

IM 8447 2. Ex

Institut für Informatik

TRANSFORMATION PARALLEL ABLAUFENDER PROGRAMME

Manfred Broy

geb. am 10.8.49 in Landsberg am Lech

vollständiger Abdruck der von der
Fakultät für Mathematik
der Technischen Universität München
zur Erlangung des Grades eines

Dr. rer. nat.

genehmigten Dissertation

Vorsitzender: Prof. Dr. M. Paul

1. Prüfer: Prof. Dr. Dr. h.c. F.L. Bauer
2. Prüfer: Prof. Dr. G. Seegmüller
3. Prüfer: Prof. Dr. R. Bulirsch

Die Dissertation wurde am

21. 11. 1979

bei der Technischen Universität München eingereicht
und durch die Fakultät für Mathematik am

29.1.1980

angenommen.

Tag der Promotion:

4.2.1980

Vorbemerkung

Prof. F.L. Bauer und Prof. G. Seegmüller verdanke ich zahlreiche, wertvolle Anregungen und Ratschläge. Eine Reihe von Diskussionen mit meinen Kollegen vom SFB 49 und Projekt CIP, insbesondere P. Pepper und M. Wirsing, die so freundlich waren, eine vorläufige Fassung dieser Arbeit durchzusehen, haben maßgebend zur Verbesserung der Darstellung beigetragen. Ihnen allen möchte ich auf das herzlichste danken.



Einführende Zusammenfassung

Entwickelt man Programme durch schrittweisen Übergang von abstrakten zu konkreteren Formulierungen und damit durch allmähliche Ergänzung einer Spezifikation zu einem vollständigen Programm - einer effizienten Version eines Algorithmus, so stehen die einzelnen Glieder in der Kette der Entwicklung in bestimmten Relationen zueinander. Meistens sind sie im Sinn der mathematischen Semantik ("funktional") äquivalent. Die Definition der funktionalen Äquivalenz genügt im allgemeinen um die mathematische Semantik einer Programmiersprache zu charakterisieren.

Wie für die mathematische Semantik lassen sich auch Klassen von Kongruenzrelationen auf Programm(ausdrück)en definieren, die verschiedene Begriffe von "operativ äquivalent" formal faßbar machen und bedeutend feiner sind, als die durch die mathematische Semantik erzeugte Kongruenzrelation. Die verschiedenen Versionen eines Programms in einer Programmentwicklung sind natürlich im allgemeinen operativ keineswegs äquivalent.

Eine zusätzliche Dimension erhält die Begriffswelt operativer Semantik, wenn auch parallel ablaufende Programm(teil)e betrachtet werden. In allgemeinen Programmsystemen mit parallel ablaufenden Teilen treten charakteristische Phänomene auf, die formale Aussagen über die operative und mathematische Semantik solcher Programme stark erschweren:

- Die Kommunikation zwischen parallel ablaufenden Prozessen bewirkt logische Abhängigkeiten zwischen Programmteilen, die in Aufschreibung (und Ablauf) stark von einander entfernt sind.
- Zusätzlich tritt Nichtdeterminismus als Abstraktion der nicht determinierten zeitlichen Verzahnung der Kommunikationsaktionen parallel ablaufender Prozesse auf.

Gerade hierbei lassen sich Kongruenzrelationen und definierende Transformationen erfolgreich für die Definition der Semantik einsetzen.

In vielen Ansätzen zur Behandlung von Parallelität in Programmiersprachen sind die obigen Phänomene zusätzlich überlagert durch Probleme der Synchronisation, die insbesondere einen breiten Raum einnehmen, wenn über "gemeinsame" Programmvariable kommuniziert wird. Häufig werden dann zur Definition der Semantik quasiparallele Vorstellungen verwendet. Dabei werden gewisse "atomare" Aktionen (vgl. /Plotkin 79/) in beliebiger Weise gemischt. Dieses Zurückgehen auf eine Sequentialisierung kann jedoch nicht

immer voll befriedigen, weil dadurch häufig weder eine ausreichende Begriffsbildung "echter" Parallelität, noch eine von möglichen Implementierungen her angemessene Beschreibung erreicht wird. Andere Ansätze trennen parallele Programme völlig von üblichen Sprachelementen sequentieller Programmiersprachen (vgl. Petrinetze in /Petri 62/).

Im Gegensatz dazu wird in dieser Arbeit ein Versuch unternommen, von einer im Prinzip funktionalen Programmiersprache ausgehend Sprachelemente für die parallele Programmierung herzuleiten und diese damit mit klassischen sequentiellen Sprachelementen zu integrieren.

Dadurch sind die Ziele der vorliegenden Arbeit bestimmt:

- Abklärung und Präzisierung von Begriffen aus dem Bereich der parallelen Programmierung,
- Aufzeigen grundlegender Konzepte, auf die Sprachelemente für die parallele Programmierung abgestützt werden können,
- Herleitung adäquater Sprachelemente für die parallele Programmierung in einer Breitbandsprache durch den Ausbau applikativer und prozeduraler Sprachstile,
- Entwicklung formaler Methoden für die Beschreibung der Semantik solcher Sprachelemente, insbesondere
- Einsatz von definierenden Transformationen für die formale Definition der Semantik und die Entwicklung parallel ablaufender Programme,
- Erarbeitung von Ansätzen für eine Programmiermethodologie zur formal abgesicherten Entwicklung (und damit zur Verifikation) paralleler Programme,
- Untersuchung gewisser Phänomene und Begriffe im Bereich der Theorie paralleler Programme wie Fairness, unbeschränkter Nichtdeterminismus, nichtterminierende Programme, Kommunikation etc.

Die Arbeit gliedert sich in drei Teile entsprechend der parallelen Programmierung der applikativen Ebene, der prozeduralen Ebene und einiger breiter ausgeführter Beispiele für die Entwicklung paralleler Programme. Jeder Teil beginnt mit einer ausführlichen, informellen Übersicht über die in ihm enthaltenen Ideen und Konzepte, um so die Zusammenhänge klar herauszuarbeiten.

Nach einer knappen Zusammenstellung der wichtigsten notationellen Grundlagen und formalen Begriffe werden im ersten Kapitel Abläufe (vgl. "Traces" in /Hoare 78a/) als algebraische Abstraktion operativer Semantik formal definiert. Darauf aufbauend lassen sich Begriffe wie Aktionen und Prozesse beschreiben. Zur Formalisierung der Begriffe Parallelität und Sequentialität werden Ablaufordnungen auf den Aktionenmengen eingeführt. Verschiedene "Effizienzmaße" gestatten die Bewertung von Programmen nach unterschiedlichen Gesichtspunkten, wie Prozessoren-, Rechen- und Zeitaufwand bei paralleler bzw. sequentieller Auswertung.

Im zweiten Kapitel werden wohlbekannte Sprachelemente der applikativen Ebene auf Möglichkeiten paralleler Verarbeitung untersucht. Geringfügige Erweiterungen der Notation der Kollektivdeklaration zur Paralleldeklaration bilden die Basis für eine nichtdeterministische Berechnungsregel ("Call-in-Parallel", nicht zu verwechseln mit den "Parallel-Substitution-Rules" in /Manna et al. 73/) für rekursive Funktionen, die durch Transformationsregeln beschrieben wird. Diese Regel kann als operative Semantik der im allgemeinen durch Flußdiagramme beschriebenen Programme der "Data Flow Languages" verstanden werden.

Im dritten Kapitel werden gewöhnliche Objektdeklarationen zu einem Sprachelement für eine (eingeschränkte) Kommunikation zwischen parallel ablaufenden, applikativen Programmen, zu sogenannten Resultatbezeichnungen, erweitert. Durch abstrakte Typen und Komprehension über Mengen von Abläufen formal beschriebene "Kommunikationstabellen" gestatten darüber hinaus eine sehr allgemeine Kommunikation zwischen parallel ablaufenden, applikativen Programmen. Dies wird am Beispiel der fünf dinierenden Philosophen demonstriert. Die Definition eines nichtstrikten (ω -absorbierenden) Auswahloperators erlaubt die Formulierung disjunktiver ("multipler") Wartezustände und eines Schemas für die parallele Auswertung bewachter Ausdrücke. Daneben ergibt sich eine enge Verwandtschaft mit nichtstrikten, symmetrischen, logischen Ausdrücken wie der nichtstrikten Disjunktion oder der nichtstrikten Konjunktion.

Im vierten Kapitel werden schließlich die Begriffe Ablauf, Aktion, Prozeß und Ablaufordnung auf die Variablenebene übertragen. Ausgehend von konfliktfreien, kollektiven Zuweisungen werden Parallelanweisungen

eingeführt. Die Semantik bewachter, kritischer Bereiche (vgl. /Hoare 71/) wird durch definierende Transformationen auf bereits vorhandene, sequentielle Sprachelemente abgestützt. Dabei wird das Sprachelement der einfach bewachten, kritischen Bereiche schrittweise verallgemeinert, bis auch Probleme disjunktiven Wartens und geschachtelte Parallelität behandelt werden können. Transformationsregeln und "Transformationsinduktion" ergeben eine Verifikationsmethode, die im Gegensatz zur Methode nach Owicki (vgl. /Owicki 75/) keine Einführung von Hilfsvariablen benötigt. Durch die Transformationsregeln wird ein Einstieg in die formale Entwicklung paralleler Programme angeboten.

Im fünften Kapitel werden spezielle Systeme paralleler Programme behandelt. So werden als Beispiel Ströme (vgl. "Streams" in /Landin 65/) als Sprachelemente für eine Kommunikation definiert. Damit nähern wir uns klassischen Fragestellungen der Systemprogrammierung. Es werden Begriffe wie lokaler, globaler und unbeschränkter Nichtdeterminismus und Fairnessannahmen untersucht. Insbesondere das Problem der Verträglichkeit allgemeiner Fairnessbedingungen mit der Stetigkeit entsprechender Sprachelemente wird behandelt. Zur Beschreibung unendlich lang laufender Systeme wird nichtterminierenden Prozedur(aufruf)en über die funktionale Gleichsetzung mit undefiniert hinaus eine Semantik durch unendliche Abläufe zugeordnet. Dadurch können beispielsweise Petrinetze in der Programmiersprache durch Programme repräsentiert werden. Es wird gezeigt, daß auch solche Programme durch Transformationen aus sequentiellen Programmen entwickelt werden können. Abschließend werden Übergänge von applikativen auf prozedurale Programme unter dem Gesichtspunkt paralleler Verarbeitung untersucht.

Im sechsten Kapitel werden ausgehend von durch nichtlineare Rekursion definierten Funktionen parallele Programme mit Hilfe der eingeführten Methoden entwickelt. Das Beispiel des Interpolationsalgorithmus wird stellvertretend für die Klasse der durch Wertverlaufsrekursion definierten Funktionen gewählt und führt auf ein System paralleler Programme mit gerichteten Kommunikationsbeziehungen. Ein allgemeineres Beispiel stellen Algorithmen für das Durchlaufen von Graphen dar. Sie ergeben Systeme paralleler Programme, die wechselseitig miteinander kommunizieren. Die Arbeit schließt mit einer knappen Übersicht über die Entwicklung der Theorie paralleler Programme mit Schwerpunkten auf der formalen Beschreibung, Verifikationsmethoden und einer Erörterung offener Fragestellungen.

INHALTSVERZEICHNIS

0	Notation, Konventionen, semantische Grundlagen	10
	<u>Teil I : Paralleler Ablauf applikativer Programme</u>	16
	Informelle Übersicht	
1	Operative Semantik, Effizienzbegriffe	20
1.1	Abläufe applikativer Programme	20
1.1.1	Definition von Abläufen	20
1.1.2	Ablaufäquivalenz	24
1.1.3	Reduzierte Abläufe	25
1.2	Aktionen, Prozesse, Ablaufordnungen, Effizienzmaße	26
1.2.1	Aktionen	26
1.2.2	Partielle Ordnungsrelationen auf Aktionenmengen	29
1.2.3	Parallele und sequentielle Aktionen	32
1.2.4	Effizienzmaße	33
2	Parallele Abläufe der applikativen Ebene	35
2.1	Konzepte der applikativen Ebene	35
2.1.1	Funktionsapplikation und parallele Auswertung	36
2.1.2	Gleiche Teilausdrücke und parallele Auswertung	37
2.2	Parallele Berechnungen	39
2.2.1	Die Paralleldeklaration	40
2.2.2	Transformationsregeln zur Berechnung rekursiver Funktionen	42
2.2.3	Eine nichtdeterministische Berechnungsregel	43
2.2.4	Ablaufordnungen und Effizienzmaße für die Paralleldeklaration	49
2.2.5	Zyklisches Warten und Verklemmung	51
2.3	Parallele Berechnung und Programmentwicklung	53
2.3.1	Sequentielle, kollektive und parallele Deklaration	53
2.3.2	Parallele Berechnung und bewachte Ausdrücke	55
2.3.3	Teilberechnungen	56
3	Kommunikation zwischen parallel ablaufenden Programmen	57
3.1	Resultatbezeichnungen	58
3.1.1	Syntax und informelle Beschreibung von Resultatbezeichnungen	59
3.1.2	Formale Definition der Semantik von Resultatbezeichnungen	61
3.1.3	Transformationsregeln für Resultatbezeichnungen	65
3.1.4	Ableitung des Beispiels "modiv"	68
3.2	Kommunikationstabellen	69
3.2.1	Informelle Beschreibung von Kommunikationstabellen	70
3.2.2	Formale Definition der Semantik von Kommunikationstabellen	73
3.2.3	Programmentwicklung mit Kommunikationstabellen	75
3.2.4	Eine Regel für gerichtete Kommunikation	77
3.3	Nichtstrikte Auswahl und disjunktives Warten	78
3.3.1	Der nichtstrikte Auswahloperator	78
3.3.2	Abläufe für den Auswahloperator	80
3.3.3	Disjunktives Warten	80
3.3.4	Parallele Auswertung bewachter Ausdrücke	82
3.4	Programmbeispiele	83
3.4.1	Das Erzeuger/Verbraucher Problem	83
3.4.2	Die fünf dinierenden Philosophen	84

