und kopiert den gesamten Strom von sukzessive *i* nach *o*. Wenn i endlich, aber partiell ist, wird die Schleife irgendwann beim Versuch, **isclosed.***i* auszuwerten blockiert. Wenn i endlich und total ist, liefert **isclosed.***i* irgendwann true, die Schleife bricht ab, wegen *stop* := true bricht die äußere Schleife ab, und *o* wird durch **close.***o* abgeschlossen.

Die Korrektheit der vierten Regel ergibt sich unmittelbar aus der Korrektheit der dritten:

4. $n > 0$

$$A \rightarrow\!\rightarrow o \equiv \mathbf{rt}^{n}\mathbf{.}i \quad \rightarrow \quad A;\ i?x \rightarrow\!\rightarrow o \equiv \mathbf{rt}^{n-1}\mathbf{.}i$$

Für die Konfiguration $A \rightarrow\!\rightarrow o \equiv E_1 \,\&\, E_2$ ist folgendes zu beachten: Weil E_1 ein Objektausdruck ist, muß jeder Stromausdruck E', der in E_1 vorkommt, in der Form **ft.**E' bzw. **isempty.**E' vorkommen, denn **ft** und **isempty** sind die einzigen Funktionen, die Ströme auf Objekte abbilden. Aufgrund der syntaktischen Anforderungen an stromrepetitive Agenten folgt: E' = $\mathbf{rt}^{n}\mathbf{.}i$, $n \geq 0$. Weiterhin läßt sich jede Gleichung $o \equiv E_1 \,\&\, E_2$ so umformen, daß in E_1 kein ▯ und kein **if**.**then**.**else**.**fi** vorkommt[1]. Wir gehen daher davon aus, daß E_1 diese zusätzlichen Restriktionen erfüllt. Die Abbildung

$$\text{access: ‹exp›} \rightarrow \mathbf{Nat}^{\perp}$$

sei für Objektausdrücke E wie folgt definiert:

$$\text{access(E)} = \begin{cases} \perp & \text{falls, } \mathbf{ft} \text{ und } \mathbf{isempty} \text{ nicht in E vorkommen} \\ n & \text{falls, } \mathbf{isempty.rt}^{n}\mathbf{.}i \text{ in E vorkommt und} \\ & \quad \text{kein } \mathbf{ft.rt}^{m}\mathbf{.}i,\ m \geq n, \text{ und} \\ & \quad \text{kein } \mathbf{isempty.rt}^{m}\mathbf{.}i,\ m > n \\ n+1 & \text{falls, } \mathbf{ft.rt}^{n}\mathbf{.}i \text{ in E vorkommt und} \\ & \quad \text{kein } \mathbf{ft.rt}^{m}\mathbf{.}i,\ m > n, \text{ und} \\ & \quad \text{kein } \mathbf{isempty.rt}^{m}\mathbf{.}i,\ m > n \end{cases} .$$

access(E) gibt an, ob die Auswertung von E den Zugriff auf die Eingabe *i* nötig macht. Damit lassen sich für die Konfiguration $A \rightarrow\!\rightarrow o \equiv E_1 \,\&\, E_2$ zwei Fälle unterscheiden.

---

[1] ▯ kann mit den *choice*-Transformationen auf Seite 92 entfernt werden. **if.then.else.fi** kann ebenfalls durch Transformation entfernt werden. Dabei ist zu beachten, daß in E_1 nur strikte Operationen vorkommen.

Erstens: Die Auswertung von E_1 macht keinen Zugriff auf *i* notwendig. Dann ist E_1 auch ein (syntaktisch) korrekter PL-Ausdruck und der Gesamtstrom o, den die Konfiguration beschreibt, hat die Gestalt $o = o_1 \hat{} e \hat{} o_2$. Dabei ist o_1 der Strom, den die Anweisung A erzeugt, e das Element, das sich bei Auswertung von E_1 ergibt und o_2 der Strom, den der Stromausdruck E_2 definiert. Weil bei Auswertung von E_1 nicht auf *i* zugegriffen werden zu braucht, wird $o = o_1 \hat{} e \hat{} o_2$ auch durch die Konfiguration $A; o!E_1 \rightarrow o \equiv E_2$ beschrieben. Dies gilt auch, wenn die Auswertung von E_1 nicht terminiert ($e = \perp$). Die Regel lautet daher:

5a. $access(E_1) = \perp$

$$A \rightarrow o \equiv E_1 \,\&\, E_2 \quad \rightarrow \quad A; o!E_1 \rightarrow o \equiv E_2$$

Zweitens: Die Auswertung von E_1 macht einen Zugriff auf die Eingabe notwendig, d.h. in E_1 kommen Teilausdrücke der Art **ft.rt**^{n}**.***i* oder **isempty.rt**^{n}**.***i* vor. Dann ist E_1 kein PL-Ausdruck und muß erst in einen PL-Ausdruck $\underline{E}_1$ umgewandelt werden. Die Umformung besteht darin, (Teil-)Ausdrücke der Art **isempty.rt**^{n}**.***i* durch **isclosed.***i* und (Teil-)Ausdrücke der Art **ft.rt**^{n}**.***i* durch "head(tail^n(q))" zu ersetzen. An die Stelle des direkten Zugriffs auf *i*, tritt also auf der prozeduralen Ebene der Zugriff auf die Speicherschlange *q*. Das setzt voraus, daß auf *q* die "richtigen" Werte vorhanden sind. Vor Auswertung von $\underline{E}_1$ müssen daher (eventuell) Werte von *i* gelesen werden.

Beispiel 5.2: Gegeben sei folgender Agent:

agent *f* ≡ **chan nat** *i* → **chan nat** *o*:
  o ≡ 0 & **ft.rt**^{2}**.***i* & **ft.***i* & *f*(**rt**^{4}**.***i*)
end

Um **ft.rt**^{2}**.***i* auszuwerten, ist der Zugriff auf das dritte Element von *i* nötig (access(**ft.rt**^{2}**.***i*) = 3), und um **ft.***i* auszuwerten, der Zugriff auf das erste (access(**ft.***i*) = 1). Auf der applikativen Ebene sind Zugriffe auf *i* seiteneffektfrei, auf der prozeduralen Ebene führen sie dazu, daß der gelesene Wert entfernt wird. Um also in diesem Fall das dritte Element zu lesen, müssen auch die beiden ersten gelesen (und auf *q* gespeichert) werden. Die Auswertung von **ft.***i* macht dann aber keinen weiteren Zugriff auf *i* mehr nötig.
Beim ersten rekursiven Aufruf von *f*(**rt**^{4}**.***i*))sind dann nicht das 3. und 1., sondern das 7. und 5. Element zu lesen, während die ersten vier Elemente "gelöscht" werden können. Da sie schon teilweise eingelesen und auf *q* gespeichert wurden, müssen sie von dort wieder entfernt werden.

□

Das Beispiel macht deutlich, daß bei der Behandlung einer Konfiguration der Art $A \rightarrow o \equiv E_1 \,\&\, E_2$ Information darüber nötig ist, wieviel Elemente auf die Speicherschlange gelesen und wieviel von dort entfernt werden müssen. Diese Information ist laufzeit- bzw. eingabeabhängig, da in A Verzweigungen (**if** … **fi**) auftreten können, die eingabeabhängig durchlaufen werden. In der

folgenden Regel verwenden wir daher die Variablen *ap* und *rd*, die bei jedem Lade- und Reduktionsschritt aktualisiert werden. Die Regel lautet:

5b. $\text{access}(E_1) = n,\ n \geq 0$

$A \rightarrow\!\rightarrow o \equiv E_1 \,\&\, E_2$

$\rightarrow$ A; **while** *ap* < n **do** *i*?*x*; *q* := stock(*q*, *x*); *ap* := *ap* + 1 **od**;
while *rd* > 0 **do** *q* := tail(*q*); *rd* := *rd* - 1 **od**;
o!$\underline{E}_1 \rightarrow\!\rightarrow o \equiv E_2$

wobei $\underline{E}_1$ der Ausdruck ist, der aus E_1 entsteht wenn man jeden Teilausdruck der Art **ft.rt**^{n}**.***i* durch "head(tail^n(q))" und jeden Teilausdruck **isempty.rt**^{n}*i* durch **isclosed.***i* ersetzt. Wie oben beschreibt die Konfiguration auf der linken Seite von $\rightarrow$ den Strom $o = o_1 \hat{} e \hat{} o_2$, mit o_1, o_2 und e wie oben. Durch die beiden **while**-Schleifen wird in der Konfiguration auf der rechten Seite von $\rightarrow$ dafür gesorgt, daß alle benötigten Werte (**ft.rt**^{n}**.***i* tritt in E_1 auf) auf *q* geladen werden (z.B. referenziert "head(tail^n(q))" nach Ausführung beider Schleifen genau den Wert, den **ft.rt**^{n}**.***i* auf der applikativen Ebene referenziert). Deshalb liefert die Auswertung des PL-Ausdrucks $\underline{E}_1$ ebenfalls e. Also beschreiben beide Konfigurationen denselben Strom $o = o_1 \hat{} e \hat{} o_2$.