Teil II : <u>Paralleler Ablauf prozeduraler Programme</u>	88
Informelle Übersicht	
4 Grundelemente prozeduraler, paralleler Programme	92
4.1 Abläufe der prozeduralen Ebene	93
4.1.1 Abläufe für Anweisungen	95
4.1.2 Reduzierte Abläufe von Anweisungen	96
4.1.3 Aktionen, Prozesse, Ablaufordnungen prozeduraler Programme	96
4.2 Parallele Abläufe der prozeduralen Ebene	97
4.2.1 Parallele Ausführung von Zuweisungen	97
4.2.2 Die Bernsteinbedingung	99
4.2.3 Parallele Ausführung von nicht konfliktfreien Anweisungen	99
4.3 Transformationssemantik für bewachte, kritische Bereiche	101
4.3.1 Bewachte, kritische Bereiche	101
4.3.2 Definierende Transformationsregeln für bewachte, kritische Bereiche	102
4.3.3 Beweisen durch Transformationen	104
4.3.4 Abgeleitete Transformationsregeln	106
4.3.5 Erweiterungen bewachter, kritischer Bereiche	107
5 Spezielle Systeme paralleler Programme	109
5.1.1 Resultatbezeichnungen und parallele Anweisungen	109
5.1.2 Kommunikationsströme	110
5.1.3 Anwendungen und Beispiele	111
5.1.4 Kommunikation über Schlüssel	113
5.2 Unbeschränkter Nichtdeterminismus	115
5.2.1 Beschränkter und unbeschränkter Nichtdeterminismus	115
5.2.2 Dijkstra's Kalkül und unbeschränkter Nichtdeterminismus	116
5.2.3 Lokaler und globaler Nichtdeterminismus	117
5.3 Koordination und Fairness	118
5.3.1 Kritische Phasen, gegenseitiger Ausschluß, Synchronisation	118
5.3.2 Fairness	120
5.3.3 Fixpunktsemantik, Fairness und nichtstetige Funktionale	121
5.3.4 Bemerkungen zur Stetigkeit und Fairnessbedingungen	123
5.4 Unendliche Abläufe und nichtterminierende Prozeduren	125
5.4.1 Bedeutung unendlicher Abläufe	125
5.4.2 Programmentwicklung für nichtterminierende Systeme	128
5.4.3 Petri-Netze als nichtterminierende parallele Programme	130
5.5 Von applikativen zu prozeduralen, parallelen Programme	132
5.5.1 Zur parallelen Auswertung von Anweisungen	132
5.5.2 Zur Erhaltung der Parallelität bei Übergängen	134

Teil III : <u>Beispiele, abschließende Bemerkungen</u>	136
Informelle Übersicht	
6 Programmentwicklung durch Transformationen für parallele Programme	137
6.1 Gerichtete Kommunikation - Parallelisierung	137
6.1.1 Nichtlineare Rekursion	137
6.1.2 Das Schema nach Aitken-Neville	138
6.1.3 Parallelisierung	138
6.2 Wechselseitige Kommunikation	141
6.2.1 Durchlaufen von Graphen	141
6.2.2 Paralleles Durchlaufen von Graphen	143
6.2.3 Paralleles Durchlaufen mit fest vorgegebener Prozessorzahl	144
7 Historische Entwicklungen und abschließende Bemerkungen	146
7.1 Historischer Überblick	146
7.1.1 Sprachelemente für die parallele Programmierung	146
7.1.2 Formale Beschreibung der Semantik	148
7.1.3 Verifikationsmethoden	149
7.2 Abschließende Bemerkungen und offene Probleme	151
8 Literatur	152

0 Notation, Konventionen, semantische Grundlagen

Wir verwenden die Notation des ALGOL-Dialekts der Breitbandsprache CIP-L (vgl. /Bauer et al.77, 78a, 78b/, für die Begriffswelt vgl. auch /Bauer, Wössner 79/). Die Struktur der Breitbandsprache läßt sich grob durch folgende Sprachstile charakterisieren:

- Deskriptiver Stil: Quantifizierung, unendliche Auswahl, Kennzeichnung und Mengenkompensation.
- Applikativer (algorithmischer) Stil: Funktionsabstraktion, Rekursion, bewachte Ausdrücke, Konstantenvereinbarung und Parameterunterdrückung.
- Prozeduraler (algorithmischer) Stil: Programmvariablen, Zuweisung, bewachte Anweisungen, Prozeduren, Wiederholungsanweisungen und Sprünge.

Dabei wollen wir uns in dieser Arbeit den folgenden Kontextbeschränkungen unterwerfen:

- Ausdrücke sind frei von Seiteneffekten: Im Inneren eines Ausdrucks oder einer in einem Ausdruck aufgerufenen Prozedur treten keinerlei Zuweisungen an global zum Ausdruck vereinbarte Variablen oder Sprunganweisungen auf global zum Ausdruck vereinbarte Marken auf.
- Keine Mehrfachdefinitionen von Bezeichnungen: Jede Bezeichnung wird innerhalb eines Bereichs höchstens einmal vereinbart. Aus Gründen der Vereinfachung soll in dieser Arbeit sogar allgemein eine verschärfte Bedingung gelten. In einem Programm soll jede Bezeichnung höchstens einmal vereinbart werden.

Durch diese verschärfte Bedingung wird die "Überlagerung" von Bezeichnungen ausgeschlossen und damit die Beschreibung der in den folgenden Kapiteln behandelten Sprachelemente erheblich vereinfacht. Man beachte, daß diese Bedingung keine Beschränkung der Allgemeinheit bedeutet, da im Falle von Bezeichnungskonflikten eine Umbenennung stets möglich ist, wenn wir über eine unbeschränkte Menge von Bezeichnungen verfügen können.

Im folgenden werden verschiedene Äquivalenzrelationen auf Programm(term)en erklärt, da Programme sowohl als syntaktische Objekte, als Terme, als auch als Repräsentanten semantischer Objekte betrachtet werden. Zusätzlich wird die Gleichheit als Operation in Programmen verwendet. Um Mißverständnissen vorzubeugen, vereinbaren wir, daß das Symbol " \cong " die syntaktische Gleichheit zweier Programme prüft.

Ein weiteres syntaktisches Prädikat "occurs" prüft das Auftreten einer freien Bezeichnung in einem Programm. So trifft $\text{occurs}(x \text{ in } P)$ genau dann zu, wenn die Bezeichnung x im Programm P frei auftritt. Häufig verwenden wir die Abkürzung:

$$\text{occurs}(x_1, \dots, x_n \text{ in } P_1, \dots, P_m) = \text{def} \bigvee_{1 \leq i \leq n} \bigvee_{1 \leq j \leq m} \text{occurs}(x_i \text{ in } P_j).$$

Da gerade bei der parallelen Ausführung von Programmen Nichtdeterminismus eine zentrale Rolle spielt, stützen wir uns in dieser Arbeit grundlegend auf die formale Definition der Semantik nichtdeterministischer Sprachelemente in /Broy et al. 78b/. Dabei setzen wir voraus, daß jeder Art (bezeichnung) \underline{m} eine Menge $U(\underline{m})$ von semantischen Objekten zugeordnet ist. Die für die Arten als Bestandteile von Rechenstrukturen verfügbaren Funktionen nennen wir primitiv.

Ein CIP-L-Term E der syntaktischen Einheit "Ausdruck" heißt Ausdruck in x_1, \dots, x_n , wenn E höchstens x_1, \dots, x_n als freie Bezeichnungen enthält. Die Menge solcher Ausdrücke bezeichnen wir mit $\text{EXP}[\underline{m}_1 x_1, \dots, \underline{m}_n x_n]$, wobei die \underline{m}_i die Arten der freien Bezeichnungen kennzeichnen, und für $n=0$ mit EXP . Für jeden Ausdruck $E \in \text{EXP}[\underline{m}_1 x_1, \dots, \underline{m}_n x_n]$ der Art \underline{r} ist dann

$$(\underline{m}_1 x_1, \dots, \underline{m}_n x_n)_{\underline{r}}: E$$

ein Ausdruck für eine nichtprimitive Funktion.

Analog zu /Broy et al. 78b/ sei die Semantik eines Ausdrucks $E \in \text{EXP}$ durch seine Breite $B(E) \subseteq U(\underline{m})$, wobei \underline{m} die Art des Ausdrucks E bezeichne, und seine Definiertheit $d(E) \in \{\text{TRUE}, \text{FALSE}\}$ gegeben. Die Breite $B(E)$ gibt die Menge der möglichen definierten

Resultate von Auswertungen von E wieder, die Definiertheit gibt an, ob jede Auswertung mit einem definierten Resultat terminiert. Dementsprechend gilt für den "undefinierten"

Ausdruck error:

$$B(\text{error}) = \emptyset, \quad d(\text{error}) = \text{FALSE}.$$

Ein Ausdruck E heißt determiniert, abgekürzt "determinate(E)", wenn:

$$(|B(E)| = 1 \wedge d(E) = \text{TRUE}) \vee (B(E) = \emptyset \wedge d(E) = \text{FALSE})$$

Für determinierte Ausdrücke verwenden wir aus Abkürzungsgründen auch die Semantikfunktion V mit

$$V(E) =_{\text{def}} \begin{cases} x & \text{falls } B(E) = \{x\} \text{ und } d(E) = \text{TRUE}; \\ \emptyset & \text{falls } B(E) = \emptyset \text{ und } d(E) = \text{FALSE}. \end{cases}$$

Die Definitionen der Breite und Definiertheit der einzelnen Sprachelemente findet sich in /Broy et al. 78b/ und soll hier nicht wiederholt werden. Auf der Menge der Ausdrücke EXP verwenden wir zwei wichtige Relationen, die Egli-Milner Ordnung \sqsubseteq und die Implementierungs- oder Abkömmlingsordnung \sqsubseteq_I . Für $E_1, E_2 \in \text{EXP}$ gilt:

$$E_1 \sqsubseteq E_2 \iff_{\text{def}} B(E_1) \subseteq B(E_2) \wedge (d(E_1) \Rightarrow (d(E_2) \wedge B(E_1) = B(E_2)));$$

$$E_1 \sqsubseteq_I E_2 \iff_{\text{def}} B(E_1) \subseteq B(E_2) \wedge (d(E_2) \Rightarrow d(E_1)).$$

Auf der durch B und d auf den Ausdrücken erzeugten Quotientenstruktur sind beide Relationen partielle Ordnungen. Für die Egli-Milner Ordnung gilt:

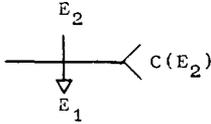
- error repräsentiert das kleinste Element,
- jede \sqsubseteq -monotone Folge besitzt eine kleinste obere Schranke,
- alle Sprachelemente der Breitbandsprache sind \sqsubseteq -monoton,
- alle endlichen Sprachelemente sind \sqsubseteq -stetig,
- alle Ausdrücke $E \in \text{EXP}$ mit $d(E) = \text{TRUE}$ sind maximal.

Damit ist die Egli-Milner Ordnung für die Definition nicht-deterministischer, rekursiver Funktionen als schwächste Fixpunkte stetiger Funktionale geeignet.

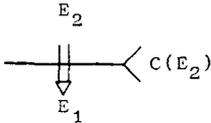
Für die Implementierungsordnung gilt:

- determinierte Ausdrücke sind minimal,
- alle Elemente der Breitbandsprache sind \subseteq_I -monoton.

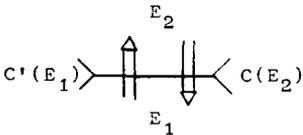
Damit ist die \subseteq_I -Ordnung für die Entwicklung nichtdeterministischer Programme geeignet. Für Ausdrücke E_1, E_2 und ein Prädikat C schreiben wir häufig statt $C(E_2) \Rightarrow E_1 \subseteq_I E_2$:



und statt $C(E_2) \Rightarrow (E_1 \subseteq_I E_2 \wedge E_2 \subseteq_I E_1)$:¹⁾



Gilt zusätzlich $C'(E_1) \Rightarrow (E_1 \subseteq_I E_2 \wedge E_2 \subseteq_I E_1)$, so schreiben wir:



Im Gegensatz zu Ausdrücken werden wir Anweisungen nicht durch die Angabe einer Semantikfunktion eine "denotationelle Semantik" geben, sondern nur im Kontext von Ausdrücken eine Bedeutung zuzuordnen. Eine Anweisung S heißt Anweisung in x_1, \dots, x_n , wenn in S höchstens x_1, \dots, x_n als freie Bezeichnungen auftreten. Die Menge solcher Anweisungen bezeichnen wir mit $STA[\underline{m}_1 x_1, \dots, \underline{m}_n x_n]$, wobei die \underline{m}_i die Arten der freien Bezeichnungen beschreiben. Damit ist für jede Anweisung $S \in STA[\underline{m}_1 x_1, \dots, \underline{m}_n x_n]$

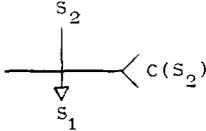
$$(\underline{m}_1 x_1, \dots, \underline{m}_n x_n): S$$

ein Ausdruck für eine Prozedur. Für Anweisungen

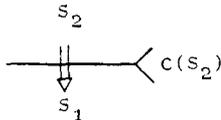
$S_1, S_2 \in STA[\underline{m}_1 x_1, \dots, \underline{m}_n x_n]$ sagen wir " S_1 implementiert S_2 "

¹⁾ Die Unsymmetrie dieser Schreibweise ergibt sich aus dem Umstand, daß die Kontextkorrektheit von E_1 von $C(E_2)$ abhängen kann.

unter der Bedingung $C(S_2)$ ", wenn für jeden Kontext K gilt:
 $C(S_2) \wedge K[S_2] \in \text{EXP} \Rightarrow (K[S_1] \in \text{EXP} \wedge K[S_1] \vDash_I K[S_2])$ und
 schreiben:



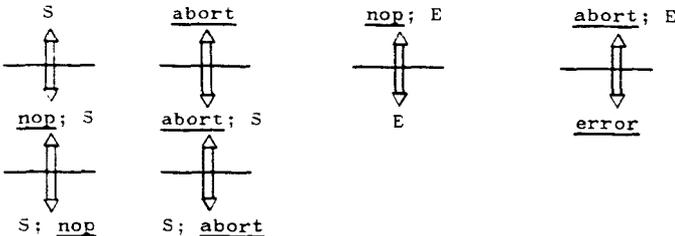
Gilt $C(S_2) \wedge K[S_2] \in \text{EXP} \Rightarrow (K[S_1] \in \text{EXP} \wedge K[S_1] \vDash_I K[S_2] \wedge K[S_2] \vDash_I K[S_1])$, so sagen wir " S_1 ist semantisch äquivalent zu S_2
unter der Bedingung $C(S_2)$ " und schreiben:



Führt man nun jeden Ausdruck, der Anweisungen enthält, auf einen Ausdruck ohne Anweisungen zurück (vgl. /Pepper 79/), so wird auf Ausdrücken mit Anweisungen eine Semantik induziert und auf der Menge der Anweisungen eine Äquivalenzrelation. Jede Anweisung $S \in \text{STA}[\text{var } m_1 v_1, \dots, \text{var } m_n v_n]$ kann dann semantisch durch die Funktion

$$\begin{aligned}
 & (\underline{m}_1 x_1, \dots, \underline{m}_n x_n) (\underline{m}_1, \dots, \underline{m}_n) : \\
 & (\text{var } \underline{m}_1 v_1, \dots, \text{var } \underline{m}_n v_n) := (x_1, \dots, x_n); \\
 & S; \\
 & (v_1, \dots, v_n)
 \end{aligned}$$

charakterisiert werden. Betrachtet man die definierenden Transformationsregeln für die leere Anweisung nop und die nichtterminierende Anweisung abort:



dann entspricht nop der Identitätsfunktion und abort der nirgends terminierenden Funktion Ω .

Um Beweise über Programmausdrücke zu führen, verwenden wir häufig Induktion.

Lemma: Sei funct $f = (\underline{m}_1 x_1, \dots, \underline{m}_n x_n)_{\underline{r}}$: E und gelte

$$\begin{array}{c} f(x_1, \dots, x_n) \\ \hline \Downarrow \\ T[f, x_1, \dots, x_n] \end{array}$$

dann gilt $f' \sqsubseteq^* f$ mit funct $f' = (\underline{m}_1 x_1, \dots, \underline{m}_n x_n)_{\underline{r}}$: $T[f', x_1, \dots, x_n]$.

Dabei sei für jede Ordnung \sqsubseteq die Ordnung \sqsubseteq^* für Funktionen f, g der Art funct $(\underline{m}_1, \dots, \underline{m}_n)_{\underline{r}}$ definiert durch

$$f \sqsubseteq^* g \iff \text{def } \forall (x_1, \dots, x_n) \in U(\underline{m}_1) \times \dots \times U(\underline{m}_n): \\ f(x_1, \dots, x_n) \sqsubseteq g(x_1, \dots, x_n).$$

Beweis: Durch "Computational Induction"

Sei funct $f_0 = (\underline{m}_1 x_1, \dots, \underline{m}_n x_n)_{\underline{r}}$: error,

$$\text{funct } f_{i+1} = (\underline{m}_1 x_1, \dots, \underline{m}_n x_n)_{\underline{r}}: T[f_i, x_1, \dots, x_n].$$

Es gilt $f_0 \sqsubseteq^* f$. Sei nun $f_i \sqsubseteq^* f$. Für alle

$(y_1, \dots, y_n) \in U(\underline{m}_1) \times \dots \times U(\underline{m}_n)$ gilt:

$$f(y_1, \dots, y_n) = T[f, y_1, \dots, y_n] \sqsupseteq T[f_i, y_1, \dots, y_n] = f_{i+1}(y_1, \dots, y_n),$$

und damit gilt $f_{i+1} \sqsubseteq^* f$, also nach Induktion $f' = \text{lub}\{f_i\} \sqsubseteq^* f$. \square

Dieses Lemma zeigt, daß FOLD eine nur partiell korrekte Transformationsregel ist (vgl. /Kott 78/). Gilt jedoch für

$$\begin{array}{l} \text{zwei Funktionen } \text{funct } f = (\underline{m}_1 x_1, \dots, \underline{m}_n x_n)_{\underline{r}}: T[f, x_1, \dots, x_n], \\ \text{funct } f' = (\underline{m}_1 x_1, \dots, \underline{m}_n x_n)_{\underline{r}}: T[f', x_1, \dots, x_n] \end{array}$$

sowohl: , als auch:

$$\begin{array}{cc} f(x_1, \dots, x_n) & f'(x_1, \dots, x_n) \\ \hline \Downarrow & \Downarrow \\ T[f, x_1, \dots, x_n] & T[f', x_1, \dots, x_n] \end{array}$$

Dann gilt $f \sqsubseteq^* f'$ und $f' \sqsubseteq^* f$, also $f \equiv^* f'$. Wir sagen f und f' sind nach Transformationsinduktion mathematisch äquivalent.

Teil I

Paralleler Ablauf applikativer Programme

Parallelität wird im allgemeinen durch Sprachelemente beschrieben, die Bestandteil der prozeduralen Ebene sind. Jedoch lassen sich Parallelität und verwandte Begriffsbildungen ebenso für applikative Sprachen entwickeln. Da sich Parallelität grundsätzlich auf die Art und Weise bezieht, in der Programme verarbeitet werden sollen, erscheint für eine formale Begriffsbildung die Formalisierung operativer Semantik unumgänglich. Im Gegensatz zu bekannten Ansätzen zur Beschreibung operativer Semantik durch die Definition mathematischer Maschinenmodelle soll hier ein algebraischer, sprachorientierter Begriff operativer Semantik gebildet werden. Dazu werden Programmausdrücken Mengen von Abläufen zugeordnet. Unter Abläufen versteht man dabei selbst wieder Programmterme, die jedoch nur noch Objektdeklarationen, Applikationen primitiver Funktionen und einfach bedingte Ausdrücke mit genau einer Alternative enthalten. Damit bilden Abläufe eine Teilsprache, in der operative Äquivalenz und syntaktische Gleichheit - (gegebenenfalls) modulo der Umbenennung gebundener Bezeichnungen und der Permutierung kollektiver Deklarationen - übereinstimmen (gewissermaßen eine "Normalformdarstellung").

Darüberhinaus ergibt sich für Abläufe in natürlicher Weise eine Eins-zu-Eins-Korrespondenz zwischen der Menge der Teilausdrücke eines Ablaufs und der Menge der bei Ausführung des Programms anfallenden Aktionen, da jeder Unterausdruck bei der Auswertung des Programms genau einmal ausgewertet werden muß. Damit erhält man folgende Begriffsbildung: Ein Programmausdruck entspricht eine Menge von Abläufen, Abläufe setzen sich aus den Abläufen ihrer Unterausdrücke zusammen, denen jeweils gewisse Aktionen(mengen) zugeordnet sind. Durch die Relation "ist Unterausdruck" wird die Baumstruktur des abstrakten Syntaxbaums auf die Aktionenmengen übertragen.

Die zeitlichen Beziehungen zwischen zwei Aktionen a und b eines Ablaufs lassen sich durch ein Tripel transitiver Relationen "a beginnt gleichzeitig oder früher als b beginnt", "b endet gleichzeitig oder früher als a endet" und "a ist beendet, bevor b beginnt" formal beschreiben. So gelten die beiden ersten Relationen, wenn b Teilaktion der Aktion a ist. Die dritte Relation gilt, wenn a Teilaktion einer "linken" Aktion eines sequentiellen Sprachelements (z.B. "a; b") und b Teilaktion der "rechten" Aktion ist. Die innere Beziehung zwischen den drei Relationen kann durch Axiome beschrieben werden. Ein Tripel von Relationen, das diesen Axiomen entspricht, nennen wir Ablaufordnung. Ein Ablauf mit Ablaufordnung heißt Prozeß.

Bzgl. der durch die Relation "ist Unterausdruck" induzierten Baumstruktur auf der Menge der Aktionen eines Ablaufs entsprechen die transitiven Relationen der Ablaufordnung bestimmten, bekannten Durchlaufstrategien.

Um über verschiedene Effizienzbegriffe der sequentiellen bzw. parallelen Auswertung von Programmen formale Aussagen machen zu können, definieren wir den Begriff des Effizienzmaßes, das sind rationalzahlwertige, (bzgl. der Relation "ist Unterausdruck") monotone Funktionen auf Abläufen. Für unsere Zwecke genügen drei einfache Effizienzmaße:

- die Abarbeitungszeit bei unbeschränkter Anzahl von parallel arbeitenden Prozessoren,
- die benötigte Prozessoranzahl bei paralleler Verarbeitung,
- die Prozessorzeit, d.h. die Summe der Belegungszeiten der einzelnen Prozessoren.

Auf dieser Basis lassen sich die Grenzen der parallelen Verarbeitung für die zur Verfügung stehenden applikativen Sprachelemente aufzeigen. Davon ausgehend kann die applikative Sprachebene schrittweise um Sprachelemente erweitert werden, die Bedürfnissen paralleler Verarbeitung entsprechen.

So gestattet der Übergang von der Kollektivdeklaration zur Paralleldeklaration die Definition einer nichtdeterministischen Berechnungsregel zur Berechnung nichtdeterminierter, rekursiver Funktionen, die bei voller Ausschöpfung ihrer Möglichkeiten die parallele Auswertung sowohl der Argumentausdrücke, als auch des Rumpfes der Funktion gestattet. Diese Regel kann durch Transformationen formal beschrieben werden. Sie kann im Sinne der parallelen Verarbeitung als Aufspaltung eines Funktionsaufrufs mit n Parametern in $n+1$ parallele Prozesse verstanden werden (die gegebenenfalls in weitere parallele Prozesse aufgespalten werden). Die Prozesse berechnen Argumente und Rumpf der Funktion parallel. Tritt bei der Auswertung des Rumpfes eine formale Bezeichnung für ein Argument auf, dessen parallel ablaufende Berechnung noch nicht abgeschlossen ist, so wird der Abschluß dieser Berechnung abgewartet und anschließend mit dem vorliegenden Wert fortgesetzt. Der so entstehende Aufrufmechanismus "Call-in-Parallel" (nicht zu verwechseln mit Mannas "Parallel-Substitution-Rules") kann so gehalten werden, daß er Wadsworths "Call-by-Need" und Vuillemins "Delayed Evaluation" einschließt, wobei die Auswertung der Argumentausdrücke nicht verzögert wird, sondern alle Auswertungen parallel vorgenommen werden.

Da wir von einer mathematischen Semantik ausgehen, die der "Call-by-Value" Regel gehorcht und daher nur strikte, rekursive Funktionen betrachten, entspricht die parallele Berechnung der üblichen Auffassung, nach der ein System paralleler Prozesse genau dann terminiert, wenn jeder einzelne der Prozesse terminiert. Eine geringfügige Änderung der Transformationsregeln unserer Berechnungsregel ergibt eine in der "Call-by-Name" Auffassung korrekte Berechnungsregel. Der dann entstehende Begriff paralleler Berechnung entspricht einem System parallel ablaufender Prozesse mit einem ausgezeichneten Prozeß, dessen Terminierung mit der Terminierung des Gesamtsystems zusammenfällt. Noch nicht beendete andere Prozesse (zur Berechnung von Argumenten) werden bei der Terminierung dieses Hauptprozesses (zur Berechnung des Wertes des Rumpfes) abgebrochen.

Die Paralleldeklaration schließt bereits eine eingeschränkte Kommunikation über Objektbezeichnungen zwischen den parallel ablaufenden Berechnungen ein. Auch die Phänomene des Wartens und der Verklemmung treten schon auf. Jedoch findet ein "Informationstransfer" erst statt, wenn der "sendende" Prozeß seine Berechnung schon völlig abgeschlossen hat.

Eine konsequente Erweiterung dieser Kommunikationsmöglichkeit führt auf Resultatbezeichnungen. Sie gestatten die im Laufe einer Berechnung anfallende "Zwischenresultate" schon vor Abschluß der Berechnung nach außen zur Verfügung zu stellen. Sie entstehen in natürlicher Weise (durch Transformationen beschreibbar) aus Objektdeklarationen, wenn man die Deklaration der Bezeichnung und ihre unauflösbare Bindung an ein Objekt trennt. Die Semantik von Programmen, die mit solchen Resultatbezeichnungen arbeiten, läßt sich, neben definierenden Transformationsregeln, auch durch die Auszeichnung gewisser "zulässiger", konsistenter Abläufe aus einer Menge von unter gewissen Umständen in Betracht kommenden Abläufen beschreiben.

Diese Technik wird fundamental eingesetzt bei der Beschreibung der Semantik von Kommunikationstabellen. Gestattet doch eine Resultatbezeichnung nur einen Kommunikationsvorgang, so erlauben Kommunikationstabellen unbeschränkt viele Kommunikationsvorgänge, da sie von parallel ablaufenden Programmen konsekutiv aufgebaut werden können. Jeder Eintrag in der Tabelle steht dann unter einem gewissen Index zur Verfügung und kann wiederum von den parallel ablaufenden Programmen gelesen werden, ohne daß jedoch die eingetragene Information entfernt wird oder überschreibbar ist. Kommunikationstabellen stellen damit nur ein mögliches Beispiel für die Kommunikation zwischen applikativen Programmen dar.

Obwohl also keinerlei Überschreiben von Information auftritt, ist das entstehende Sprachelement mächtig genug, um z.B. das "klassische" Beispiel der fünf dinierenden Philosophen formulieren zu können.

In Kommunikationstabellen spiegelt die "zufällig" zustandegekommene Reihenfolge der Einträge die nichtdeterminierte zeitliche Verzahnung der parallel durchgeführten Eintragsaktionen wider. Damit kommt bei Kommunikationstabellen die Nichtdeterminiertheit der zeitlichen Verzahnung paralleler Aktionen erstmals voll zur Geltung.

Im Nichtdeterminismus des Auswahloperators findet sich noch eine weitere Grundlage der parallelen Programmierung. Der in unserer Breitbandsprache enthaltene Auswahloperator " \square " ist strikt, d.h. bei der Auswertung des Ausdrucks $E_1 \square E_2$ kann die Auswahlentscheidung, welcher der beiden Ausdrücke auszuwerten ist, "statisch" getroffen werden, ohne darauf zu achten, ob eventuell die Auswertung des ausgewählten Ausdrucks nicht terminiert. Ganz anders ist das bei der Betrachtung des "nichtstrikten" Auswahloperators " \boxplus ", der bei der Auswertung des Ausdrucks $E_1 \boxplus E_2$ die Wahl eines Ausdrucks mit nichtdefinierter (endlicher) Auswertung nur dann akzeptiert, wenn auch die Auswertung des anderen Ausdrucks nicht definiert ist. Damit erfordert die Auswertung von Ausdrücken der Form $E_1 \boxplus E_2$ die (quasi-)parallele Auswertung beider Ausdrücke E_1 und E_2 .

So stellt die nichtstrikte Auswahl ein Sprachelement dar, das die parallele Auswertung erfordert. Im Zusammenhang mit dem nichtstrikten Auswahloperator werden drei wichtige Konzepte diskutiert:

- Nichtstrikte symmetrische Ausdrücke, wie nichtstrikte symmetrische Disjunktion und Konjunktion,
- Disjunktives Warten bei parallelen Prozessen,
- Parallele Auswertung bewachter Ausdrücke.

Damit ist das in Verbindung mit der "Call-by-Name" Semantik erwähnte Konzept des Abbruchs gewisser parallel ablaufender Prozesse bei Terminierung des Hauptprozesses auf Systeme gleichberechtigter Prozesse erweitert worden, die genau dann definierte Ergebnisse liefern, wenn zumindest einer der Prozesse definierte Ergebnisse liefert und die anderen zumindest terminieren.

1 Operative Semantik, Effizienzbegriffe

Die operative Semantik wird algebraisch über Abläufe definiert. Darauf aufbauend lassen sich Aussagen über zeitliche Beziehungen in Abläufen und über verschiedene Effizienzbegriffe machen.