Beispiel 5.2 (Fortsetzung): Der Agent *f* von oben wird durch Anwendung der Regeln 5a und 5b in folgende Zwischenversion transformiert:

agent *f* ≡ **chan nat** *i* → **chan nat** *o*:
var bool *stop* := false;
var nat *x*; **var queue nat** *q* := eq;
var int *ap*, *rd* := 0, 0;
while ¬*stop* **do**
o!0;
while *ap* < 3 **do** *i*?*x*; *q* := stock(*q*, *x*); *ap* := *ap* + 1 **od**;
while *rd* > 0 **do** *q* := tail(*q*); *rd* := *rd* - 1 **od**;
o!head(tail^3(*q*));
while *ap* < 1 **do** *i*?*x*; *q* := stock(*q*, *x*); *ap* := *ap* + 1 **od**;
while *rd* > 0 **do** *q* := tail(*q*); *rd* := *rd* - 1 **od**;
o!head(*q*);

›› *f*(**rt**^{4}**.***i*)

od;
close.*o*
end

Offensichtlich wird das zweite Schleifenpaar nie durchlaufen, da nach Durchlauf des ersten Paares gilt: $ap \geq 3$ und $rd \leq 0$. Dies entspricht der oben gemachten Beobachtung, das für die Auswertung von **ft.***i* kein Zugriff auf *i* mehr notwendig ist. Wir schreiben verkürzt:

agent *f* ≡ **chan nat** *i* → **chan nat** *o*:

…
while ¬*stop* **do**
o!0;
while *ap* < 3 **do** *i*?*x*; *q* := stock(*q*, *x*); *ap* := *ap* + 1 **od**;
while *rd* > 0 **do** *q* := tail(*q*); *rd* := *rd* - 1 **od**;
o!head(tail^3(*q*)); *o*!head(*q*);

›› *f*(**rt**^{4}**.***i*)

od;
close.*o*

end
□

Für A ›› *o* ≡ **if** B **then** E_1 **else** E_2 **fi** gilt das gleiche wie für A ›› *o* ≡ E_1 & E_2. Auch hier sind zwei Fälle zu unterscheiden, nämlich, ob die Auswertung von B einen Zugriff auf *i* notwendig macht oder nicht. Im zweiten Fall müssen wieder Elemente von der Eingabe gelesen und auf *q* gespeichert werden. Die Korrektheitsargumentation verläuft in beiden Fällen analog zu den Regeln 5a und 5b. Die Regeln lauten (B ist genau oben definiert):

6a. access(B) = ⊥

A ›› *o* ≡ **if** B **then** E_1 **else** E_2 **fi**
→ A; **if** B **then** (**skip** ›› *o* ≡ E_1) **else** (**skip** ›› *o* ≡ E_2) **fi**

6b. access(B) = n, n ≥ 0

A ›› *o* ≡ **if** B **then** E_1 **else** E_2 **fi**
→ A; **while** *ap* < n **do** *i*?*x*; *q* := stock(*q*, *x*); *ap* := *ap* + 1 **od**;
while *rd* > 0 **do** *q* := tail(*q*); *rd* := *rd* - 1 **od**;
if B **then** (**skip** ›› *o* ≡ E_1) **else** (**skip** ›› *o* ≡ E_2) **fi**

Die Korrektheit der siebten Regel ist unmittelbar einleuchtend:

7. A ›› *o* ≡ E_1 ▯ E_2 → A; (**skip** ›› *o* ≡ E_1 ▯ **skip** ›› *o* ≡ E_2)

Um schließlich die Korrektheit der letzten Regel einzusehen, muß man sich ins Gedächnis rufen, daß jeder (rek.) Aufruf von *f* einem Durchlauf der **while** ¬*stop* **do** … **od** Schleife auf der prozeduralen Ebene entspricht. Die Konfiguration A ›› *o* ≡ *f*(**rt**^{n}**.***i*) muß also so übersetzt werden, daß auf der proz. Ebene ein neuer Schleifendurchlauf eingeleitet wird. Dabei ist folgendes zu beachten: Wenn auf der applikativen Ebene der rekursive Aufruf *f*(**rt**^{n}**.***i*) ausgewertet wird, so heißt das, daß auf die ersten n Werte von *i* nicht mehr zugegriffen werden kann. Auf der prozeduralen Ebene können sie von der Eingabe bzw. von *q* gelöscht werden. Dabei muß man berücksichtigen, daß das applikative *f* im Gegensatz zur prozedurale Leseoperation *i*?*x* nicht-strikt ist. Würde man die möglichen Lesezugriffe direkt bei der "Übersetzung" des rekursiven

Aufrufs ansiedeln, so könnte es dazu kommen, daß der prozedurale Agent divergiert, obwohl es sein applikatives Gegenstück nicht tut. Die entsprechenden Anweisungen müssen daher zurückgestellt werden und dürfen erst unmittelbar vor dem ersten "echt notwendigen" Zugriffspunkt ausgeführt werden. Dies erreicht man, indem man die Variablen *ap* und *rd* entsprechend setzt: Grob gesagt gilt: Wenn *rd* auf *rd* + n gesetzt wird, dann werden (im nächsten Durchlauf) die ersten n Elemente von *q* gelöscht und wenn *ap* auf *ap* - n gesetzt wird, dann werden (im nächsten Durchlauf) n neue Werte eingelesen. Die Regel lautet:

8. $A \rangle\rangle\ o \equiv f(\mathbf{rt}^n.i) \quad \rightarrow \quad A;\ ap := ap - n;\ rd := rd + n$

Damit läßt sich der Agent *f* aus unserem Beispiel vollständig übersetzen:

Beispiel 5.2 (Fortsetzung): Die prozedurale Version des Agenten *f* sieht wie folgt aus:

```
agent f ≡ chan nat i → chan nat o:
        …
        while ¬stop do
                o!0;
                while ap < 3 do i?x; q := stock(q, x); ap := ap + 1 od;
                while rd > 0 do q := tail(q); rd := rd - 1 od;
                o!head(tail^3(q)); o!head(q); ap := ap - 4; rd : = rd + 4
        od;
        close.o
end
```

Man beachte, daß beim ersten Durchlauf der **while** ¬*stop* **do** … **od** - Schleife die ersten drei Elemente von *i* gelesen werden und keines gelöscht wird, während beim zweiten Durchlauf die Elemente 4 - 7 gelesen werden (sofern vorhanden) und die Elemente 1 - 4 gelöscht werden. Im zweiten Durchlauf referenziert der Ausdruck head(tail^3(*q*)) also das siebte Stromelement. Dies entspricht genau dem Verhalten der applikativen Version von *f*. Der Beweis, daß die prozedurale Version und die applikative äquivalent sind, läßt sich auf der Grundlage der Semantiken von AL und PL erbringen (vgl. z.B. den Korrektheitsbeweis der Transformation *recursion-to-iteration I*).

□

Die Regeln des Kalküls sind am syntaktischen Aufbau der vorkommenden Gleichungen orientiert. Für jede "stromrepetitive" Gleichung gibt es eine Regel. Weiterhin ist die Regelmenge offensichtlich konfluent und noethersch, so daß sich aus einer Startkonfiguration (**skip** ›› $o \equiv E_1$) eindeutig eine Anweisung A ableiten läßt.

Für eine eingeschränkte Klasse von Agenten bietet dieser (Meta-)Kalkül die Möglichkeit zur schematischen Übersetzung. Er bildet sozusagen einen regelorientierten Compiler. Entsprechend ist der erzeugte "PL-Zielcode" nicht immer optimal. Geeignete Optimierungen auf der prozeduralen Ebene können sich, wie im Beispiel gesehen, anschließen.
In der vorliegenden Form sind stromrepetitive Agenten mit einem Eingabe- und einem Ausgabestrom behandelbar. Eine Erweiterung auf mehrere Eingabeströme und zusätzliche Objektparameter ließe sich einfach angeben.

5.3.2 Transformation von Agentennetzen

Wie in Abschnitt 3.5 beschrieben, entsprechen AL-Agenten Netzen kommunizierender Agenten. Auf der prozeduralen Ebene werden solche Netze mit Hilfe einer Parallelanweisung realisiert.

Parallelanweisungen haben die syntaktischen Gestalt von Gleichungssystemen. Sie sehen folgendermassen aus:

$$s_{11}, \ldots, s_{1n_1} \equiv f_1(t_{11}, \ldots, t_{1m_1}),$$
$$\ldots$$
$$s_{p1}, \ldots, s_{pn_p} \equiv f_p(t_{p1}, \ldots, t_{pm_p}).$$

Die i-te Gleichung repräsentiert den Aufruf des Agenten f_i (parallel zu allen übrigen) mit den aktuellen Eingabekanälen $t_{i1}, \ldots, t_{im_i}$ und den aktuellen Ausgabekanälen $s_{i1}, \ldots, s_{in_i}$.
Ein so definierter (hierarchischer) PL-Agent läßt sich sowohl syntaktisch als auch semantisch als AL-Agent betrachten und umgekehrt: Ein AL-Agent, dessen Rumpf aus einem Gleichungssystem besteht, das den syntaktischen Restriktionen von PL genügt (siehe Abschnitt 3.1), stellt auch einen PL-Agenten dar. Das bedeutet, daß ein AL-Agent in einen (hierarchischen) PL-Agenten transformiert werden kann, indem man das Gleichungssystem in seinem Rumpf so umformt, daß es die (strengeren) syntaktischen Bedingungen von PL erfüllt. Diese Umformungen können im wesentlichen auf der applikativen Ebene durchgeführt werden. Benutzt werden dazu die Netztransformationen aus Abschnitt 5.2.

Beispiel 5.3 (Transformation in hierarchische Form): Der applikative Agent f

agent $f \equiv$ **chan u** $i \rightarrow$ **chan u** o:
$o \equiv$ K[**ft.**$g(s_1)$, **ft.**$h(s_1)$] & f(**rt.**i),
$s_1 \equiv l(i)$
end

soll in einen hierarchischen PL-Agenten transformiert werden. g, h, l repräsentieren dabei (Aufrufe) andere(r) Agenten. f ist daher *nicht* stromrepetitiv.
Im ersten Schritt werden Agentenaufrufe und "komplizierte" Ausdrücke auf Parameterpositionen durch Strombezeichner ersetzt. Dies geschieht mit Hilfe der Regeln *netexpand* und *streamfold*:

agent $f \equiv$ **chan u** $i \rightarrow$ **chan u** o:
$o \equiv$ K[**ft.**s_2, **ft.**s_3] & s_4,
$s_1 \equiv l(i)$,
$s_2 \equiv g(s_1)$,
$s_3 \equiv h(s_1)$,
$s_4 \equiv f(s_5)$,
$s_5 \equiv$ **rt.**i
end

Im zweiten Schritt werden die definierenden Ausdrücke für o und s_5 durch Agentenaufrufe ersetzt. Dazu werden neue Deklarationen eingeführt (mit frischen Bezeichnern!) und die Regel *non-recursive agentfold* angewandt:

agent $f \equiv$ **chan u** $i \rightarrow$ **chan u** o:
$o \equiv k(s_2, s_3, s_4)$,

$s_1 \equiv l(i)$,
$s_2 \equiv g(s_1)$,
$s_3 \equiv h(s_1)$,
$s_4 \equiv f(s_5)$,
$s_5 \equiv rt(i)$
end

agent $k \equiv$ **chan u** $i_1, i_2, i_3 \rightarrow$ **chan u** o:
$o \equiv$ K[**ft.**i_1, **ft.**i_2] & i_3
end

agent $rt \equiv$ **chan u** $i \rightarrow$ **chan u** o:
$o \equiv$ **rt.**i
end