1.1 Abläufe applikativer Programme

Durch die mathematische Semantik wird jedem Ausdruck eine Menge von Objekten zugeordnet. Ausdrücke sind mathematisch äquivalent, wenn ihnen die gleichen Objektmengen zugeordnet werden. Damit können Spezifikationen (vgl. /Broy et al 78a/) und hocheffiziente Programme mathematisch äquivalent sein. Operativ sind solche Programme natürlich keineswegs äquivalent. Um eine operative Äquivalenz definieren zu können, ordnen wir jedem Ausdruck eine Menge von Abläufen zu.

1.1.1 Definition der Abläufe

Wir betrachten hier nur operative CIP-L Programme, da wir Spezifikationen keine Abläufe zuordnen wollen. Jeder operative CIP-L Ausdruck besitzt - nichtdeterministisch - eine Menge von Abläufen oder - deterministisch - genau einen Ablauf.

Definition: Ein Ablauf eines applikativen Programms ist ein CIP-L Ausdruck, der nur noch Objektdeklarationen, Tupel, bewachte Ausdrücke mit genau einer Alternative und Aufrufe primitiver Funktionen enthält (vgl. dazu /Bauer 77/).

Da primitive Funktionen als Bestandteile von Rechenstrukturen stets als determiniert vorausgesetzt werden können, sind Abläufe selbst determinierte Ausdrücke. Aus nichtterminierenden Funktionsaufrufen können unendliche Abläufe entstehen. Unendliche Abläufe kennzeichnen wir mit dem Prädikat "infinite" und ordnen ihnen stets error als mathematische Semantik zu. Es gelte also für jeden Ablauf e:

$$\text{infinite}(e) \Rightarrow (B(e) = \emptyset \wedge d(e) = \text{FALSE}).$$

Da wir auch Abläufe von Ausdrücken mit freien Bezeichnungen definieren wollen, benötigen wir den Begriff der Umgebung.

Eine Umgebung ordnet den freien Bezeichnungen eines Ausdrucks bestimmte Werte zu.

Definition: Eine Umgebung ENV ist eine Menge von Gleichungen der Form $x = e$, wobei x eine Bezeichnung und e einen Ablauf oder ein Objekt darstellen. Jede Bezeichnung tritt dabei genau einmal auf der linken Seite einer der Gleichungen auf.

Mit Hilfe einer Umgebung $ENV = \{x_1 = e_1, \dots, x_n = e_n\}$ kann ein Ausdruck $E \in \text{EXP}[x_1, \dots, x_n]$ zu einem Ausdruck $E' \in \text{EXP}$ instantiiert werden, durch

$$E' = E[e_1/x_1, \dots, e_n/x_n].$$

Die Instantiierung eines Ausdrucks durch eine Umgebung kürzen wir ab durch:

$$E' = E//ENV.$$

Sei nun E ein Ausdruck und ENV eine Umgebung. Wir definieren die Menge der Abläufe $A(E, ENV)$ von E in der Umgebung ENV durch strukturelle Rekursion:

(1) Freie Bezeichnungen

$$A(x, ENV) =_{\text{def}} \{x\};$$

(2) Tupel

$$A(E_1, \dots, E_n, ENV) =_{\text{def}} \{(e_1, \dots, e_n) : e_i \in A(E_i, ENV)\};$$

(3) Bewachte Ausdrücke

$$A(\text{if } B_1 \text{ then } E_1 \square \dots \square B_n \text{ then } E_n \text{ fi}, ENV) =_{\text{def}} \bigcup_{1 \leq i \leq n} A(\text{if } B_i \text{ then } E_i \text{ else } R_i \text{ fi}, ENV),$$

$R_i \cong \text{error}$, falls $n = 1$ und

$R_i \cong \text{if } B_1 \text{ then } E_1 \square \dots \square B_{i-1} \text{ then } E_{i-1} \square \square B_{i+1} \text{ then } E_{i+1} \square \dots \square B_n \text{ then } E_n \text{ fi}$
falls $n > 1$.

(3a) Bedingte Ausdrücke

$$\begin{aligned}
 A(\text{if } B \text{ then } E_1 \text{ else } E_2 \text{ fi}, \text{ENV}) &=_{\text{def}} \\
 \bigcup_{b' \in A(B, \text{ENV})} \{ &\text{if } b' \text{ then } e \text{ fi} : b \hat{=} b' // \text{ENV} \\
 &\wedge (d(b) \wedge V(b)) \Rightarrow e \in A(E_1, \text{ENV}) \wedge b'' \hat{=} b' \\
 &\wedge (d(b) \wedge \neg V(b)) \Rightarrow e \in A(E_2, \text{ENV}) \wedge b'' \hat{=} \neg b' \\
 &\wedge \neg d(b) \Rightarrow e \hat{=} \underline{\text{error}} \quad \wedge b'' \hat{=} b' \}
 \end{aligned}$$

(4) Objektdeklaration

$$\begin{aligned}
 A(\overline{(\underline{m}_1 x_1, \dots, \underline{m}_n x_n)} = E; E_{\underline{O}}, \text{ENV}) &=_{\text{def}} \\
 \{ \overline{(\underline{m}_1 x_1, \dots, \underline{m}_n x_n)} = e; e_{\underline{O}} : e \in A(E, \text{ENV}) \\
 &\wedge d(e) \Rightarrow e_{\underline{O}} \in A(E_{\underline{O}}, \text{ENV}') \\
 &\wedge \neg d(e) \Rightarrow e_{\underline{O}} \hat{=} \underline{\text{error}} \}
 \end{aligned}$$

wobei $\text{ENV}' = \text{ENV} \cup \{x_1 = e_1, \dots, x_n = e_n\}$
 und $(e_1, \dots, e_n) = V(e // \text{ENV})$

(5) Funktionsapplikation

Sei f eine primitive Funktion, dann gilt

$$\begin{aligned}
 A(f(E_1, \dots, E_n), \text{ENV}) &=_{\text{def}} \\
 \{f(e_1, \dots, e_n) : e_i \in A(E_i, \text{ENV})\}
 \end{aligned}$$

Falls f keine primitive Funktion ist, und f definiert ist durch funct $f = (\underline{m}_1 x_1, \dots, \underline{m}_n x_n) \underline{f} : E$, dann gilt

$$\begin{aligned}
 A(f(E_1, \dots, E_n), \text{ENV}) &=_{\text{def}} \\
 A(\overline{(\underline{m}_1 x_1, \dots, \underline{m}_n x_n)} = (E_1, \dots, E_n); E_{\underline{f}}, \text{ENV})
 \end{aligned}$$

Falls f eine rekursiv definierte Funktion darstellt und der Aufruf $f(E_1, \dots, E_n)$ nicht terminiert, wird durch die obige Gleichung ein unendlicher Ablauf definiert (vgl. /Nivat 75/ und Abschnitt 5.4). Um die Konvention beibehalten zu können, nach der jede Bezeichnung in einem Programm höchstens einmal deklariert wird, müssen die gebundenen Bezeichnungen gegebenenfalls umbenannt werden.

Die Abläufe eines Programms können auch durch Transformationsregeln abgeleitet werden. Sie können stets als deterministische Abkömmlinge (vgl. /Bauer, Wössner 79/) aufgefaßt werden.

Man beachte, daß auch determinierte Ausdrücke mehrere verschiedene Abläufe besitzen können, wenn sie nichtdeterministische Sprachelemente enthalten.

Die hier definierten Abläufe besitzen eine gewisse Ähnlichkeit mit den "Traces" aus /Hoare 78a/. Allerdings werden dort imperative Programme betrachtet, die die Menge aller Pfade durch das dem Programm entsprechende Flußdiagramm als Traces zugeordnet erhalten. Traces bestehen somit aus einfachen Zuweisungen und Zusicherungen, die aus den Bedingungen im Flußdiagramm entstehen. Ein "Trace" wird als zulässig ("feasible") bezeichnet, wenn alle auftretenden Zusicherungen zutreffen. Damit wird die Definition der "Traces" rein kombinatorisch ohne Einbeziehung der Semantik vorgenommen. Im Gegensatz zu den hier definierten Abläufen ist ein "Trace" selbst kein Programm.

Beispiel: Ein Ablauf für die Fakultätsfunktion

Sei funct fac = (nat n)nat: if n=0 then 1 else n * fac(n-1) fi,
dann erhält man als Ablauf von fac(3):

nat n1 = 3; if ¬n1=0 then n1*
nat n2 = n1-1; if ¬n2=0 then n2*
nat n3 = n2-1; if ¬n3=0 then n3*
nat n4 = n3-1; if n4=0 then 1 fi, fi, fi, fi

Nach "UNFOLD" der Objektdeklarationen (vgl. dazu Abschnitt 1.1.3) ergibt sich:

if¬3=0 then 3*if¬2=0 then 2*if¬1=0 then 1*
if 0=0 then 1 fi fi fi fi

und nach Elimination der Wächter erhalten wir den "reduzierten" Ablauf:

3 * (2 * (1 * 1)).

1.1.2 Ablaufäquivalenz

Das folgende Theorem stellt die Beziehung zwischen der mathematischen Semantik und den Abläufen eines Ausdrucks her.

Theorem: Für jeden Ausdruck E mit einer Umgebung ENV gilt:

- (i) alle $e \in A(E, ENV)$ sind determinierte Ausdrücke und es gilt:

$$d(e//ENV) \Rightarrow \forall (e//ENV) \in B(E//ENV)$$

$$\wedge \neg d(e//ENV) \Rightarrow \neg d(E//ENV),$$
- (ii) $|A(E, ENV)| < \infty \vee \exists e \in A(E, ENV): \text{infinite}(e),$
- (iii) $\forall e \in A(E, ENV): A(e, ENV) = \{e\},$
- (iv) $B(E//ENV) = \bigcup_{e \in A(E, ENV)} B(e//ENV), \quad d(E//ENV) = \bigwedge_{e \in A(E, ENV)} d(e//ENV).$

Beweis: (i) Die Definitionen der Abläufe enthalten nur deterministische Sprachelemente. Der Rest folgt aus (iv).

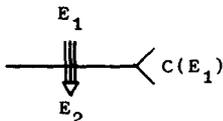
(ii) Induktion über die Länge der Terme liefert die Behauptung (Königs Lemma)¹⁾

(iii) Folgt trivial aus der Definition der Abläufe.

(iv) Ein Vergleich der Definitionen der Breite und Definiertheit mit der Definition der Abläufe liefert über strukturelle Induktion die Behauptung. □

Durch die Abläufe läßt sich nun eine feinere Äquivalenzrelation als die mathematische Äquivalenz auf Programmen definieren.

Definition: Zwei Ausdrücke E_1 und E_2 heißen ablaufäquivalent, wenn für jede Umgebung ENV gilt: $A(E_1, ENV) = A(E_2, ENV)$. Gilt unter der Bedingung $C(E_1)$ die Ablaufäquivalenz von E_1 und E_2 , so schreiben wir:



Gilt nur $C(E_1) \Rightarrow A(E_2, ENV) \subseteq A(E_1, ENV)$, so sagen wir " E_2 implementiert E_1 bzgl. des Ablaufverhaltens". Trivialerweise ergibt sich aus der Ablaufäquivalenz die mathematische Äquivalenz.

¹⁾ vgl. D. König: Theorie der endlichen und unendlichen Graphen. Chelsea Publishing Company, New York, N.Y. 1950

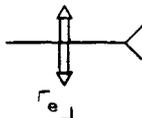
1.1.5 Reduzierte Abläufe

Abläufe enthalten nur noch Sprachelemente der Form:

- Blöcke und Objektdeklarationen,
- bewachte Ausdrücke mit genau einer Alternative,
- Applikationen primitiver Funktionen.

Entfernt man in einem (definierten) Ablauf alle bewachten Ausdrücke durch die Regel:

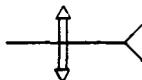
if b then e fi



$$d(b) \wedge B(b) = \{ \underline{\text{true}} \}$$

und alle kollektiven Objektdeklarationen durch die Regel:

$$\underline{(\overline{m_1} x_1, \dots, \overline{m_n} x_n)} = (e_1, \dots, e_n); \underline{e}$$



$$\forall i, 1 \leq i \leq n: \text{determinate}(e_i) \wedge d(e_i)$$

$$e[e_1/x_1, \dots, e_n/x_n]$$

dann entsteht ein Ausdruck, genannt reduzierter Ablauf, der nur noch ein (instantiierter) Term der Termalgebra des zugrundeliegenden abstrakten Typs (vgl. /Broy et al. 79/) ist.

Damit beschreiben Programme (eine Menge von) Terme(n) der Termalgebra eines abstrakten Typs. Auf diese Weise kann man eine weitere Kongruenzrelation auf der Menge der Programme definieren: zwei Programme sind genau dann algorithmisch äquivalent, wenn sie die gleichen (Mengen von) reduzierten Abläufe besitzen.

Damit lassen sich durch Abläufe sowohl die Beziehung zwischen abstrakten Typen und darüber definierten, rekursiven Funktionen (strukturelle Rekursion), als auch gewisse Äquivalenzrelationen auf der Menge der Programme beschreiben.

1.2 Aktionen, Prozesse, Ablaufordnungen, Effizienzmaße

Bei der Abarbeitung eines Programms in einer Rechenanlage werden eine Reihe von Aktionen ausgeführt. Die Menge der in einer gewissen zeitlichen Abfolge ausgeführten Aktionen bezeichnen wir als Prozeß. Die Aktionen eines Prozesses werden teilweise hintereinander ("sequentiell") und teilweise nebeneinander ("parallel") ausgeführt. Um dies auszudrücken, definieren wir gewisse transitive Relationen ("Ablaufordnungen") auf der Menge der Aktionen.

Die Menge der Aktionen eines Ablaufs bestimmt auch den Aufwand, den die Abarbeitung in einer Rechenanlage mit sich bringt. Da in einem Ablauf die Menge der Aktionen, die zu seiner Ausführung notwendig sind, in einer eins-zu-eins Korrespondenz zu den Unter-
ausdrücken des Ablaufs stehen, lassen sich für Abläufe einfach "Effizienzmaße" definieren. Mit Hilfe dieser Effizienzmaße können die Unterschiede zwischen "sequentieller" und "paralleler" Ausführung bzgl. des Zeit- und Betriebsmittelaufwands formal beschrieben werden.

1.2.1 Aktionen

Jede Auswertung eines Programm(teil)ausdrucks bezeichnen wir als Aktion. Aktionen heißen elementar, wenn sie nicht weiter in Unteraktionen zerlegbar sind. Anderenfalls heißen Aktionen zusammengesetzt. Sie zerfallen dann in eine Anzahl von Unteraktionen, entsprechend den Teilausdrücken des vorliegenden Ablaufs.

Bei den meisten Programm(ausdrück)en werden bestimmte Teilausdrücke mehrfach ausgewertet, wie z.B. die Rumpfe rekursiver Funktionen oder Wiederholungsanweisungen. Andere Teilausdrücke werden - geschützt durch Wächter in bewachten Ausdrücken - unter Umständen überhaupt nicht ausgewertet. Die Abbildung der Menge der durchgeführten Aktionen bei der Auswertung eines Programms auf die Menge der Teilausdrücke des Programms ist im allgemeinen weder surjektiv, noch injektiv.

Bei einem Ablauf jedoch ergibt sich eine bijektive Beziehung zwischen den Teilausdrücken (d.h. der statischen Aufschreibung) und den bei der Ausführung des Programms anfallenden Aktionen (dem dynamischen Ablauf). Diese eindeutige Zuordnung von Aktionen zu (Sub-)Termen eines Ablaufes erleichtert es, über die Aktionen zu sprechen, da jede Aktion über den ihr entsprechenden Teilausdruck bezeichnet werden kann. Somit spiegeln die in Abschnitt 1.1 definierten Abläufe wirklich den "dynamischen" Ablauf eines Programmes wider.

Treten in einem Ablauf textuell gleiche Subterme auf, so kann stets durch eine entsprechende Indizierung eine eindeutige Kennzeichnung der verschiedenen Exemplare erreicht werden.

Definition: Einem Ablauf E sei durch "act(E)" diejenige Aktion zugeordnet, die bei Abarbeitung von E durchgeführt wird. Die Menge der (Teil-)Aktionen, die bei der Ausführung von E erforderlich ist, bezeichnen wir mit "actions(E)". Induktiv definieren wir:

- (1) Falls $E \hat{=} (e_1, \dots, e_n)$
$$\text{actions}(E) = \bigcup_{1 \leq i \leq n} \text{actions}(e_i) \cup \text{act}(E)$$
- (2) Falls $E \hat{=} "(\underline{m}_1 x_1, \dots, \underline{m}_n x_n) = e"$
$$\text{actions}(E) = \text{actions}(e) \cup \text{act}(E)$$
- (3) Falls $E \hat{=} \lceil D; e_0 \rceil$, wobei D Definition vom Typ (2) sei
$$\text{actions}(E) = \text{actions}(e_0) \cup \text{actions}(D) \cup \text{act}(E)$$
- (4) Falls $E = \text{if } p \text{ then } e \text{ fi}$
$$\text{actions}(E) = \text{actions}(p) \cup \text{actions}(e) \cup \text{act}(E)$$
- (5) Falls $E \hat{=} f(e_1, \dots, e_n)$ wobei f primitiv sei
$$\text{actions}(E) = \bigcup_{1 \leq i \leq n} \text{actions}(e_i) \cup \text{act}(E)$$

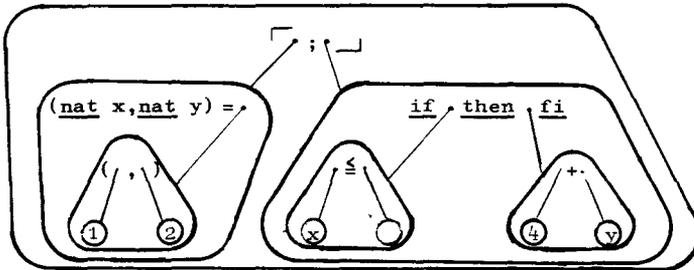
Die Vereinigung in dieser Definition wird stets als disjunkt vorausgesetzt (bei vorheriger Indizierung textuell gleicher Teilausdrücke) und kann als direkte Summe verstanden werden. Ein Ausdruck, der nur aus einer Bezeichnung besteht, heißt elementar.

Durch die Definition der Aktionen wird eine Struktur auf den Programmtermen eines Ablaufs induziert, die der Struktur des "abstrakten Syntaxbaums" ("Kantorovicz-Baum") entspricht.

Beispiel: Wir betrachten einen Ablauf über der Rechenstruktur der natürlichen Zahlen:

$$\overline{(\text{nat } x, \text{nat } y) = (1,2); \text{ if } x \leq 3 \text{ then } 4+y \text{ fi}}$$

In dem folgenden Diagramm wird die Struktur dieses Ablaufs deutlich:



Jedes im obigen Diagramm von einer Linie umschlossene Gebiet entspricht einer Aktion und enthält die Menge aller Teilaktionen, in die sie zerfällt.

Damit wird deutlich, daß die Definition der Aktionen eines Ablaufs genau der häufig anzutreffenden Mengentheoretischen Definition von Bäumen entspricht:

Definition: (Mengentheoretische Beschreibung von Bäumen)

Ein (orientierter) Baum über einer gegebenen Menge M ist eine Menge, die genau ein Element aus M , genannt "Wurzel", und eine endliche, eventuell leere Menge von Bäumen enthält.

Die obige Definition kann etwas exakter, unter Ausschluß unendlicher Bäume, induktiv erfolgen. Sei $M_0 =_{\text{def}} \{x\} : x \in M\}$

$$M_{i+1} =_{\text{def}} M_i \cup \{ \{x\} \cup p : x \in M \wedge p \subset M_i \wedge |p| < \infty \}$$

dann ergibt $\bigcup_{i \in \mathbb{N}} M_i$ die Menge der orientierten Bäume über M .

1.2.2 Partielle Ordnungsrelationen auf Aktionen

Auf der Menge $\text{actions}(E)$ von Aktionen eines Ablaufs E definieren wir nun Präordnungen zur Beschreibung der zeitlichen Beziehung zwischen zwei Aktionen (vgl. /Greif 77/, für die Verwendung solcher Relationen bei der Beschreibung von Speichern vgl. /Plickert 77/). Ein zu einfaches Modell erhält man, wenn man jeder Aktion einen Punkt der Zeitachse zuordnet, da so komplexere, zeitliche Überlagerungen von Aktionen nicht mehr ausgedrückt werden können.

Deshalb wählen wir ein komplexeres, aber dafür realistischeres Modell. Da die absoluten Zeiten ohne Bedeutung sind und uns nur die zeitlichen Relationen zwischen den Aktionen interessieren, abstrahieren wir von der absoluten Zeit durch die Einführung dreier Relationen. Sei E ein Ablauf und $a, b \in \text{actions}(E)$:

$a \varepsilon_a b \Rightarrow_{\text{def}}$ Die Aktion a beginnt bevor die Aktion b beginnt oder gleichzeitig mit ihr.

$a \varepsilon_z b \Rightarrow_{\text{def}}$ Die Aktion a endet bevor die Aktion b endet oder gleichzeitig mit ihr.

$a \varepsilon_t b \Rightarrow_{\text{def}}$ Die Aktion a endet, bevor die Aktion b beginnt.

Ein Tripel von Relationen heißt Ablaufordnung für den Ablauf e , wenn die Relationen ε_a , ε_z und ε_t die folgenden Bedingungen erfüllen:

- (1) ε_a , ε_z und ε_t sind transitiv,
- (2) ε_a und ε_z sind reflexiv,
- (3) ε_t ist strikt, also asymmetrisch und irreflexiv,
- (4) Für alle $a, b, c \in \text{actions}(e)$ gilt:

$$a \varepsilon_t b \Rightarrow a \varepsilon_a b \wedge a \varepsilon_z b \wedge \neg b \varepsilon_a a \wedge \neg b \varepsilon_a a,$$

$$a \varepsilon_t b \wedge c \varepsilon_z a \Rightarrow c \varepsilon_t b,$$

$$a \varepsilon_t b \wedge c \varepsilon_a a \Rightarrow c \varepsilon_a b \wedge \neg b \varepsilon_a c,$$

$$b \in \text{actions}(a) \Rightarrow a \varepsilon_a b \wedge b \varepsilon_z a.$$

$$(5) \quad a = \text{act}(D; E) \Rightarrow \text{act}(D) \sqsubseteq_t \text{act}(E);$$

$$(6) \quad a = \text{act}(\underline{\text{if } B \text{ then } E \text{ fi}}) \Rightarrow \text{act}(B) \sqsubseteq_t \text{act}(E);$$

Hier wird eine sequentielle Abarbeitung von Bedingung und Alternative in einem bewachten Ausdruck vorausgesetzt. Dies ist jedoch nicht zwingend. Man kann auch die Auswertung von Bedingung und Alternative simultan vornehmen, um z.B. wie in /Friedman, Wise 78/ vorgeschlagen, die Äquivalenz

$$\underline{\text{if } B \text{ then } A \text{ else } A \text{ fi}} \quad \Leftrightarrow \quad A$$

auch für nichtterminierende Wächter B sicherzustellen. Diese Vorgehensweise erscheint jedoch weder vom Standpunkt der zuverlässigen Programmentwicklung noch vom Gesichtspunkt der Effizienz als sinnvoll (vgl. dazu Abschnitt 2.3 und 3.3).

Definition: Eine mögliche Ablaufordnung heißt vollständig, wenn für alle Aktionen a, b gilt

$$(a \sqsubseteq_a b \vee b \sqsubseteq_a a) \wedge (a \sqsubseteq_z b \vee b \sqsubseteq_z a);$$

gilt zusätzlich

$$((a \sqsubseteq_a b \wedge b \sqsubseteq_a a) \Rightarrow (a = b))$$

$$((a \sqsubseteq_z b \wedge b \sqsubseteq_z a) \Rightarrow (a = b))$$

d.h. sind \sqsubseteq_a und \sqsubseteq_b Halbordnungen, so heißt die Ablaufordnung total.

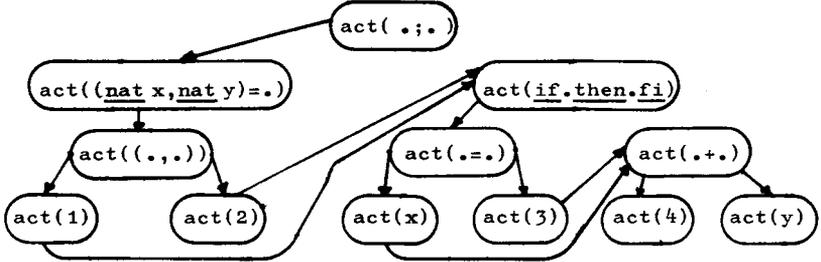
Jedes Programm besitzt mindestens eine totale Ablaufordnung.

Definition: Eine Ablaufordnung heißt sequentiell, wenn für alle Aktionen a, b gilt:

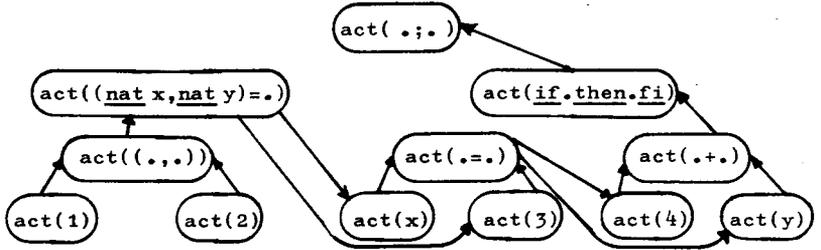
$$a \sqsubseteq_t b \vee b \sqsubseteq_t a \vee a \in \text{actions}(b) \vee b \in \text{actions}(a)$$

Dem hier definierten Begriff von Ablaufordnung liegt die Annahme zugrunde, daß es keine Aktionen gibt, die keinerlei Zeit beanspruchen. Deshalb ist \sqsubseteq_t als asymmetrische Relation definiert. Aktionen können jedoch gleichzeitig beginnen oder enden.

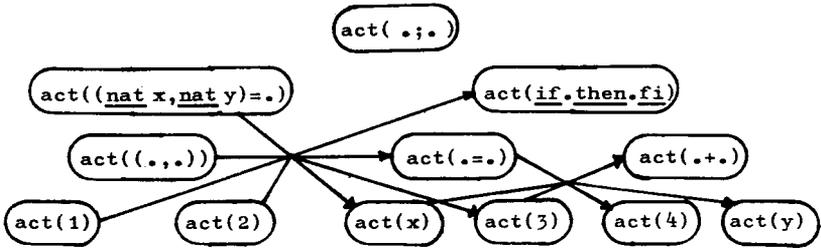
Um die Struktur der Ablaufordnungen genauer zu erläutern, geben wir nun für das Beispiel aus Abschnitt 1.2.1 die drei Relationen graphisch wieder. Der Ξ_a -Relation entspricht:



Der Ξ_z -Relation entspricht:



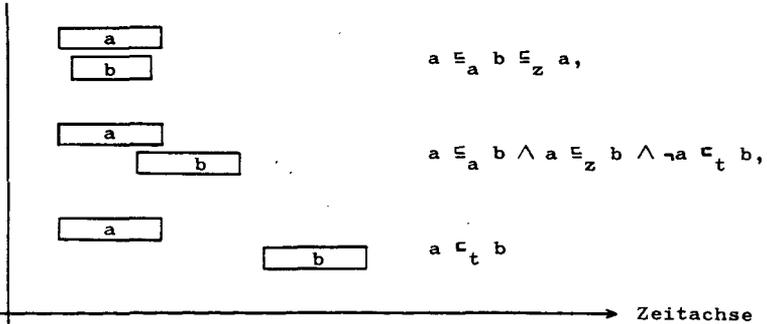
Der Ξ_t -Relation entspricht:



Jede der Relationen einer Ablaufordnung enthält die durch die obigen Graphen erzeugten transitiven Relationen. Man beachte, daß die Ξ_a -Relation in gewisser Weise dem Durchlaufen des abstrakten Syntaxbaums in Vorordnung, die Ξ_z -Ordnung dem Durchlaufen in Endordnung entspricht.

1.2.3 Parallele und sequentielle Aktionen

Einen Ablauf e mit einer Ablaufordnung $(\varepsilon_a, \varepsilon_z, \varepsilon_t)$ nennen wir Prozeß. Für zwei Aktionen a, b lassen sich im wesentlichen drei elementare Situationen zeitlicher Verzahnung angeben:



Definition: Zwei Aktionen a, b eines Prozesses heißen sequentiell, wenn gilt: $a \varepsilon_t b \vee b \varepsilon_t a$. Nichtsequentielle Aktionen, die keine gemeinsamen Unteraktionen besitzen, heißen parallel.

Damit wären im obigen Diagramm die beiden ersten Paare Beispiele für parallele, das letzte ein Beispiel für sequentielle Aktionen.

Die Menge der Ablaufordnungen für einen Ablauf läßt sich partiell ordnen. Für Ablaufordnungen $(\varepsilon_a, \varepsilon_z, \varepsilon_t)$ und $(\varepsilon'_a, \varepsilon'_z, \varepsilon'_t)$ sagen wir " $(\varepsilon_a, \varepsilon_z, \varepsilon_t)$ ist schwächer als $(\varepsilon'_a, \varepsilon'_z, \varepsilon'_t)$ ", falls für alle Aktionen a, b gilt:

$$(a \varepsilon_a b \Rightarrow a \varepsilon'_a b) \wedge (a \varepsilon_z b \Rightarrow a \varepsilon'_z b) \wedge (a \varepsilon_t b \Rightarrow a \varepsilon'_t b).$$

Da der Durchschnitt zweier Ablaufordnungen selbst wieder eine Ablaufordnung darstellt, existiert eine schwächste Ablaufordnung.