Schließlich muß die Doppelverwendung von i und s_1 durch *split*-Knoten beseitigt werden:

agent $f \equiv$ **chan u** $i \rightarrow$ **chan u** o:
(*)
$o \equiv k(s_2, s_3, s_4)$,
$s_1 \equiv l(i_1)$,

$s_2 \equiv g(s_{11})$,
$s_3 \equiv h(s_{12})$,

$s_4 \equiv f(s_5)$,
$s_5 \equiv rt(i_2)$,
$i_1, i_2 \equiv split(i)$,
$s_{11}, s_{12} \equiv split(s_1)$
end

agent $split \equiv$ **chan u** $i \rightarrow$ **chan u** o_1, o_2:
$o_1 \equiv i, o_2 \equiv i$
end

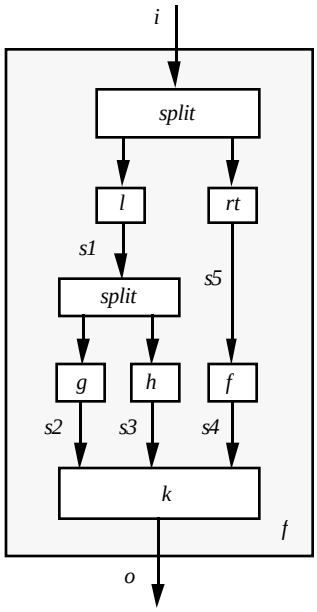
Die neu eingeführten Agenten *k*, *rt* und *split* sind trivialerweise stromrepetitiv. Sie lassen sich sehr einfach in prozedurale Form überführen:

agent $k \equiv$ **chan u** $i_1, i_2, i_3 \rightarrow$ **chan u** o:
var u x, y;
$i_1?x$; $i_2?y$; $o!$K[x, y];
while ¬**isclosed.**i_3 **do** $i_3?x$; $o!x$ **od**;
close.o
end

agent $rt \equiv$ **chan u** $i \rightarrow$ **chan u** o:
    var u x; $i?x$;
    while ¬**isclosed.**i **do** $i?x$; $o!x$ **od**;
    close.o
end

agent $split \equiv$ **chan u** $i \rightarrow$ **chan u** o_1, o_2:
    var u x;
    while ¬**isclosed.**i **do** $i?x$; $o_1!x$; $o_2!x$ **od**;
    close.o_1; **close.**o_2
end

Insgesamt wird f nun durch folgendes Netz realisiert:



Figur 5.5: Netzdarstellung des Agenten f

In der Form (*) erfüllt f die Kontextbedingungen von PL. Es wurde also in einen (hierarchischen) PL-Agenten transformiert.

□

Anhand des Beispiels wird die allgemeine Vorgehensweise bereits deutlich. Sie beruht auf fünf verschiedenen Transformationsregeln, die zumeist mehrfach angewandt werden müssen. Sei

agent $f \equiv$ **chan u** $i \rightarrow$ **chan v** o:
    $s_1 \equiv S_1, \ldots, s_i \equiv S_i, \ldots, s_n \equiv S_n$
end

ein AL-Agent für den gilt: Wenn in einem Ausdruck S_i ein Agentenaufruf $g(S)$ mit einem Objektparameter S vorkommt, dann ist S ein *reiner* Objektausdruck, d.h. die Operationen **ft** und **isempty** werden in S nicht benutzt. Der Grund für diese Einschränkung wird weiter unten erläutert.

Ein applikativer Agent, der dieser Bedingung genügt, kann mit folgenden Transformationsregeln in einen hierarchischen PL-Agenten transformiert werden. Da es sich bei allen Regeln um Kombinationen der Regeln aus Abschnitt 5.2 handelt, folgt ihre Korrektheit unmittelbar aus der Korrektheit der Basisregeln.

Transformation *separate***:** Herauslösen von Aufrufen aus Kontexten

agent $f \equiv$ **chan u** $i \rightarrow$ **chan v** o:
$s_1 \equiv S_1, \ldots, s_i \equiv S_i[g(S)], \ldots, s_n \equiv S_n$
end

\O(- NEW s
————————;↓)

agent $f \equiv$ **chan u** $i \rightarrow$ **chan v** o:
$s_1 \equiv S_1, \ldots, s_i \equiv S_i[s], s \equiv g(S), \ldots, s_n \equiv S_n$
end

Transformation *substitute***:** Substitution von Bezeichnern für Stromparameter

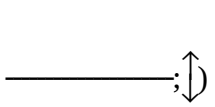
agent $f \equiv$ **chan u** $i \rightarrow$ **chan v** o:
$s_1 \equiv S_1, \ldots, s_i \equiv g(S), \ldots, s_n \equiv S_n$
end

\O(- NEW s,
S *ist ein Stromausdruck,* $S \notin$ SID
————————;↕)

agent $f \equiv$ **chan u** $i \rightarrow$ **chan v** o:
$s_1 \equiv S_1, \ldots, s_i \equiv g(s), s \equiv S, \ldots, s_n \equiv S_n$
end

Transformation *abstract***:** Abstraktion trivialer[1] Ausdrücke

agent $f \equiv$ **chan u** $i \rightarrow$ **chan v** o:
$s_1 \equiv S_1, \ldots, s_i \equiv S_i[s_{i_1}, \ldots, s_{i_k}], \ldots, s_n \equiv S_n$
end

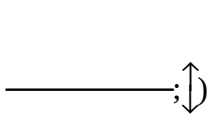


agent $f \equiv$ **chan u** $i \rightarrow$ **chan v** o:
$s_1 \equiv S_1, \ldots, s_i \equiv g(s_{i_1}, \ldots, s_{i_k}), \ldots, s_n \equiv S_n$
end
agent $g \equiv$ **chan** $\mathbf{w_1}$ $i_1, \ldots,$ **chan** $\mathbf{w_k}$ $i_k \rightarrow$ **chan** $\mathbf{w_{k+1}}$ o:
$o \equiv S_i[i_1/s_{i_1}, \ldots, i_k/s_{i_k}]$
end

Durch Anwendung dieser Regeln können die rechten Gleichungsseiten von f auf die Form $g(s_1, \ldots, s_k)$ bzw. $g(E, s_1, \ldots, s_k)$ gebracht werden, wobei E ein reiner Objektausdruck ist. Damit ist man schon fast am Ziel. Es ist allerdings noch möglich, daß ein Bezeichner s_i auf mehrfach auf rechten Gleichungsseiten vorkommt. Die Kontextbedingungen von PL schließen dies aus, da Kanäle gerichtete Punkt-zu-Punkt-Verbindungen darstellen. Durch Einführen von Split-Knoten können Doppelverwendungen beseitigt werden.

Transformation *splitintro***:** Einführen von split-Knoten

agent $f \equiv$ **chan u** $i \rightarrow$ **chan v** o:
$s_1 \equiv S_1, \ldots, s_i \equiv S_i[s], \ldots, s_j \equiv S_j[s], \ldots, s_n \equiv S_n$
end



agent $f \equiv$ **chan u** $i \rightarrow$ **chan v** o:
$s_1 \equiv S_1, \ldots, s_i \equiv S_i[t_1/s], \ldots, s_j \equiv S_j[t_2/s], \ldots, s_n \equiv S_n,$
$t_1, t_2 \equiv split(s)$
end

---

[1] Ein Stromausdruck möge "trivial" heißen, wenn kein Agentenaufruf in ihm vorkommt.

agent *split* ≡ **chan w** $i \rightarrow$ **chan w** o_1, o_2:
    $o_1 \equiv i, o_2 \equiv i$
end

Wenn ein Strombezeichner *s* mehrfach auf der rechten Seite *einer* Gleichung vorkommt, wird es ebenfalls notwendig, einen split-Knoten einzuführen. Die entsprechende Regel ist analog zur obigen.

Mit Ausnahme der beschriebenen Einschränkung, die sich auf die Verwendung von Strömen in Objektparametern bezieht, läßt sich jeder AL-Agent mit diesen Regeln in einen hierarchischen PL-Agenten transformieren. Die vier (bzw. fünf) Regeln sind also recht allgemein. Das Resultat kann man jedoch durchaus kritisch betrachten:

- Die rekursive Struktur der applikativen Ebene wird voll auf die prozedurale Ebene übertragen. Ein effizienzsteigernder Übergang von rekursiven zu iterativen Darstellungen findet nicht statt.
- Es entstehen eine Vielzahl "kleiner" Prozesse, deren interne Rechenleistung im Vergleich zum erforderlichen Kommunikationsaufwand gering ist. Erzeugt wird ein feingranulares paralleles Programm. Wenn man die Realisierung auf einem verteilten Multicomputersystem im Sinn hat (vgl. Abschnitt 3.5), ist das problematisch.

Der letzte Einwand kann jedoch entkräftet werden. Man kann nämlich in vielen Fällen auf der prozeduralen Ebene *Optimierungstransformationen* ausführen, die darauf abzielen, die Anzahl der Agenten zu reduzieren, und gleichzeitig die Komplexität der verbleibenden Agenten zu erhöhen (vgl. die "stream elimination transformations" in [Barstow 85]). Wir gehen dazu noch einmal auf das vorige Beispiel ein:

Beispiel 5.3 (Fortsetzung): Betrachtet man das oben für *f* entstandene Netz, so lassen sich in einem Schritt der *rt*-Knoten und der obere *split*-Knoten integrieren: *split'* übernimmt die Aufgabe beider Agenten:

agent *split'* ≡ **chan u** $i \rightarrow$ **chan u** o_1, o_2:
    var u *x*;
    if isclosed.*i* **then close.***o*$_1$ **else** *i*?*x*; o_1!*x* **fi**;
    while ¬**isclosed.***i* **do**
        i?*x*; o_1!*x*; o_2!*x*
    od;
    close.*o*$_1$; **close.***o*$_2$
end

Wenn die Definition von *l* bekannt und *l* ein sequentieller Agent ist, dann kann man *l* und den zweite *split*-Knoten zusammenfassen: Man ersetzt *l* durch *l'*, wobei *l'* ein Agent ist, der einen weiteren Ausgabekanal o_2 besitzt und dessen Rumpf genauso aussieht wie der von *l* bis auf die

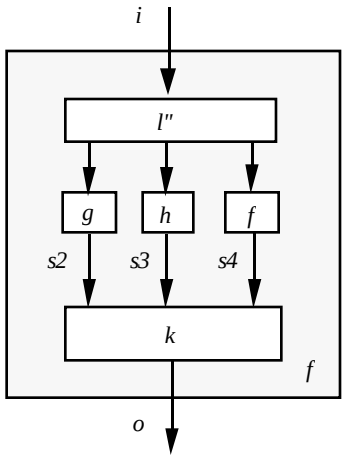
Tatsache, daß er überall wo *l* eine Ausgabe auf o_1 macht, dieselbe Ausgabe zusätzlich auf o_2 macht. *l'* könnte z.B. wie folgt aussehen:

agent $l' \equiv$ **chan u** $i \rightarrow$ **chan u** o_1, o_2:
    var u x, y;
    while ¬**isclosed.***i* **do** $i?x$; $y := L[x]$; $o_1!y$; $o_2!y$ **od**
    close.o_1; **close.**o_2
end

In einem weiteren Integrationsschritt können nun *l'* und *split'* zu *l''* zusammengefaßt werden:

agent $l'' \equiv$ **chan u** $i \rightarrow$ **chan u** o_1, o_2, o_3:
    var u x, y;
    if isclosed.*i* **then close.**o_1; **close.**o_2 **else** $i?x$; $y := L[x]$; $o_1!y$; $o_2!y$ **fi**;
    while ¬**isclosed.***i* **do**
        $i?x$; $y := L[x]$; $o_1!y$; $o_2!y$; $o_3!x$
    od;
    close.o_1; **close.**o_2; **close.**o_3
end

Insgesamt reduziert sich das Netz für *f* dann auf



Figur 5.6: Darstellung der optimierten Version von *f*

□

Die Transformationsregeln *separate*, *substitute*, *abstract* und *splitintro* erlauben den Übergang in prozedurale Form nur für solche Agenten, in deren Rumpf Objektparameter in einer "disziplinierten" Art und Weise gebraucht werden. Der Agent *sieve* aus Beispiel 3.4 gehört nicht dazu:

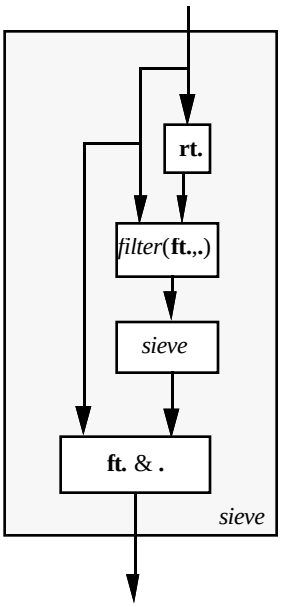
agent *sieve* ≡ **chan nat** *i* → **chan nat** *o*:

o ≡ **ft.***i* & *sieve*(*s*)

s ≡ *filter*(**ft.***i*, **rt.***i*)

end

filter wird hier mit einem nicht-reinen Objektausdruck, nämlich **ft.***i*, aufgerufen. Sein Objektparameter (vgl. die Definition von *filter* auf Seite 26) wird so wie ein zusätzlicher Eingabekanal genutzt. Das wird besonders deutlich, wenn man *sieve* entfaltet:



Figur 5.7: Entfaltete Darstellung von *sieve*

Eigentlich greift (diese Instanz von) *filter* also auf zwei Eingabekanäle zu.

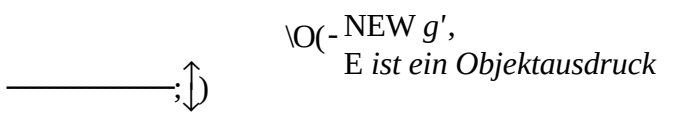
Generell führt die Verwendung von Strömen auf Objektparameterpositionen (in Verbindung mit den Operationen **ft** und **isempty**) zu *impliziten Kanälen*, die auf der prozeduralen Ebene *explizit* gemacht werden müssen. Es gibt mehrere Möglichkeiten, implizite Kanäle zu vermeiden. In Beispiel 4.2 ist eine prozedurale Version von *filter* angegeben, die mit einem Eingabekanal auskommt und keinen Objektparameter hat. Eine allgemeine Transformationsregel beruht darauf, Aufrufe mit nicht-reinen Objektparametern geeignet einzubetten:

Transformation *embed***:** Einbetten von Aufrufen mit nicht-reinen Objektparametern

agent *f* ≡ **chan u** *i* → **chan v** *o*:

$s_1 \equiv S_1, \ldots, s_i \equiv g(E[\textbf{ft.}s_j]), \ldots, s_n \equiv S_n$

end



agent $f \equiv$ **chan u** $i \rightarrow$ **chan v** o:

$s_1 \equiv S_1, \ldots, s_i \equiv g'(s_j), \ldots, s_n \equiv S_n$

end

agent $g' \equiv$ **chan w** $i \rightarrow$ **chan x** o:

$o \equiv g(E[\textbf{ft.}i])$

end

Der implizite Kanal, der auf die spezifische Verwendung des Objektparameters von *g* zurückgeht, wird durch den Übergang zu *g'* explizit gemacht, d.h. er tritt auch in der Parameterleiste von *g'* auf. Wenn *g* in einen sequentiellen PL-Agenten transformiert werden kann, z.B. weil es stromrepetitiv ist, dann läßt sich darauf aufbauend auch *g'* in einen sequentiellen Agenten überführen. Dazu kann man eine Variante der Regel *recursion-to-iteration III* auf Seite 112 benutzen. Wenn *g* selber hierarchisch ist, so muß der Aufruf im Rumpf von *g'* durch die Regel *non-recursive agentunfold* (Seite 99) aufgefalten werden, um den Objektausdruck in eine Gleichung "zu verlagern", die sich in einen Aufruf eines seq. PL-Agenten umwandeln läßt. Dies ist nicht immer möglich.

Aufgrund dieses Problems, ist die Regelmenge bestehend aus den fünf Regeln dieses Abschnitts und den Regeln zur Transformation stromrepetitiver Agenten (damit sind alle Regeln gemeint, die in Abschnitt 5.3.1 angegeben sind, sowie alle die, die sich mit dem dortigen Metakalkül ableiten lassen) *nicht* vollständig. Sie ist jedoch sehr allgemein: Jeder Agent für den das beschriebene Problem nicht auftritt, läßt sich mit diesen Regeln in einen hierarchischen PL-Agenten (bzw. ein System von hierarchischen und sequentiellen PL-Agenten) umwandeln.
Die dabei ausgeführten Transformationsschritte verlaufen sehr schematisch. Sie führen stets von rekursiven Ausgangsprogrammen zu rekursiven Zielprogrammen, d.h. zu potentiell unbeschränkten Netzstrukturen auch auf der prozeduralen Ebene. Das kann für manche Anwendungen durchaus erwünscht sein, weil so der Parallelisierungsgrad auf elegante Weise durch die Eingabe gesteuert wird. Die dadurch gewonnene Flexibilität stellt aber höhere Anforderungen an das ausführende *Laufzeitsystem*, das das dynamische Wachsen und Schrumpfen des Prozeßsystems kontrollieren muß. Soll ein hoher Grad an Parallelverarbeitung erreicht werden, ist es nötig, die neu entstehenden Prozesse "gut" auf die verfügbaren Prozessoren zu verteilen. In gewissem Sinn erfordert eine rekursive Programmstruktur daher dynamische Lastverteilungsverfahren (vgl. Abschnitt 4.3). Der Übergang von unbeschränkten Netzstrukturen auf der applikativen Ebene zu statischen Strukturen auf der prozeduralen Ebene ist Gegenstand des folgenden Abschnitts.

5.3.3 Behandlung nicht-repetitiver Rekursion

In den beiden vorigen Abschnitten sind zwei Möglichkeiten vorgestellt worden, applikative Agenten in prozedurale zu transformieren:

- Der erste Ansatz ist auf die Teilklasse der stromrepetitiven Agenten beschränkt. Er ermöglicht den Übergang von Rekursion zu Iteration.
- Der zweite Ansatz ist allgemeiner, führt aber auf rekursive Programmdarstellungen (rekursive Netze) auch auf der prozeduralen Ebene.

Rekursive Netze führen (zur Laufzeit) zu einer dynamisch wachsenden, potentiell unbeschränkten Anzahl von Prozessen. Auf diese Weise erreicht man eine gute Anpassung der Verteilungsstruktur des Programms an die Problemstellung und die speziellen Eingaben, und zwar ohne das dafür besondere aufschreibungstechnische Vorkehrungen getroffen werden müssen (z.B. explizites Ausprogrammieren der Prozeßkreation). Dies ist ein genereller Vorzug datenflußorientierter Programmierung.
Auf der anderen Seite setzen beschränkte Ressourcen der ungebremsten Netzexpansion natürliche Grenzen: Sobald mehr Prozesse entstanden, als Prozessoren verfügbar sind, muß zu einer teilweise quasiparallelen Abarbeitung übergegangen werden. Für Rechensysteme, die keine Prozeßmigration erlauben, kann es darüber hinaus generell günstiger sein, ein statisches Programm (mit fester Prozeßzahl) fix auf die gegebenen Prozessoren abzubilden. Man vermeidet dadurch die Ballung vieler (unverschickbarer) Prozesse auf einem Prozessor und damit eine sich mglw. stetig verschlechternde Lastverteilung. Technisch ist der Übergang von dynamischen zu statischen Netzen mit der Auflösung *nicht-repetitiver* Rekursion verbunden.

Im Bereich der sequentiellen Programmierung haben derartige Fragestellungen besonders in den 70er Jahren große Beachtung gefunden (vgl. [Strong 71], [Manna 74], aber auch [Bauer, Wössner 81] und [Stefãnescu 87]). Autoren wie Strong und Manna beschreiben in ihren Arbeiten bestimmte Klassen von *Flußgraphen* ("flow charts") und untersuchen, welche rekursiven Programmschemata sich in Flußgraphen transformieren, d.h. entrekursivieren, lassen. Die Darstellungsmächtigkeit der Flußgraphen hängt dabei wesentlich von den zugelassenen Strukturelementen ab:

Erlaubt man die Verwendung allgemeiner Stapel, so läßt sich jedes beliebige Programmschema entrekursivieren (vgl. [Bauer, Wössner 81], S. 235). Flußgraphen mit mindestens 2 Zählern sind ausdrucksstärker als Flußgraphen ohne Zähler (vgl. [Garland, Luckham 73], S. 126)
Paterson und Hewitt (vgl. [Paterson, Hewitt 70]) haben gezeigt, daß linear rekursive Schemata auch durch Flußgraphen darstellbar sind, die weder Stapel noch Zähler verwenden. In der gleichen Arbeit findet sich ein kaskadenartig rekursives Schema, von dem gezeigt wird, daß es sich nicht in "stapellose" Flußgraphen übersetzen läßt.