Definition: Zwei Unter ausdrücke e_1, e_2 eines Ablaufs e heißen sequentiell (bzw. parallel), falls die Aktionen $\text{act}(e_1)$ und $\text{act}(e_2)$ in der schwächsten Ablaufordnung sequentiell (bzw. parallel) sind.

Ein Prozeß heißt quasiparallel, falls alle elementaren Aktionen sequentiell, aber gewisse nichtelementare Aktionen parallel sind.

1.2.4 Effizienzmaße

Um über die Effizienz von Programmen sprechen zu können, führen wir Bewertungsfunktionen auf Abläufen ein.

Definition: Eine Abbildung M , die jedem Ablauf e eine rationale Zahl $M(e)$ zuordnet, heißt Effizienzmaß, wenn M monoton ist, d.h. wenn für alle Abläufe e_1, e_2 gilt:

$$\text{actions}(e_1) \subseteq \text{actions}(e_2) \Rightarrow M(e_1) \leq M(e_2).$$

Aus der Fülle der möglichen Effizienzmaße seien drei besonders elementare herausgegriffen. Für Bezeichnungen und primitive Funktionen geben wir bewusst keine Werte für die Effizienzmaße an. Wir setzen vielmehr irgendwelche gegebene Werte für die "Zugriffszeiten" bei Bezeichnungen und die "Auswertungszeiten" für die Aufrufe primitiver Funktionen voraus.

Das Zeitmaß gibt die Ablaufzeit bei unbeschränkter Anzahl von Prozessoren wieder (bzgl. einer Umgebung ENV):

$$\begin{aligned} \text{time}(e_1, \dots, e_n) &=_{\text{def}} \max \{ \text{time}(e_1), \dots, \text{time}(e_n) \}, \\ \text{time}(\text{if } b \text{ then } e \text{ fi}) &=_{\text{def}} \text{time}(b) + \text{time}(e), \\ \text{time}(f(e_1, \dots, e_n)) &=_{\text{def}} \text{time}(f(v(e_1/\text{ENV}), \dots, v(e_n/\text{ENV}))) + \\ &\quad \text{time}(e_1, \dots, e_n), \\ \text{time}(\overline{m_1 x_1, \dots, m_n x_n} = e_1; e_2) &=_{\text{def}} \text{time}(e_1) + \text{time}(e_2). \end{aligned}$$

Das Prozessorenmaß gibt die maximale Anzahl gleichzeitig benötigter Prozessoren wieder:

$$\begin{aligned} \text{processor}(e_1, \dots, e_n) &=_{\text{def}} \sum_{1 \leq i \leq n} \text{processor}(e_i), \\ \text{processor}(\text{if } b \text{ then } e \text{ fi}) &=_{\text{def}} \max \{ \text{processor}(b), \text{processor}(e) \}, \\ \text{processor}(f(e_1, \dots, e_n)) &=_{\text{def}} \\ &\quad \text{processor}(f(v(e_1/\text{ENV}), \dots, v(e_n/\text{ENV}))) + \sum_{1 \leq i \leq n} \text{processor}(e_i), \\ \text{processor}(\overline{m_1 x_1, \dots, m_n x_n} = e_1; e_2) &=_{\text{def}} \\ &\quad \max \{ \text{processor}(e_1), \text{processor}(e_2) \}. \end{aligned}$$

Die Prozessorzeit gibt die Summe über die Belegungszeiten der einzelnen Prozessoren wieder:

$$\text{ptime}(e_1, \dots, e_n) =_{\text{def}} \sum_{1 \leq i \leq n} \text{ptime}(e_i),$$

$$\text{ptime}(\text{if } b \text{ then } e \text{ fi}) =_{\text{def}} \text{ptime}(b) + \text{ptime}(e),$$

$$\text{ptime}(f(e_1 // \text{ENV}, \dots, e_n // \text{ENV})) =_{\text{def}} \text{ptime}(f(V(e_1 // \text{ENV}), \dots, V(e_n // \text{ENV}))) + \sum_{1 \leq i \leq n} \text{ptime}(e_i),$$

$$\text{ptime}(\sqrt{\underline{m}_1 x_1, \dots, \underline{m}_n x_n} = e_1; e_2) =_{\text{def}} \text{ptime}(e_1) + \text{ptime}(e_2).$$

Mit Hilfe dieser drei Effizienzmaße können wir in den folgenden Abschnitten die Effizienz bestimmter Sprachelemente bzgl. paralleler Auswertung diskutieren. Natürlich sind die eingeführten Maße nur eingeschränkt für praktische Untersuchungen zu gebrauchen, weil beispielsweise in der Realität stets nur von einer fest vorgegebenen oberen Schranke p für die Anzahl der gleichzeitig zur Verfügung stehenden Prozessoren ausgegangen werden kann.

Allerdings kann man dann $\text{ptime}(e)/p$ als untere Schranke und $\text{ptime}(e)$ als obere Schranke für die Ausführungszeit verwenden, falls die Anzahl der parallel benötigten Prozessoren die Anzahl der zur Verfügung stehenden Prozessoren übersteigt, d.h. falls $\text{processor}(e) > p$. Präzisere Maßzahlen sind stark von den verwendeten "Sequentialisierungsstrategien" abhängig.

Ebenso wird in den drei Effizienzmaßen der Verwaltungsaufwand für die "Aufspaltung" und "Zusammenführung" paralleler Aktionen vernachlässigt. Neben den obigen Effizienzmaßen lassen sich natürlich noch weitere definieren, um zum Beispiel den Speicherplatzbedarf zu messen. Für unsere Zwecke sind die drei obigen Effizienzmaße jedoch völlig ausreichend.

2 Parallele Abläufe der applikativen Ebene

In den im vorangegangenen Kapitel definierten Abläufen sind gewisse Ausdrücke parallel auswertbar. So können die durch Kommata getrennten Ausdrücke in den Parameterlisten von Funktionen und in Tupeln parallel abgearbeitet werden. Dies spiegelt sich sowohl in der Definition der Ablaufordnungen, als auch in der Definition der Effizienzmaße wider, wo stets eine parallele Auswertung unterstellt wird. Jedoch lassen sich so nicht alle Möglichkeiten für eine parallele Auswertung in befriedigender Weise erfassen.

Deshalb werden in dem folgenden Abschnitt die Sprachelemente geringfügig modifiziert, so daß eine eingeschränkte Kommunikation zwischen parallel ablaufenden Programmen möglich wird. Dabei werden die gebundenen Bezeichnungen in einer parallelen Objektdeklaration als Schlüssel- und Stichworte für die Kommunikation verwendet.

Auf der Basis der Parallelerdeklaration läßt sich eine "nicht-deterministische" Berechnungsregel definieren. Eine Ausschöpfung aller Möglichkeiten dieser Regel in Hinblick auf parallele Verarbeitung führt zur "Call-in-Parallel"-Auswertungsstrategie, die für eine ganze Reihe von Funktionen eine weitgehend parallele Abarbeitung sicherstellt.

2.1 Konzepte der applikativen Ebene

Der Begriff der "applikativen Ebene" ist geprägt von der Funktionsapplikation als beherrschendem Sprachelement. Da Funktionen ausschließlich von ihren Argumenten abhängen, können mehrere Funktionsapplikationen ohne die Gefahr irgendwelcher Konflikte parallel abgearbeitet werden. Dabei können Funktionen als "simultan benutzbar" im Sinne von /Seegmüller 74/ vorausgesetzt werden. Funktionsaufrufe können auch als Muster für eine spezielle Form der Kommunikation zwischen parallelen Prozessen verstanden werden. Jedoch muß eine solche Auffassung etwas erweitert werden, um bestimmte Fälle für parallele Verarbeitung befriedigend behandeln zu können.

2.1.1 Funktionsapplikation und parallele Auswertung

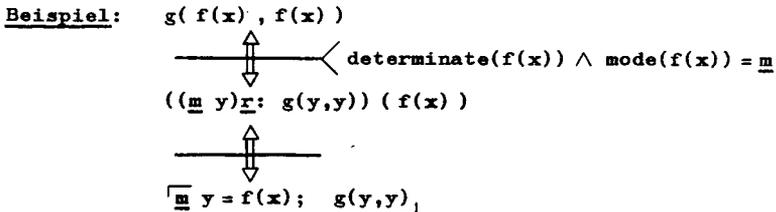
Da Ausdrücke stets frei von konflikträchtigen Seiteneffekten sind, ist eine parallele Auswertung der Liste der aktuellen Argumentausdrücke einer Funktionsapplikation oder der Liste der Komponentenausdrücke eines Tupels problemlos möglich. Dementsprechend sind auch die Definitionen für die Effizienzmaße in Abschnitt 1.2.4. Die parallele Auswertung der n Argumentausdrücke eines Aufrufs einer n -stelligen Funktion kann man durch die Begriffe der Aufspaltung und Sammlung (vgl. /Bauer, Wössner 79/) in folgender Weise beschreiben. Die Auswertung des Ausdrucks beginnt mit einer Aufspaltung des Berechnungsprozesses in n unabhängige, parallel ablaufende Prozesse zur Auswertung der n Argumentausdrücke. Diese n Prozesse werden nach ihrer Beendigung wieder zu einem Prozeß zusammengeführt ("gesammelt"), der dann die Berechnung des Funktionswerts entsprechend der vorliegenden Argumentwerte vornimmt. Diese Vorgehensweise ergibt sich aus der "Call-by-Value"-Auffassung oder einer "Innermost"-Regel (genauer gesagt einer "Innermost-Parallel"-Regel, vgl. /Manna et al. 73/).

Die oben beschriebene parallele Auswertung der Parameterliste von Funktionsaufrufen entspricht einem sehr starren Schema von Aufspaltung und Sammlung, da jeder Aufspaltung in n parallele Prozesse genau eine Sammlung der n parallelen Prozesse gegenübersteht. Zwischen den parallel ablaufenden Prozessen findet keinerlei Informationsaustausch, also keinerlei Kommunikation, statt.

Wie im folgenden Abschnitt demonstriert wird, genügt jedoch dieses einfache Schema der parallelen Auswertung von Funktionsaufrufen nicht, um für alle Ausdrücke befriedigend effiziente, parallele Auswertungen zu erreichen.

2.1.2 Gleiche Teilausdrücke und parallele Auswertung

Um unnötigen Mehraufwand zu vermeiden, d.h. um neben dem Effizienzmaß "time" auch die Prozessorenzeit "ptime" so klein wie möglich zu halten, ist man bestrebt gleiche Teilausdrücke in Objektvereinbarungen herauszuziehen.



Komplizierter wird die Situation jedoch, wenn wir Ausdrücke der Form E_1 betrachten. Die Umformung liefert den Ausdruck E_2 .

$$E_1 \hat{=} (g_1(f(x), h_1(x)), g_2(f(x), h_2(x))),$$

$$E_2 \hat{=} ((\underline{m} \ y) \underline{x}: (g_1(y, h_1(x)), g_2(y, h_2(x))) (f(x)),$$

Doch in E_2 wird bei strikter Anwendung der "Call-by-Value"-Berechnungsregel mit der Auswertung des Funktionsrumpfes erst begonnen, wenn die Berechnung des Wertes von $f(x)$ schon abgeschlossen ist, obwohl $h_1(x)$ und $h_2(x)$ parallel zu $f(x)$ ausgewertet werden können. Dies geschieht in E_1 oder in E_3 .

$$E_3 \hat{=} ((\underline{m} \ y, \underline{m}_1 \ y_1, \underline{m}_2 \ y_2) \underline{x}: (g_1(y, y_1), g_2(y, y_2))) (f(x), h_1(x), h_2(x)).$$

Jedoch wird in E_3 mit der Berechnung von $g_1(y, y_1)$ bzw. $g_2(y, y_2)$ erst begonnen, wenn die Auswertung von $f(x)$, $h_1(x)$ und $h_2(x)$ abgeschlossen ist, obwohl für die Berechnung von $g_1(y, y_1)$ (bzw. $g_2(y, y_2)$) der Wert von $h_2(x)$ (bzw. $h_1(x)$) nicht benötigt wird.

Wir wollen nun die unterschiedlichen Werte der Effizienzmaße "time" und "ptime" für die drei Ausdrücke E_1 , E_2 und E_3 vergleichen. Sei $t = \text{time}(f(x))$ und für $i = 1, 2$ sei $t_i = \text{time}(h_i(x))$, $z_i = \text{time}(g_i(y, y_i))$, wobei $y = V(f(x))$ und $y_i = V(h_i(x))$. Dann gilt:

$$\text{time}(E_1) = \max \{ z_1+t, z_1+t_1, z_2+t, z_2+t_2 \},$$

$$\text{time}(E_2) = \max \{ z_1+t+t_1, z_2+t+t_2 \},$$

$$\text{time}(E_3) = \max \{ z_1+t, z_1+t_1, z_1+t_2, z_2+t, z_2+t_1, z_2+t_2 \}.$$

Sei $p = \text{ptime}(f(x))$ und für $i = 1, 2$ sei $p_i = \text{ptime}(h_i(x))$,
 $q_i = \text{ptime}(g_i(y, y_i))$, wobei $y = V(f(x))$ und $y_i = V(h_i(x))$. Es
gilt:

$$\text{ptime}(E_1) = 2p + p_1 + p_2 + q_1 + q_2,$$

$$\text{ptime}(E_2) = \text{ptime}(E_3) = p + p_1 + p_2 + q_1 + q_2.$$

Damit ergibt sich der folgende Vergleich:

$$\text{time}(E_1) \leq \text{time}(E_2) \wedge \text{time}(E_1) \leq \text{time}(E_3)$$

$$\text{ptime}(E_1) \geq \text{ptime}(E_2) = \text{ptime}(E_3).$$

Die Effizienz des Ausdrucks E_1 liegt damit im Vergleich zu den
anderen beiden Ausdrücken, was das Zeitmaß "time" betrifft,
am günstigsten, jedoch am schlechtesten in Beziehung auf
die Prozessorenzeit "ptime". Man beachte, daß unter der
realistischen Annahme, daß alle Werte $t, t_1, t_2, z_1, z_2,$
 p, p_1, p_2, q_1 und q_2 echt größer als 0 sind, sogar gilt:

$$\text{time}(E_1) < \text{time}(E_2),$$

$$\text{ptime}(E_1) > \text{ptime}(E_2) = \text{ptime}(E_3).$$

Die Werte von $\text{time}(E_2)$ und $\text{time}(E_3)$ sind unvergleichbar.