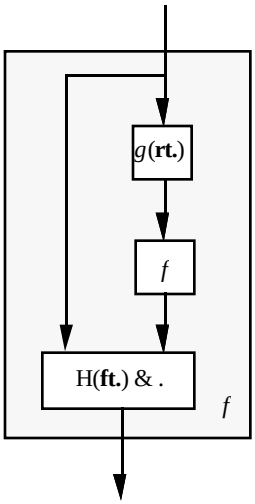
Die (Grund-)Funktionen, die in den Flußgraphen auftreten, werden in allen zitierten Arbeiten als *strikt* oder *total* interpretiert (vgl. [Manna 74], S. 244). Im Unterschied dazu, sind die in dieser Arbeit behandelten Agenten in der Regel *nicht-strikt* und außerdem über *nicht-flachen* Bereichen (Strömen), definiert. Der folgende Abschnitt kann daher als Ansatz zur Erweiterung der obigen Konzepte verstanden werden. Im Zusammenhang mit Nicht-Striktheit und Parallelverarbeitung ist der Übergang von dynamischen zu statischen Programmen das Gegenstück zum Übergang von rekursiven Programmschemata zu Flußgraphen aus dem Sequentiellen. Demonstriert werden soll in diesem Abschnitt die Idee, die diesem Übergang zugrunde liegt. Eine grundsätzliche Analyse,

und wenn möglich Anpassung, der vielen aus bekannten (seq.) Transformationsschemata (vgl. [Walker, Strong 73], [Bauer, Wössner 81]), kann hier nicht geleistet werden. Ebenso bleibt es späteren Arbeiten vorbehalten, analog zu den Resultaten von Paterson und Hewitt (vgl. [Paterson, Hewitt 70]) bzw. Strong (vgl. [Strong 71]), eine Klasse von Agenten syntaktisch abzugrenzen, die alle Agenten enthält, die auf statische Netze abgebildet werden können.

Wir betrachten im folgenden den Agenten:

agent $f \equiv$ **chan v** $i \rightarrow$ **chan w** o:
$o \equiv$ H[**ft.** i] & f(g(**rt.** i))
end

Dabei sei g ein weiterer stromverarbeitender Agent. f ist wegen des geschachtelten Aufrufs nicht stromrepetitiv, kann aber zu einem rekursiven Netz aufgefaltet werden:



Figur 5.8: Darstellung des nicht-repetitiven Agenten f

Die Frage, die nun untersucht werden soll, ist, ob, und wenn ja, wie, f durch ein statisches Netz oder einen sequentiellen Agenten realisieren werden kann. Dies hängt von den Eigenschaften von g ab. Wir betrachten zuerst einen speziellen und dann einen allgemeineren Fall:

Sei g wie folgt definiert:

agent $g \equiv$ **chan v** $i \rightarrow$ **chan v** o:
$o \equiv$ K[**ft.** i] & g(**rt.**i)
end

Der Einfachheit halber nehmen wir an, daß auf H und K die Attribute DET H und DET K zutreffen, dann läßt sich ein Aufruf von f wie folgt auffalten:

$f(i)$ = H[**ft.** i] & H[K[**ft.rt.** i]] & H[K[K[**ft.rt.rt.** i]]] & … & H[K^n[**ft.rt**^{n}**.**i]] …

Die Basisoperation K wird also n-mal auf das n-te Element der Eingabe angewandt (wobei wir mit dem Zählen bei 0 beginnen). Auf der Grundlage dieser Beobachtung kann *f* in einen sequentiellen Agenten transformiert werden:

Definiere einen Agenten *f'*, der sich über einen Zähler die Anzahl der gelesenen Eingaben merkt und K entsprechend oft anwendet. *f* wird auf *f'* abgestützt:

agent $f \equiv$ **chan v** $i \rightarrow$ **chan w** o:
    $o \equiv f'(0, i)$
end
agent $f' \equiv$ **nat** n, **chan v** $i \rightarrow$ **chan w** o:
    $o \equiv \mathrm{H}[K(\mathrm{n},$ **ft.** $i)] \,\&\, f'(n+1,$ **rt.** $i)$
end

funct $K \equiv$ **nat** n, **v** $x \rightarrow$ **v**:
    if $n = 0$ **then** x **else** $K(n\text{-}1, \mathrm{K}[x])$ **fi**
end

$K(n, x)$ ist die syntaktisch zulässige Darstellung von $K^n[x]$. Aus der applikativen Variante läßt sich mit Hilfe der Regel für eingebettete Agenten *recursion-to-iteration III* eine prozedurale Version ableiten:

agent $f \equiv$ **chan v** $i \rightarrow$ **chan w** o:
    var nat *zähler* := 0; **var v** x;
    loop
        $i?x$; $o!\mathrm{H}[K(zähler, x)]$; *zähler* := *zähler*+1
    pool
end

Dieser Übergang ist auch dann korrekt ist, wenn DET H und DET K nicht gelten. H[x] und K[x] müssen jedoch in x strikt sein, d.h. H[$\perp$] = $\perp$.

Das Verfahren, Rekursion über einen Zähler zu steuern (und letztlich durch Iteration abzulösen) ist eine klassische Methode zur Transformation rekursiver Programmschemata (vgl. [Manna 74], S. 330-331, [Bauer, Wössner 81], S. 306). Sie wird hier in geeigneter Weise auf den Fall nicht-strikter Operationen (nämlich &) zugeschnitten.

Wir betrachten nun den allgemeineren Fall und definieren dazu den Begriff der p-Synchronität:

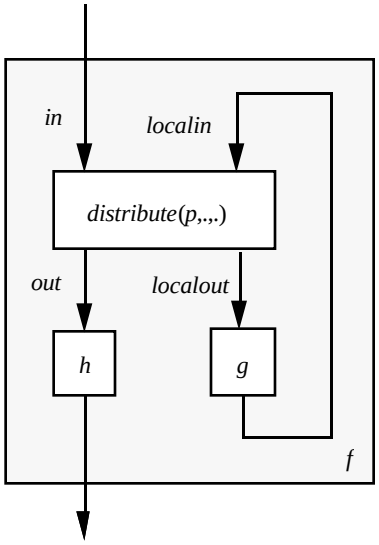
Sei f: $\mathbf{V}^\omega \rightarrow \mathbf{V}^\omega$ eine stromverarbeitende Funktion und $0 \neq p \in$ **Nat**. f heißt *p-synchron*, wenn für alle $v \in \mathbf{V}^\omega$ gilt:

$$\#v < p \Rightarrow \#f(v) = 0,$$
$$\#v \geq p \wedge f(v) \neq \perp \Rightarrow \#f(v) = 1 + \#f(rt^p(v)).$$

Dabei bezeichnet # den Längenoperator, der hier wie folgt definiert ist: $\#\varepsilon = \#\bot = 0$ und $v_0 \neq \bot \Rightarrow \#(v_0\&v) = 1 + \#v$. Ein Agent **agent** *f* … **end** heißt *p-synchron* (bzgl. einer Umgebung δ), wenn jedes $f \in \mathbf{F}_\delta[\![\textbf{agent } f \ldots \textbf{end}]\!]$ p-synchron ist.

p-synchrone Agenten erzeugen erzeugen erst dann eine Ausgabe, wenn der Eingabestrom mindestens p Elemente enthält. Ist das der Fall, so erzeugen sie aus je p Elementen der Eingabe ein Element der Ausgabe. Zum Beispiel gilt für ein p-synchrones f (das auf V^* total ist) und für $v \in V^*$ mit #v = p*n: #f(v) = n. Der auf der vorigen Seite definierte Agent *g* ist 1-synchron. Man beachte jedoch, daß nicht jede 1-synchrone Funktion auch eine "Map"-Funktion ist. (Zur Erinnerung, eine Map-Funktion ist eine stromverarbeitende Funktion f, die eine Basis-(objekt)funktion F elementweise auf ihren bzw. ihre Eingabeströme anwendet).

Wir betrachten nun erneut den Agenten *f* von Seite 133 und nehmen an, daß das dort aufgerufene *g* p-synchron ist und durch ein statisches Netz realisiert werden kann. Dann kann *f* durch das folgende statische Netz realisiert werden:



Figur 5.9: Statisches Netz zur Realisierung p-synchroner Agenten

Die grundlegende Idee ist dabei dieselbe wie zuvor: Das erste Element des Eingabestroms wird sofort an *h* weitergeleitet, wobei *h* die durch den Ausdruck H bestimmte Map-Funktion ist:

agent $h \equiv$ **chan v** $i \rightarrow$ **chan w** o:
  $o \equiv$ **if isempty.** i **then** ε **else** H[**ft.** i] & h(**rt.** i) **fi**
end

Die nächsten Elemente der Eingabe müssen *g* mit zunehmender Häufigkeit durchlaufen. Der Agent *distribute* steuert diesen Prozeß. Seine Regelmäßigkeit beruht auf der p-Synchronität von *g*. Würde die Anzahl der benötigen Elemente in Abhängigkeit von der Eingabe schwanken, wäre das Problem so nicht lösbar. *distribute* ist wie folgt definiert:

```
agent distribute ≡ nat p, chan v in, localin → chan v out, localout:
        var nat n := 0; var nat a; var v x;
        in?x; out!x;
        loop
                n := n+1; a := 1;
                while a ≤ p^n do  in?x; localout!x; a := a+1  od;
                a := 1;
                while a ≤ Σ_{i=1}^{n-1} p^i do  localin?x; localout!x; a := a+1  od;
                localin?x; out!x
        pool
end
```

distribute leitet das erste Element von *in* direkt auf *out* weiter und arbeitet danach so:
Im n-ten Durchlauf der **loop**-Schleife werden p^n Elemente von *in* gelesen und auf *localout* ausgegeben, d.h. an *g* gesandt (erste **while**-Schleife). Weil *g* p-synchron ist, reduziert es die p^n Elemente zu p^{n-1} Elementen, die wieder an *distribute* zurückgehen. *distribute* liest sie ein und sendet sie erneut an *g* (zweite **while**-Schleife). *g* reduziert die p^{n-1} Werte zu p^{n-2} Werten usw..
Das ganze geht solange, bis *g* insgesamt n-mal angewandt wurde. Die p^n Anfangswerte wurden dann zu einem Ergebniswert reduziert. Der wird schließlich von *localin* gelesen, auf *out* ausgegeben und gelangt über *h* nach außen.
Falls p = 1 ist, sorgt *distribute* dafür, daß das n-te Element der Eingabe n-mal in der *distribute-g*-Schleife zirkuliert, bevor es an *h* weitergereicht wird.
distribute ist hier prozedural realisiert. Im Sinne der auf Seite 128 angedeuteten Optimierungen kann man *distribute* und *h* verschmelzen. Wenn *g* als sequentieller Agent oder zumindest als statisches Netz realisiert werden kann, dann ist das gesamte Netz statisch.
Man beachte, daß *distribute* und *g* und *h* durchaus parallel arbeiten können: Während *distribute* z.B. dabei ist, die p^n Werte von *in* zu lesen, und schon einen Teil auf *localout* ausgegeben hat, kann *g* seine Berechnungen mit diesen Werten bereits beginnen.

Offenbar ist diese Transformation nicht universell anwendbar. Sie berücksichtigt weder Agenten, die nicht p-synchron sind, noch andere Rekursionsformen. Das Konzept, das hinter dem Übergang von dynamischen zu statischen Programmen steht, wird aber deutlich. Es besteht darin die Rekursion auf der Funktionsebene (*Funktionsrekursion*) durch Rekursion auf der Stromebene (*Stromrekursion*) abzulösen. Während erstere für dynamische Netze sorgt, führt letzere zu rückgekoppelten Netzen.
Funktionsrekursion führt zu "Rekursion im Ort": Es entstehen verschiedene Instanzen eines Agenten (oben *g*), die gleichzeitig arbeiten. Die einzelnen Elemente der Eingabe werden an verschiedenen Orten bearbeitet.
Stromrekursion führt zu "Rekursion in der Zeit": Es gibt nur eine Instanz des betreffenden Agenten, der die Eingaben nacheinander bearbeitet.

Die in Kapitel 5 eingeführten Transformationsregeln zielen, wenn man sie insgesamt betrachtet, darauf ab, AL-Programme in PL-Programme zu transformieren. Sie sind also in "Entwicklungsrichtung" (Top-Down) ausgerichtet. In der Literatur wird teilweise auch die entgegengesetzte, "semantische" Richtung (Bottom-Up) studiert (vgl. [Pepper 79], [Broy 80]). Bei letzterer Betrachtungsweise dienen die Regeln dazu, die Semantik bestimmter Sprachkonstrukte, durch

(Rück-) Transformation in andere zu erklären. Transformationsregeln sind dann Mittel zur *Semantikdefinition*. Ihre Korrektheit braucht nicht mehr gesondert bzgl. unabhängig vorliegender Sprachsemantiken nachgewiesen werden.

Die in Kapitel 5 vorgestellten AL $\rightarrow$ PL - Transformationen kann man zumindest in ihrer jetzigen Form *nicht* als transformationelle Semantikdefinition von PL ansehen. Dazu wäre (syntaktische) Vollständigkeit in Bezug auf PL notwendig: Jedes PL-Programm müßte mit den gegebenen Transformationen auf ein AL-Programm zurückgeführt werden können. Aufgrund der methodischen "Top-Down"-Orientierung der Regeln, werden aber nur bestimmte PL-Programme "getroffen". Gerade so wie ein Compiler nicht alle syntaktisch zulässigen Programme seiner Zielsprache erzeugt.

Sei Trans(AL) $\subseteq$ PL die Menge der PL-Programme, die durch die angegebenen Transformationen "getroffen" werden. Eine Möglichkeit, diese Menge in Bezug auf PL zu vervollständigen, besteht darin, auf der PL-Ebene Regeln anzugeben, durch die jedes Programm aus PL\Trans(AL) in ein Programm aus Trans(AL) übersetzt werden kann. Die Programme aus Trans(AL) haben eine bestimmte Form. Sie bestehen aus hierarchischen Agenten, die sich als Ergebnis der Transformationen aus Abschnitt 5.3.2 ergeben und bestimmten sequentiellen Agenten, die sich als Ergebnis der Transformationen aus Abschnitt 5.3.1 ergeben (inklusive der Regeln, die mit dem in 5.3.1 angegebenen Metakalkül abgeleitet werden können). Da hierarchische PL-Agenten kanonisch als PL-Agenten aufgefaßt werden können, liegt die Hauptschwierigkeit bei der angesprochenen Übersetzung darin, beliebige sequentielle PL-Agenten auf die in Trans(AL) vorkommenden Formen zu bringen. Sequentielle Agenten aus Trans(AL) sind i.w. dadurch gekennzeichnet, daß ihr Rumpf aus einer umfassenden **while**-Schleife besteht (vgl. das Schema auf Seite 115), innerhalb derer endlich viele Elemente von den Eingabekanälen gelesen und endlich viele Elemente auf die Ausgabekanäle ausgegeben werden. Prinzipiell, d.h. vom Standpunkt semantischer Äquivalenz, ist es keine Schwierigkeit, jeden beliebigen seq. Agenten in eine solche Darstellung zu überführen. Rein syntaktisch sind hierzu jedoch sehr viele Regeln notwendig. Analog zu dem Metakalkül aus Abschnitt 5.3.1, wäre es daher sinnvoll, einen weiteren Metakalkül zu entwickeln, mit dem sich die entsprechenden Regeln herleiten liessen.

6. Ausblick

In den vorangegangenen Kapiteln sind zwei *algorithmische Sprachen* – AL und PL – zur Beschreibung verteilter Systeme vorgestellt worden. Beide Sprachen sind syntaktisch "schmal", aber mit voll ausgearbeiteten *denotationellen Semantiken* versehen, die sich ganz bewußt auf auf die gleichen mathematischen Konzepte, insbesondere Ströme und stromverarbeitende Funktionen stützen. Dadurch passen sich AL und PL bruchlos in den Rahmen von FOCUS, der übergeordneten Entwurfsmethode (vgl. Abschnitt 1.1), ein und sind auch untereinander kompatibel.

Der Übergang von AL nach PL erfolgt durch *Transformation*. Regeln dafür sind im fünften Kapitel angegeben. Ihre Korrektheit wird dort zum großen Teil bewiesen. In die Beweise fließen die semantischen Definitionen aus den Kapiteln 3 und 4 direkt ein. Die Beweisführung wird durch die Uniformität der Semantiken erleichtert.

Methodisch haben sich zwei Möglichkeiten zur Transformation von AL-Programmen in prozedurale Form herauskristallisiert:
Die eine besteht in der Anwendung von Gleichungssystemtransformationen. Sie führt zu einem hohen Parallelisierungsgrad und in der Regel zu rekursiven prozeduralen Programmen.
Die andere versucht, rekursive Strukturen der applikativen Ebene auf statische, aber rückgekoppelte Netze auf der prozeduralen Ebene abzubilden.
In beiden Fällen werden Agenten einer speziellen Rekursionsform (*stromrepetitive Agenten*) in sequentielle PL-Agenten übersetzt, deren Rumpf im wesentlichen aus einer while-Schleife besteht. Während der erste Weg in den meisten Fällen zum Ziel führt, eine Konsequenz auch der syntaktischen Abstimmung zwischen den Sprachen, stellt der andere härtere (semantische) Bedingungen an das Ausgangsprogramm. Ähnlich gelagerte Problemstellungen sind für sequentielle Programme unter dem Stichwort "Transformation rekursiver Programmschemata in Flußgraphen" untersucht worden.
Für die Klasse der stromrepetitiven Agenten enthält die Arbeit einen Metakalkül, der es erlaubt, zu jedem dieser Agenten eine korrekte Transformationsregel und damit eine äquivalente prozedurale Version abzuleiten.

Es ist fast natürlich, daß bei einer Arbeit wie der vorliegenden Fragen offen bleiben bzw. neu aufgeworfen werden. Einige weiterführende Untersuchungspunkte sollen im folgenden kurz angeführt werden:

An erster Stelle ist dabei die *praktische Evaluation* des skizzierten Transformationsansatzes zu nennen. Die eingestreuten kleineren Beispiele können eine umfangreiche und aussagekräftige *Fallstudie* nicht ersetzen. Wünschenswert wäre eine (Beispiel-)Entwicklung, die sich über alle Phasen der FOCUS-Methodik erstreckt und so die Durchgängigkeit des Gesamtansatzes aufzeigt. Erfahrungen, die dabei anfallen, würden sicher zu einer Reihe von Verbesserungen führen und außerdem zur Anreicherung der Bibliothek vorhandener Regeln beitragen.

Methodisch bedeutsam wären darüber hinaus Untersuchungen, die aufzeigen wie der bestehende *Trade-Off* zwischen hohem Parallelisierungsgrad und rekursiven Programmstrukturen auf der einen Seite und grobgranularen Prozessen und statischen Strukturen auf der anderen Seite adäquat ausbalanciert werden kann. Daraus liessen sich Rückschlüsse ziehen, in welcher "Mischung" Transformationsregeln der oben beschriebenen Typen zum Einsatz kommen sollten. Ergebnis wäre eine Strategie, die dem Anwender Leitlinien zur zielgerichteten Anwendung der einzelnen Regeln an die Hand gibt.

Untersuchungen zur Übersetzung von dynamischen in statische Strukturen können auf den Ergebnissen aufbauen, die für den sequentiellen Fall vor allem in den 70er Jahren erzielt wurden. Die dort entwickelten Transformationsregeln müßten systematisch daraufhin analysiert werden, ob und in welcher Weise sie auf nicht-strikte Funktionen über nicht-flachen Bereichen angepaßt werden können. Weitergehend könnte man dann (im Sinne von Paterson und Hewitt) nach *theoretischen Resultaten* suchen, um die Klasse der überhaupt statisch realisierbaren Agenten-schemata zu charakterisieren.