Natürlich ist man daran interessiert, die Effizienzvorteile
von E_1 bzgl. des Zeitmaßes "time" mit den Vorzügen von E_2
bzw. E_3 bzgl. der Prozessorenzeit "ptime" zu vereinigen.
Dies ist mit Hilfe der im folgenden Abschnitt eingeführten
Paralleldeklaration von Objekten möglich.

Falls übrigens der Ausdruck $f(x)$ nicht determiniert ist, ist
der Übergang von E_1 zu E_2 bzw. E_3 ohnehin im allgemeinen
keine Äquivalenztransformation, sondern es gilt nur $E_2 \sqsubseteq_I E_1$
bzw. $E_3 \sqsubseteq_I E_1$. Ein Übergang von E_2 oder E_3 zu E_1 ist dann
unmöglich.

2.2 Parallele Berechnungen

Die verschiedenen Berechnungsregeln, wie sie in /Manna et al. 73/ beschrieben werden, entsprechen verschiedenen Aufrufmechanismen. Sie unterscheiden sich durch die Auswahl der Aufrufstelle(n), an der rekursive Funktionsaufrufe durch den Rumpf der Funktionsdefinition ersetzt werden. Im folgenden wollen wir eine nicht-deterministische Berechnungsregel für nichtdeterministische, rekursive Funktionen mit Hilfe von Transformationsregeln definieren. Dabei wird der klassische Substitutionsschritt ("UNFOLD") in zwei Phasen aufgespalten. Zuerst ersetzen wir den rekursiven Aufruf durch einen Block, bestehend aus einer Parallelerdeklaration und dem eigentlichen Rumpf der Funktion. Anschließend werden die einzelnen Parameterwerte unabhängig voneinander (also nach Belieben "parallel") erarbeitet und jeweils nach Vorliegen substituiert. Der Ausdruck für das Resultat, der "eigentliche" Rumpf kann ebenfalls parallel dazu weiter ausgewertet werden.

Da wir eine Semantik zugrunde gelegt haben, in der die Call-by-Value-Regel korrekt ist, terminiert die parallele Berechnung genau dann, wenn jeder der parallel ablaufenden Berechnungsprozesse ordnungsgemäß terminiert. Modifiziert man die Transformationsregeln, die die Berechnungsregel verwendet, so, daß sie der Call-by-Name-Regel entsprechen, so terminiert die gesamte Berechnung genau dann, wenn der Berechnungsprozeß terminiert, der den Funktionsrumpf auswertet. Etwa noch nicht abgeschlossene Berechnungen von (nicht benötigten) Argumentwerten werden abgebrochen.

Im übrigen kann diese modifizierte Berechnungsregel so gehalten werden, daß sie die Vorteile der verzögerten Auswertung (vgl. "Call-by-Need" in /Wadsworth 71/ und "Delayed Evaluation" in /Vuillemin 74/) einschließt. Die verzögerte Auswertung erlaubt neben einer optimierenden Behandlung der Auswertung gewisser Aufrufe (vgl. "Lazy Evaluation" in /Henderson, Morris 76/ und /Friedman, Wise 76a/) auch die Auswertung bestimmter Aufrufe, die unendliche Objekte als Argumente enthalten (vgl. /Bauer 78/).

2.2.1 Die Paralleldeklaration

Um das in Abschnitt 2.1.2 aufgeworfene Problem lösen zu können, führen wir nun die "Paralleldeklaration" der folgenden Form ein:

$$\overline{m}_1 x_1 = E_1, \dots, \overline{m}_n x_n = E_n, \underline{E}$$

Sie bildet die Basis für eine Berechnungsregel in Abschnitt 2.2.2. Dabei gehen wir davon aus, daß die Ausdrücke E_1, \dots, E_n und E parallel ausgewertet werden. Dementsprechend definieren wir die Abläufe:

$$A(\overline{m}_1 x_1 = E_1, \dots, \overline{m}_n x_n = E_n, \underline{E}, ENV) =_{\text{def}} \bigcup_{1 \leq i \leq n} \left\{ \overline{m}_i x_i = e, \underline{b} : \begin{array}{l} e \in A(E_i, ENV') \wedge \neg \text{occurs}(x_1, \dots, x_n \text{ in } e) \\ \wedge (\neg d(e // ENV') \Rightarrow b \in \underline{\text{error}}) \\ \wedge (d(e // ENV') \Rightarrow b \in A(P_i, ENV \cup \{x = e // ENV\})) \end{array} \right.$$

$$\cup \left\{ \begin{array}{l} \text{falls } \exists i, 1 \leq i \leq n : \forall e \in A(E_i, ENV') : \\ \neg \text{occurs}(x_1, \dots, x_n \text{ in } e) \\ \underline{\text{error}} \text{ sonst} \end{array} \right.$$

wobei $ENV' = ENV \cup \{x_1 = \underline{\text{error}}, \dots, x_n = \underline{\text{error}}\}$,

$$P_i \hat{=} \overline{m}_1 x_1 = E_1, \dots, \overline{m}_{i-1} x_{i-1} = E_{i-1},$$

$$\overline{m}_{i+1} x_{i+1} = E_{i+1}, \dots, \overline{m}_n x_n = E_n, \underline{E}$$

Damit sind Abläufe für die Paralleldeklaration definiert. Die mathematische Semantik ergibt sich dann durch die folgende Transformationsregel:

$$\overline{m}_1 x_1 = E_1, \dots, \overline{m}_n x_n = E_n, \underline{E}$$

$$\Downarrow \neg \text{occurs}(x_1, \dots, x_n \text{ in } E_1)$$

$$\overline{m}_1 x_1 = E_1; \overline{m}_2 x_2 = E_2, \dots, \overline{m}_n x_n = E_n, \underline{E}$$

und durch Anwendung der folgenden Beziehung:

$$B(E // ENV) =_{\text{def}} \{y : e \in A(E, ENV) \wedge y = v(e // ENV)\}.$$

Man beachte, daß die Anwendbarkeitsbedingung für Paralleldeklarationen in Abläufen stets erfüllt ist. Jedoch braucht diese Bedingung für Paralleldeklarationen im allgemeinen nicht erfüllt zu sein (vgl. Abschnitt 2.2.5).

Mit der Paralleldeklaration läßt sich unser Beispiel aus Abschnitt 2.1.2 folgendermaßen formulieren:

$$\underline{\underline{m}}x = f(x), \underline{\underline{m}}_1 x_1 = h_1(x), \underline{\underline{m}}_2 x_2 = h_2(x), (g_1(y, y_1), g_2(y, y_2))$$

Die Effizienzmaße für die Paralleldeklaration werden in Abschnitt 2.2.4 definiert.

Die Beziehung zwischen Kollektivvereinbarung und Paralleldeklaration kennzeichnet die folgende Transformationsregel:

$$\begin{array}{c} \underline{\underline{m}}_1 x_1, \dots, \underline{\underline{m}}_n x_n = (E_1, \dots, E_n); E \\ \begin{array}{c} \updownarrow \\ \text{---} \\ \updownarrow \end{array} \rightarrow \neg \text{occurs}(x_1, \dots, x_n \text{ in } E_1, \dots, E_n) \\ \underline{\underline{m}}_1 x_1 = E_1, \dots, \underline{\underline{m}}_n x_n = E_n, E \end{array}$$

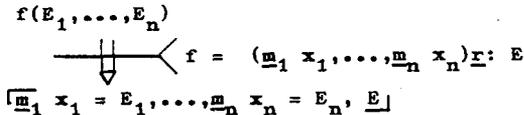
Sie läßt sich unschwer mit Hilfe der Semantikdefinition beweisen. Nicht immer braucht die Paralleldeklaration die obige Anwendbarkeitsbedingung erfüllen. Gilt beispielsweise nur das Hierarchiekriterium: Es existiert eine Permutation p_1, \dots, p_n , so daß gilt: $\forall i, 1 \leq i \leq n: \neg \text{occurs}(x_{p_1}, \dots, x_{p_n} \text{ in } E_{p_i})$, so ist die Paralleldeklaration nach entsprechender Umordnung der Reihenfolge (die Korrektheit der Vertauschung einzelner Deklarationen in der Paralleldeklaration ergibt sich trivial aus der Ablaufdefinition) in eine Folge sequentieller Deklarationen überführbar. Eine Paralleldeklaration braucht jedoch nicht einmal das Hierarchiekriterium erfüllen (vgl. Abschnitt 2.2.5).

Da durch die obige Transformationsregel Kollektivdeklarationen problemlos auf Paralleldeklarationen zurückgeführt werden können, kann man auch die Abläufe von Aufrufen nichtprimitiver Funktionen auf die Paralleldeklaration zurückführen. Eine Berechnungsregel, die die Paralleldeklaration verwendet, wird im folgenden Abschnitt definiert.

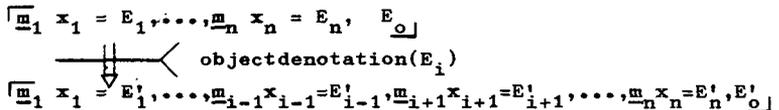
2.2.2 Transformationsregeln zur Berechnung rekursiver Funktionen

Für die Beschreibung der parallelen Berechnungsregel verwenden wir die folgenden Transformationsregeln:

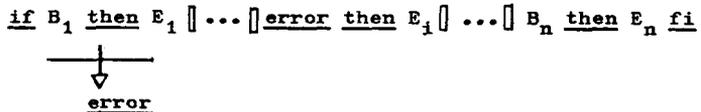
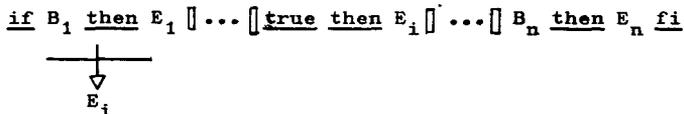
(UF) UNFOLD für Funktionen



(UO) UNFOLD für Objekte (sei $E'_k = E_k[E_i/x_i]$, wobei $1 \leq i \leq n$)



(CH) CHOICE in bewachten Ausdrücken



Man beachte, daß in (UF) nach unserer Konvention $\neg \text{occurs}(x_i \text{ in } E_j)$ gilt. Das Prädikat "objectdenotation" liefert für den Ausdruck "error" false.

Lemma: Die Regeln (UF), (UO) und (CH) sind korrekt.

Beweis: Die Paralleldeklaration in (UF) ist äquivalent zu einer Kollektivdeklaration und damit auf eine definierende Transformation in /Broy et al. 78b/ zurückführbar.

Die Regel (UO) entspricht genau der Substitution, die bei der Definition der Funktionsapplikation in /Broy et al. 78b/ verwendet wird. Die Regel (CH) folgt aus der Definition der bewachten Ausdrücke. □

2.2.3 Eine nichtdeterministische Berechnungsregel

Für die Berechnung des Resultats einer unter Umständen nicht-determinierten, rekursiv definierten Funktion definieren wir eine nichtdeterministische Berechnungsregel:

Definition: Die Berechnungsregel (P) wertet einen Aufruf einer rekursiv definierten Funktion in folgender Weise aus:

- (UF) Wende die Transformationsregel (UF) auf Aufrufe der rekursiven Funktion an, die nicht in den Alternativen bewachter Ausdrücke auftreten.
- (UO) Wende die Regel (UO) auf Objektdeklarationen an, die nicht im Innern bewachter Ausdrücke auftreten.
- (SI) Vereinfache Ausdrücke, die nur primitive Bezeichnungen enthalten und nicht in den Alternativen bewachter Ausdrücke auftreten.
- (CH) Wende auf bewachte Ausdrücke, die selbst nicht in bewachten Ausdrücken auftreten, die Regel (CH) an.

Die obigen Schritte werden, soweit anwendbar, in beliebiger Reihenfolge angewendet, bis keiner der Schritte mehr anwendbar ist.

Man beachte, daß im Vereinfachungsschritt (SI) zu Objektnotationen übergegangen wird. Im (UF)-Schritt wird gegebenenfalls eine implizite Umbenennung (α -Reduktion) der gebundenen Bezeichnungen (formaler Parameter) eingeschlossen.

Die Berechnungsregel (P) ist nichtdeterministisch, da die Reihenfolge der Anwendung der einzelnen Schritte nicht vorgeschrieben wird. Die möglichst weitgehend parallele Anwendung der einzelnen Schritte auf Teilausdrücke des vorliegenden Ausdrucks nennen wir "Call-in-Parallel"¹⁾

Definition: Die Berechnungsregel (P) liefert eine Menge von Berechnungssequenzen. Wir sagen "eine Berechnung terminiert ordnungsgemäß", wenn nach der Anwendung endlich vieler Berechnungsschritte ein Objekt (eine Objektnotation) vorliegt.

¹⁾ Nicht zu verwechseln mit Mannas "Parallel-Innermost"-Regel

Theorem: Sei E ein Ausdruck, der außer primitiven Funktionen nur Aufrufe der rekursiven Funktion f enthält, d.h.

$$E' \hat{=} \overline{\text{funct}} f = (\underline{m}_1 x_1, \dots, \underline{m}_n x_n) \underline{r}: R; \underline{E}_j \in \text{EXP},$$

dann gilt:

- (i) die parallele Berechnungsregel berechnet nur Werte aus $B(E')$;
- (ii) gilt $d(E')$, so terminiert jede Rechnung nach der parallelen Berechnungsregel ordnungsgemäß;
- (iii) gilt $\neg d(E')$, so existiert eine nichtterminierende Berechnung.

Beweis: (i) folgt unmittelbar aus der Korrektheit der verwendeten Transformationsregeln.

(ii) folgt aus dem folgenden Lemma.

(iii) folgt aus dem darauffolgenden Lemma. □

Lemma: Sei f wie oben und k die Anzahl des Auftretens von f in R. Ferner sei die Folge f_i induktiv definiert:

$$\overline{\text{funct}} f_0 = (\underline{m}_1 x_1, \dots, \underline{m}_n x_n) \underline{r}: \text{error} \text{ und}$$

$$\overline{\text{funct}} f_{i+1} = (\underline{m}_1 x_1, \dots, \underline{m}_n x_n) \underline{r}: R[f_i/f],$$

dann gilt:

$$\forall (u_1, \dots, u_n) \in U(\underline{m}_1) \times \dots \times U(\underline{m}_n):$$

$d(f_i(u_1, \dots, u_n)) \Rightarrow$ In einer Berechnungssequenz für $f(u_1, \dots, u_n)$ nach der Regel (P) treten in jedem Fall höchstens k^i (UF)-Schritte auf.

Beweis: Induktion über i:

Für $i=0$ ist die Behauptung trivial, da $\neg d(f_0(u_1, \dots, u_n))$ gilt.