Es ist klar, daß beim Übersetzung von dynamischen zu statische Agentenprogrammen nicht nur die spezielle Anwendung, sondern auch die angestrebte Zielarchitektur berücksichtigt werden muß. Sie bestimmt das, was als "gute" Balance angesehen werden kann, wesentlich mit. Um derartige Qualitätsaussagen formal faßbar zu machen, muß nach einem aussagefähigen Maßstab gesucht werden. Praktische Untersuchen zu diesem Fragenkomplex (z.B. Messungen) machen die *Implementierung* von AL und/oder PL erforderlich. Dieser Aspekt ist im Verlauf der Arbeit bereits an einzelnen Stellen angeklungen. Konkret müßte hierzu ein Compiler entwickelt werden, der PL z.B. auf den MMK (vgl. Abschnitt 4.3) abbildet.

AL und insbesondere PL sind bewußt mit einfachen Sprachmitteln für die Kommunikation ausgestattet: nicht-blockierendes Senden (!), blockierendes Empfangen (?) und einem blockierenden Kanaltest (isclosed). Grund für diese Festlegung ist der Wunsch, die denotationelle Semantik handhabbar zu halten. Weitergehend könnte man *andere Kommunikationsprimitive* untersuchen. Denkbar wären nicht-blockierendes Lesen oder Abfragen, disjunktives Warten (auf zwei oder mehr Kanäle) oder explizite Realzeit-Konstrukte ("delay" aus ADA). Es ergäben sich dann erweiterte Programmiermöglichkeiten. Die semantische Behandlung würde jedoch komplizierter. Der formale Umgang mit Realzeitsprachen, das erweiterte PL wäre dann eine Realzeitsprache, ist ein augenblicklich vielbeachtetes Forschungsfeld. Wirklich befriedigende allgemein anerkannte und praktikable Lösungen haben sich aber bisher noch nicht herauskristallisiert.

Quellenverzeichnis

[Back, Sere 91] R.J.R. Back, K. Sere: Deriving an Occam Implementation of Action Systems. In: C. Morgan, J.C.P. Woodcock (eds.): *3rd Refinement Workshop,* Series: Workshops in Computing, Springer 1991, 9-30

[Backus 78] J. Backus: Can Programming Be Liberated from the Von Neumann Style? A Functional Style and Its Algebra of Programms. *Communications of the ACM **21**(8),* 1978, 613-641

[Bal et al. 89] H.E. Bal, J.E. Steiner, A.S. Tanenbaum: Programming Languages for Distributed Computing Systems. *ACM Computing Surveys **21**(3),* 1989, 261-322

[Barendregt 90] H.P. Barendregt: Functional Programming and Lambda Calculus. In: J. van Leeuwen (ed.): *Handbook of Theoretical Computer Science, Vol. B,* Elsevier 1990, 321-363

[Barstow 85] D. Barstow: Automatic Programming for Streams. In: *IJCAI 85, Proc. 9th International Joint Conference on Artificial Intelligence, Vol. I,* 1985, 232-237

[Barstow 88] D. Barstow: Automatic Programming for Streams II. In: *Proc. 10th International Conference on Software Engineering,* 1988, 439-447

[Bauer et al. 89] F.L. Bauer, B. Möller, H. Partsch, P. Pepper: Formal Program Construction by Transformations – Computer-Aided, Intuition-Guided Programming. *IEEE Transactions on Software Engineering **15**(2),* 1989, 165-180

[Bauer, Wössner 81] F.L. Bauer, H. Wössner: *Algorithmische Sprachen und Programmentwicklung.* Springer 1981

[Bemmerl, Ludwig 90] T. Bemmerl, T. Ludwig: MMK – A Distributed Operating System Kernel with Integrated Dynamic Loadbalancing. In: H. Burkhart (ed.): *CONPAR '90 – VAPP IV,* LNCS 457, Springer 1990, 744-755

[Bemmerl et al. 90a] T. Bemmerl, A. Bode, P. Braun, O. Hansen, P. Luksch, R. Wismüller: TOPSYS – Tools for Parallel Systems (User´s Overview and User´s Manual). Technische Berichte des Instituts für Informatik der TU München, TUM-I9047, SFB-Bericht Nr. 342/25/90 A, 1990

[Bemmerl et al. 90b] T. Bemmerl, A. Bode, T. Ludwig, S. Tritscher: MMK – Multiprocessor Multitasking Kernel (User´s Guide and User´s Reference Manual). Technische Berichte des Instituts für Informatik der TU München, TUM-I9048, SFB-Bericht Nr. 342/26/90 A, 1990

[Berghammer 90] R. Berghammer: Transformational Programming with Non-Deterministic and Higher-order Constructs. Universität der Bundeswehr München, Fakultät für Informatik, Bericht Nr. 9012, 1990

[Berghammer et al. 90] R. Berghammer, H. Ehler, B. Möller: On the Refinement of Non-Deterministic Routines by Transformations. In: *Proc. IFIP TC 2 Working Conference on Programming Concepts and Methods*, 1990, 51-69

[Bird 89] R.S. Bird: Lectures on Constructive Functional Programming. In: M. Broy (ed.): *Constructive Methods in Computing Science*, NATO ASI Series F: Computer and System Sciences, Vol. 55, Springer 1989, 151-216

[Bjørner et al. 89] D. Bjørner, C.A.R. Hoare, J. Bowen, H. JiFeng, H. Langmaack, E.-R. Olderog, U. Martin, V. Stavridou, F. Riis Nielson, H. Riis Nielson, H. Barringer, D. Edwards, H.H. Løvengreen, A.P. Ravn, H. Rischel: A ProCoS Projekt Description ESPRIT BRA 3104. *Bulletin of the EATCS 39*, 1989, 60-73

[Breu 90] M. Breu: Development of Implementations. In: [Krieg-Brückner 90] Vol. I, Section 2.2, 1990

[Brock, Ackerman 81] J.D. Brock, W.B. Ackerman: Scenarios: A Model of Non-deterministic Computation. In: J. Diaz, I. Ramos (eds.): *Foundations of Programming Concepts*, LNCS 107, Springer 1981, 252-259

[Broy 80] M. Broy: Transformation parallel ablaufender Programme. Technische Berichte des Instituts für Informatik der TU München, TUM-I8001, 1980

[Broy 85] M. Broy: Extensional Behaviour of Concurrent, Nondeterministic, Communicating Programs. In: M. Broy (ed.): *Control Flow and Dataflow, Concepts of Distributed Programming*, NATO ASI Series F: Computer and System Sciences, Vol. 14, Springer 1985, 229-276

[Broy 86] M. Broy: A Theory for Nondeterminism, Parallelism, Communication, and Concurrency. *Theoretical Computer Science **45***, 1986, 1-61

[Broy 87a] M. Broy: Specification and Top-Down Design of Distributed Systems. *Journal of Computer and System Science **34**(2/3)*, 1987, 236-265

[Broy 87b] M. Broy: Semantics of Finite and Infinite Networks of Concurrent Communicating Agents. *Distributed Computing **2**(1)*, 1987, 13-31

[Broy 87c] M. Broy: Predicative Specifications for Functional Programs Describing Communicating Networks. *Information Processing Letters **25***, 1987, 93-101

[Broy 88a] M. Broy: Views of Queues. *Science of Computer Programming* ***11***, 1988, 65-86

[Broy 88b] M. Broy: An Example for the Design of a Distributed System in a Formal Setting: The Lift Problem. Technische Berichte der Fakultät für Mathematik und Informatik, Universität Passau, MIP 8802, 1988

[Broy 89] M. Broy: Towards a Design Methodology for Distributed Systems. In: M. Broy (ed.): *Constructive Methods in Computing Science,* NATO ASI Series F: Computer and System Sciences, Vol. 55, Springer 1989, 311-364

[Broy 90] M. Broy: Functional Specification of Time Sensitive Communcating Systems. Working Material, International Summer School on Programming and Mathematical Method, Marktoberdorf, Germany, 1990

[Broy, Lengauer 91] M. Broy, C. Lengauer: On Denotational versus Predicative Semantics. *Journal of Computer and Systems Sciences* ***42(1),*** 1991, 1-29

[Broy et al. 92a] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. Gritzner, R. Weber: The Design of Distributed Systems – An Introduction to FOCUS. Technische Berichte des Instituts für Informatik der TU München, TUM-I9202, SFB-Bericht Nr. 342/2/92 A, 1992

[Broy et al. 92b] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. Gritzner, R. Weber: Summary of Case Studies in FOCUS – a Design Method for Distributed Systems. Technische Berichte des Instituts für Informatik der TU München, TUM-I9203, SFB-Bericht Nr. 342/3/92 A, 1992

[Burstall, Darlington 77] R.M. Burstall, J. Darlington: A Transformation System for Developing Recursive Programs. *Journal of the ACM* ***24(1),*** 1977, 44-67

[Chandy, Misra 88] K.M. Chandy, J. Misra: *Parallel Program Design, A Foundation.* Addison-Wesley 1988

[CIP 85] The CIP Language Group: *The Munich Project CIP. Volume I: The Wide Spectrum Language CIP-L.* LNCS 183, Springer 1985.

[Clinger 82] W. Clinger: Nondeterministic Call by Need is Neither Lazy Nor by Name. In: *Proc. ACM Symposioum on LISP and Functional Programming, Pittsburgh (Penn.),* 1982, 226-234

[Dederichs 90] F. Dederichs: Transforming Distributed Systems. unveröffentlichtes Manuskript, TU München, 1990

[Dennis 74] J.B. Dennis: First Version of a Data Flow Procedure Language. In: B. Robinet (ed.): *Colloque sur la Programmation,* LNCS 19, Springer 1974, 362-367

[Dennis 85] J.B. Dennis: Data Flow Computation. In: M. Broy (ed.): *Control Flow and Dataflow, Concepts of Distributed Programming,* NATO ASI Series F: Computer and System Sciences, Vol. 14, Springer 1985, 346-397

[Duncan 90] R. Duncan: A Survey of Parallel Computer Architectures. *IEEE Computer **33**(2),* 1990, 5-16

[Eriksen, Prehn 91] K.E. Eriksen, S. Prehn: RAISE Overview. RAISE/CRI/DOC/9/V4, 1991

[Feather 87] M.S. Feather: A Survey and Classification of Some Program Transformation Approaches and Techniques. In: L.G.L.T. Meertens (ed.): *Program Specification and Transformation,* Elsevier 1987, 165-195

[Field, Harrison 88] A.J. Field, P.G. Harrison: *Functional Programming.* Addison Wesley 1988

[Francez, Forman 91] N. Francez, I.R. Forman: Synchrony Loosening Transformations for Interacting Processes. In: J.C.M. Baeten, J.F. Groote (eds.): *CONCUR '91, 2nd International Conference on Concurrency Theory,* LNCS 527, Springer 1991, 203-219

[Garland, Luckham 73] S.J. Garland, D.C. Luckham: Program Schemes, Recursion Schemes, and Formal Languages. *Journal of Computer and Systems Sciences **7**,* 1973, 119-160

[George, Milne 91] C.W. George, R.E. Milne: Specifiying and Refining Concurrent Systems – An Example from the RAISE Project. In: C. Morgan, J.C.P. Woodcock (eds.): *3rd Refinement Workshop,* Series: Workshops in Computing, Springer 1991, 155-168

[Glasgow, MacEwen 89] J.I. Glasgow, G.H. MacEwen: An Operator Net Model for Distributed Systems. *Distributed Computing **3**(4),* 1989, 159-177

[Gordon 79] M. Gordon: *The Denotational Description of Programming Language.* Springer 1979.

[Gunter, Scott 90] C.A. Gunter, D.S. Scott: Semantic Domains. In: J. van Leeuwen (ed.): *Handbook of Theoretical Computer Science, Vol. B,* Elsevier 1990, 633-674

[Harel 87] D. Harel: Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming **8**,* 1987, 231-274

[Hehner 84] E.C.R. Hehner: Predicative Programming Part I + II. *Communications of the ACM **27**(2),* 1984, 134-151

[Herath et al. 1988] J. Herath, Y. Yamaguchi, N. Saito, T. Yuba: Dataflow Computing Models, Languages, and Machines for Intelligence Computing. *IEEE Transactions on Software Engineering **14**(12),* 1988, 1805-1828

[Hoare 78] C.A.R. Hoare: Communicating Sequential Processes. *Communications of the ACM **21**(8),* 1978, 666-677

[Hoare 85a] C.A.R. Hoare: *Communicating Sequential Processes.* Prentice Hall 1985

[Hoare 85b] C.A.R. Hoare: Programs are Predicates. In: C.A.R. Hoare, J.C. Shepherdsen (eds.): *Mathematical Logic and Programming Languages,* Prentice Hall 1985, 141-155

[Hussmann 91] H. Hussmann: Nondeterministic Algebraic Specifications. Technische Berichte des Instituts für Informatik der TU München, TUM-I9104, 1991

[Hutner, Holzner 89] F. Hutner, R. Holzner: Architektur, Programmierung und Leistungsbewertung des MIT-Datenflußrechners. *Informatik-Spekturm **12**(3),* 1989, 147-157

[Jones, Sinclair 89] S.B. Jones, A.F. Sinclair: Functional Programming and Operating Systems. *The Computer Journal **32**(2),* 1989, 162-174

[Jonsson 87] B. Jonsson: Compositional Verification of Distributed Systems. Ph.D. Thesis, Department of Computer Systems, Uppsala University, Uppsala, Sweden, DoCS 87/09, 1987

[Jonsson 89] B. Jonsson: A Fully Abstract Trace Model for Dataflow Networks. In: *Proc. 16th Annual ACM Symposium on Principles of Programming Languages,* 1989, 155-165

[Kahn 74] G. Kahn: The Semantics of a Simple Language for Parallel Programming. In: J.L. Rosenfeld (ed.): *Information Processing 74,* North Holland 1974, 471-475

[Kahn, MacQueen 77] G. Kahn, D. MacQueen: Coroutines and Networks of Parallel Processes. In: B. Gilchrist (ed.): *Information Processing 77,* North Holland 1977, 993-998

[Keller 78] R.M. Keller: Denotational Models for Parallel Programs with Indeterminate Operators. In: E.J. Neuhold (ed.): *Formal Description of Programming Concepts,* North Holland 1978, 337-366

[Kott 80] L. Kott: A System for Proving Equivalences of Recursive Programs. In: W. Bibel, R. Kowalski (eds.): *5th Conference on Automated Deduction,* LNCS 87, Springer 1980, 63-69

[Krieg-Brückner 90] B. Krieg-Brückner (ed.): PROgram Development by SPECification and TRAnsformation. Vol. I (Methodology), PROSPECTRA Report M.1.1.S3-R-55.2. Vol. II (Language Family) PROSPECTRA Report M.1.1.S3-R-56.2, Vol. III (System) PROSPECTRA Report M.1.1.S3-R-57.2, 1990

[Kröger 87] F. Kröger: *Temporal Logic of Programs.* EATCS Monograph 8, Springer 1987

[Lamport 89] L. Lamport: A simple Approach to Specifying Concurrent Systems. *Communications of the ACM **32**(1),* 1989, 32-45

[Løvengreen 85] H.H. Løvengreen: On Concurrency Formalization. ID-TR 1985-3, Instituttet f. Datateknik, Danmarks Tekniske Høsjkole, Lyngby 1988

[Lowry, Duran 89] M. Lowry, R. Duran: Knowledge-Based Software Engineering. In: A. Barry, P.R. Cohen, E.A. Feigenbaum (eds.): *The Handbook of Artificial Intelligence, Vol. IV,* Addison Wesley 1989, 214-322

[Lynch, Stark 89] N. Lynch, E. Stark: A Proof of the Kahn Principle for Input/Output Automata. *Information and Computation **82**,* 1989, 81-92

[Manna 74] Z. Manna: *Mathematical Theory of Computation*. MacGraw-Hill 1974

[McGraw 82] J.R. McGraw: The VAL Language. Description and Analysis. *ACM Transactions on Programming Languages and Systems **4**(1),* 1982, 44-82

[Milner 80] R. Milner: *A Calculus of Communicating Systems,* LNCS 92, Springer 1980

[Mosses 90] P.D. Mosses: Denotational Semantics. In: J. van Leeuwen (ed.): *Handbook of Theoretical Computer Science, Vol. B,* Elsevier 1990, 575-631

[Nückel 88] H. Nückel: Eine Zeigerimplementierung von Graphreduktion für eine Datenflußsprache. Diplomarbeit, Universität Passau, 1988

[Olderog 91] E.-R. Olderog: Towards a Design Calculus for Communicating Programs. In: J.C.M. Baeten, J.F. Groote (eds.): *CONCUR '91, 2nd International Conference on Concurrency Theory,* LNCS 527, Springer 1991, 61-77

[Panangaden, Stark 88] P. Panangaden, E.W. Stark: Computations, Residuals, and the Power of Indeterminacy. In: T. Lepistö, A.K. Salomaa (eds.): *Automata, Languages and Programming,* LNCS 317, Springer 1988, 439-454

[Park 82] D. Park: The Fairness Problem and Nondeterministic Computing Networks. In: *Proc. 4th Advanced Course on Theoretical Computer Science,* Mathematisch Centrum (CWI), Amsterdam, 1982

[Partsch 90] H.A. Partsch: *Specification and Transformation of Programs.* Texts and Monographs in Computer Science, Springer 1990

[Partsch, Steinbrüggen 83] H.A. Partsch, R. Steinbrüggen: Program Transformation Systems. *Computing Surveys **15**(3),* 1983, 199-236

[Paterson, Hewitt 70] M.S. Paterson, C.E. Hewitt: Comparative Schematology. In: *Record of the Projekt MAC Conference on Concurrent Systems and Parallel Computation (ACM)*, 1970, 119-128

[Pepper 79] P. Pepper: A Study on Transformational Semantics. Dissertation am Fachbereich Mathematik der Technischen Universität München, 1979

[Pepper 87] P. Pepper: A Simple Calculus for Program Transformation (Inclusive of Induction). *Science of Computer Programming **9***, 1987, 221-262

[Pepper 90] P. Pepper: Development of Communication Protocols by Transforming Temporal Specifications. unveröffentlichtes Manuskript, Technische Universität Berlin, 1990

[Peyton Jones 87] S.L. Peyton Jones: *The Implementation of Functional Programming Languages.* Prentice Hall 1987

[Peyton Jones 89] S.L. Peyton Jones: Parallel Implementation of Functional Programming Languages. *The Computer Journal **32**(2),* 1989, 175-186

[Plotkin 83] G. Plotkin: An Operational Semantics of CSP. In: D. Bjøner (ed.): Proc. *IFIP TC 2 Working Conference on Formal Description of Programming Concepts II,* 1983, 199-223

[Reisig 85] W. Reisig: *Petri Nets. An Introduction.* EATCS Monograph 4, Springer 1985

[Russell 89] J.R. Russell: Full Abstraction for Nondeterministic Dataflow Networks. In: *Proc 30th Annual IEEE Symposium on Foundations of Computer Science,* IEEE Computer Science Press 1989, 170-175

[Sharp 87] J.A. Sharp: *An Introduction to Distributed and Parallel Processing.* Blackwell Scientific Publications 1987

[Stefãnescu 87] G. Stefãnescu: On Flowchart Theories, Part I. *Journal of Computer and Systems Sciences **35**,* 1987, 163-191

[Streicher 87] T. Streicher: A Verification Method for Finite Dataflow Networks with Constraints Applied to the Verification of the Alternating Bit Protocol. Technische Berichte der Fakultät für Mathematik und Informatik, Universität Passau, MIP 8706, 1987

[Strong 71] H.R. Strong: Translating Recursion Equations into Flow Charts. *Journal of Computer and Systems Sciences **5**(3),* 1971, 254-285

[Turner 90] D.A. Turner: An Approach to Functional Operating Systems. In: D.A. Turner (ed.): *Research Topics in Functional Programming,* Addison Wesley 1990

[Wadge, Ashcroft 85] W.W. Wadge, E.A. Ashcroft: *Lucid, the Dataflow Programming Language.* Academic Press 1985

[Walker, Strong 73] S.A. Walker, H.R. Strong: Characterizations of Flowchartable Recursions. *Journal of Computer and Systems Sciences **7**,* 1973, 404-447

[Weber 90] R. Weber: Distributed Systems. In: [Krieg-Brückner 90] Vol. I, Section 2.3, 1990