Sei $i \neq 0$ und die Behauptung richtig für $i-1$. Die Berechnung liefert im ersten Schritt (nur ein (UF) Schritt ist möglich)

für $f(u_1, \dots, u_n)$ den Block $\overline{\underline{m}_1 x_1 = u_1, \dots, \underline{m}_n x_n = u_n, \underline{R}}$, der wenn $d(f_i(u_1, \dots, u_n))$ gilt, äquivalent ist zu

$$\overline{\underline{m}_1 x_1 = u_1, \dots, \underline{m}_n x_n = u_n, R[f_{i-1}/f]}, \text{ da } d(f_i(u_1, \dots, u_n)) \Rightarrow f_i(u_1, \dots, u_n) = f(u_1, \dots, u_n).$$

Für jeden der in $R[f_{i-1}/f]$ auftretenden Ausdrücke der Form $f_{i-1}(E_1, \dots, E_n)$, der schließlich im Laufe der Rechnung durch einen (UF)-Schritt eliminiert wird, gilt $d(f_{i-1}(E_1, \dots, E_n) // \{x_1=u_1, \dots, x_n=u_n\})$, sonst ergäbe sich ein Widerspruch zu $d(f_{i-1}(u_1, \dots, u_n))$. Da nur k Ausdrücke der Form $f_{i-1}(\dots)$ existieren, liefert die Induktionsvoraussetzung die Behauptung. □

Lemma: Seien alle Definitionen wie eben, dann gilt

$V (u_1, \dots, u_n) \in U(\underline{m}_1) \times \dots \times U(\underline{m}_n)$:
 $\neg d(f(u_1, \dots, u_n)) \Rightarrow$ es existiert eine nicht
 ordnungsgemäß terminierende
 Berechnungssequenz der Regel (P).

Beweis: Alle (UO)-, (UF)- und (SI)-Schritte reduzieren die Breite nicht. Seien $\{C_i\}_{0 \leq i \leq k}$ Berechnungssequenzen der Regel (P) mit $C_0 = f(u_1, \dots, u_n)$. Wir beweisen durch Induktion:

für alle $k \in \mathbb{N}$ gilt: Es existiert eine Berechnungssequenz nach Regel (P), die nach höchstens k Schritten nicht ordnungsgemäß terminiert oder es existiert eine Berechnungssequenz mit mehr als k Gliedern mit $\neg d(C_k)$.

Für $k = 0$ ist die Aussage trivial. Sei also $k \geq 1$ und die Behauptung zutreffend für $k-1$. Es genügt den Fall zu betrachten: Es gibt eine Berechnungssequenz mit $k-1$ Gliedern und $\neg d(C_{k-1})$. Falls einer der Schritte (UO), (SI) oder (UF) auf C_{k-1} anwendbar ist, entsteht ein C_k mit $\neg d(C_k)$. Anderenfalls ist entweder keiner der Berechnungsschritte mehr anwendbar oder nur noch ein (CH)-Schritt. Im zweiten Fall kann der (CH)-Schritt aber sicher so angewendet werden, daß $\neg d(C_k)$ für den entstehenden Term C_k gilt. □

Die Berechnungsregel (P) stellt eine Verallgemeinerung der Herbrand-Kleene-Maschine zu einer nichtdeterministischen Textersetzungsmaschine dar.

Die Schritte (UF), (SI) und (CH) können völlig unabhängig nebeneinander, also parallel auf die einzelnen Ausdrücke in einem Term der Berechnungssequenz auf der rechten Seite von Paralleldeklarationen und den Resultatausdruck angewendet werden. Jedoch der (UO)-Schritt betrifft alle Ausdrücke und kann deshalb i.a. nicht parallel zu anderen Schritten ausgeführt werden. Der (UO)-Schritt erfordert damit eine Koordination.

Beispiel: Sei f rekursiv definiert durch

$$\text{funct } f = (\text{nat } x, \text{ nat } y) \text{ nat: } \begin{array}{l} \text{if } x = 0 \text{ then } y \\ \quad \square \quad x > 0 \text{ then } f(x-1, f(x-1, y+1)) \end{array} \text{ fi};$$

Wir kürzen den Rumpf der Funktion durch $R(x,y)$ ab. Für den Ausdruck $f(1, f(1, 0))$ liefert die Regel (P):

$$\overline{\text{nat } x1 = 1, \text{ nat } y1 = \overline{\text{nat } x2 = 1, \text{ nat } y2 = 0, R(x2, y2)}}, R(x1, y1)$$

Je ein (UO)-Schritt für $x1$, $x2$ und $y2$ mit anschließender Vereinfachung liefert:

$$\overline{\text{nat } y1 = \text{if false then } 0 \quad \square \quad \text{true then } f(1-1, f(1-1, 0+1)) \text{ fi},} \\ \text{if false then } y1 \quad \square \quad \text{true then } f(1-1, f(1-1, y1+1)) \text{ fi}$$

Ein (CH)-Schritt liefert mit anschließender Vereinfachung:

$$\overline{\text{nat } y1 = f(0, f(0, 1)), \quad f(0, f(0, y1+1))}$$

Zwei (UF)-Schritte ergeben:

$$\overline{\text{nat } y1 = \overline{\text{nat } x3 = 0, \text{ nat } y3 = f(0, 1), R(x3, y3)}}, \\ \overline{\text{nat } x4 = 0, \text{ nat } y4 = f(0, y1+1), R(x4, y4)}$$

(UO)-Schritte für $x3$ und $x4$ und zwei (UF)-Schritte ergeben:

$$\overline{\text{nat } y1 = \overline{\text{nat } y3 = \overline{\text{nat } x5 = 0, \text{ nat } y5 = 1, R(x5, y5)}},} \\ \text{if true then } y3 \quad \square \quad \text{false then } f(0-1, f(0-1, y3)) \text{ fi}, \\ \overline{\text{nat } y4 = \overline{\text{nat } x6 = 0, \text{ nat } y6 = y1+1, R(x6, y6)}}, \\ \text{if true then } y4 \quad \square \quad \text{false then } f(0-1, f(0-1, y4+1)) \text{ fi}}$$

(UO)-Schritte für $x5, y5$ und $x6$ und zwei (CH)-Schritte ergeben:

$$\begin{aligned} \overline{\text{nat}}\ y1 &= \overline{\text{nat}}\ y3 = \underline{\text{if true then 1}} \ \underline{\text{[] false then f(0-1, f(0-1, 1))}} \ \underline{\text{fi}}, \\ &\quad y3 \\ \overline{\text{nat}}\ y4 &= \overline{\text{nat}}\ y6 = y1+1, \ \underline{\text{if true then y6}} \ \underline{\text{[] false then ... fi}}, \\ &\quad y4 \end{aligned}$$

Zwei (CH)-Schritte ergeben:

$$\overline{\text{nat}}\ y1 = \overline{\text{nat}}\ y3 = 1, \ \underline{y3}, \quad \overline{\text{nat}}\ y4 = \overline{\text{nat}}\ y6 = y1+1, \ \underline{y6}, \ \underline{y4}$$

Eine Kette von (U0)-Schritten ergibt:

$$\overline{\text{nat}}\ y1 = 1, \quad \overline{\text{nat}}\ y4 = \overline{\text{nat}}\ y6 = y1+1, \ \underline{y6}, \ \underline{y4}$$

$$\overline{\text{nat}}\ y4 = \overline{\text{nat}}\ y6 = 2, \ \underline{y6}, \ \underline{y4}$$

$$\overline{\text{nat}}\ y4 = 2, \ \underline{y4}$$

2

Man beachte, daß durch die Berechnungsregel (P) keine Argumentausdrücke dupliziert werden und somit eine Ineffizienz der Arbeit, wie zum Beispiel bei der "Full Computation Rule", vermieden wird.

Die Berechnungsregel (P) ließe sich auch so modifizieren, daß sie sich mathematisch äquivalent zur Call-by-Name Regel verhält, indem man die Transformationsregel

$$\overline{\underline{m}}_1\ x_1 = E_1, \dots, \overline{\underline{m}}_n\ x_n = E_n, \ \underline{E}$$



objectdenotation(E)

E

in das Repertoire der Berechnungsschritte aufnimmt. In dieser Weise erhielte man eine Berechnungsregel, die der "Delayed Evaluation" von Vuillemin stark ähnelt. Eine determinierte Auswahlstrategie für die einzelnen Berechnungsschritte kann schließlich genau eine Version der "Delayed Evaluation" liefern.

Natürlich ist die obige Regel in der "Call-by-Value" Semantik von CIP-L nicht korrekt. Man beachte, daß diese Regel den Abbruch aller parallel ablaufenden Prozesse bedeutet, wenn der "Hauptprozeß" erfolgreich abgeschlossen ist, wohingegen

der "Call-by-Value"-Semantik die übliche Auffassung paralleler Prozesse entspricht, wo ein System parallel ablaufender Programme genau dann terminiert, wenn jedes einzelne der Programme terminiert. Für eine weitergehende Behandlung paralleler Prozesse, bei denen man an der Terminierung genau eines Prozesses interessiert ist, vgl. Abschnitt 3.3.

Daß die Berechnungsregel (P) bei voller Ausschöpfung ihrer parallelen Möglichkeiten auch für einfache Beispiele schon eine recht weitgehende Parallelisierung bedeuten kann, zeigt die folgende rekursive Funktion "op", die auf zwei durch Sequenzen repräsentierten Vektoren eine komponentenweise Anwendung der Operation " f " vornimmt (Sei dabei "&" die Konkatenation).

```

funct op = (sequ a, sequ b: length(a) = length(b))sequ:
    if isempty(a) then ()
        else (top(a) & top(b)) & op(rest(a),rest(b)) fi

```

Ein Ablauf e von op(a,b) mit length(a) = length(b) = n hat die Form:

```

sequ a0 = a, sequ b0 = b,
    if ¬isempty(a0) then (top(a0) & top(b0)) &
sequ a1 = rest(a0), sequ b1 = rest(b0),
    if ¬isempty(a1) then (top(a1) & top(b1)) &
        .
        .
        .
sequ an = rest(an-1), sequ bn = rest(bn-1),
    if isempty(an) then ()
        fi . . . fi

```

Man beachte, daß für vernachlässigbar kleine Ausführungszeiten für die primitiven Funktionen "&", "rest" und "isempty" die komponentenweise Verknüpfung praktisch parallel ausgeführt wird. Sind die Zeiten für die Ausführung der Funktionen "&", "rest" und "isempty" konstant gleich t_u , t_r bzw. t_i und für " f " gleich t_o , so gilt nämlich $\text{time} = (n+1) \cdot (t_r + t_i + t_u) + t_o$, d.h. nur die Konstruktion und Selektion wird entsprechend der Struktur der Sequenzen sequentiell ausgeführt.

2.2.4 Ablaufordnungen und Effizienzmaße für die Paralleldeklaration

Die in Kapitel 1 definierten Begriffe der Ablaufordnungen und Effizienzmaße werden nun auf Paralleldeklarationen ausgedehnt. Für Abläufe der Form

$$\overline{\underline{m}} x = b, \underline{e}_j$$

vereinbaren wir: $\forall \text{act}(c) \in \text{actions}(e)$:

$$\text{occurs}(\underline{x} \text{ in } c) \Rightarrow \text{act}(c) \in_z \text{act}(\underline{m} x = b)$$

Damit wird gefordert, daß jede Aktion, in der die Bezeichnung x auftritt, d.h. der Wert von x benötigt wird, frühestens mit dem Abschluß der Berechnung von x beendet werden kann.

Entsprechend der parallelen Berechnungsregel erhalten wir für die Effizienzmaße aus Kapitel 1:

$$\begin{aligned} \text{processor}(\underline{m} x = b, e) &=_{\text{def}} \text{processor}(b) + \text{processor}(e), \\ \text{ptime}(b, e) &=_{\text{def}} \text{ptime}(b) + \text{ptime}(e). \end{aligned}$$

Damit unterscheiden sich diese beiden Effizienzmaße in keiner Weise von der nichtparallelen Deklaration $\overline{\underline{m}} x = b; \underline{e}_j$.

Komplizierter wird die Definition für das Effizienzmaß "time". Um das Maß definieren zu können führen wir einen "Zeitbonus" ein. So bedeutet

$$e \underline{\text{bon}} t,$$

daß die Auswertung von e schon um t Zeiteinheiten fortgeschritten ist. Dementsprechend definieren wir:

$$\text{time}(e \underline{\text{bon}} t) =_{\text{def}} \max \{ \text{time}(e) - t, 0 \}$$

Damit lassen sich nun die Auswertungszeiten eines definierten Ablaufs e mit Paralleldeklarationen beschreiben, falls $d(e)$ gilt. Für Paralleldeklarationen, die nicht zykliefrei sind und somit unendliche Wartezustände beinhalten, verzichten wir auf die Definition einer Auswertungszeit $\text{time}(e)$.

$$\text{time}(\overline{\underline{m}}_1 x_1 = e_1, \dots, \underline{m}_{n-1} x_{n-1} = e_{n-1}, \overline{\underline{m}}_n x_n = e_n, \underline{e}_{\underline{j}}) = \text{def}$$

$$\text{time}(\overline{\underline{m}}_1 x_1 = e_1, \dots, \underline{m}_n x_n = e'_n, \underline{e}_{\underline{j}}),$$

$$\text{wobei } e'_n = \overline{\underline{m}}_{j_1} x_{j_1} = e_{j_1}, \dots, \underline{m}_{j_k} x_{j_k} = e_{j_k}, \underline{e}_{\underline{j}},$$

$$\{j_1, \dots, j_k\} = \{i, 1 \leq i \leq n: \text{occurs}(x_i \text{ in } e_n)\},$$

$$\text{und } \forall i, i', 1 \leq i < i' \leq k: j_i < j_{i'}.$$

$$\text{time}(\overline{\underline{m}}_1 x_1 = e_1, \dots, \underline{m}_n x_n = e_n, (b_1, \dots, b_m)_{\underline{j}}) = \text{def}$$

$$\text{time}((b'_1, \dots, b'_m)),$$

$$\text{wobei } b'_i = \overline{\underline{m}}_1 x_1 = e_1, \dots, \underline{m}_n x_n = e_n, \underline{b}_{\underline{j}}.$$

$$\text{time}(\overline{\underline{m}}_1 x_1 = e_1, \dots, \underline{m}_n x_n = e_n, f(b_1, \dots, b_m)_{\underline{j}}) = \text{def}$$

$$\text{time}(f(b'_1, \dots, b'_m)),$$

wobei b'_i definiert sei wie eben.

$$\text{time}(\overline{\underline{m}}_1 x_1 = e_1, \dots, \underline{m}_n x_n = e_n, \underline{\text{if } b \text{ then } e \text{ fi}}_{\underline{j}}) = \text{def}$$

$$\text{time}(\underline{\text{if } b' \text{ then } \underline{m}}_1 x_1 = e'_1, \dots, \underline{m}_n x_n = e'_n, \underline{e \text{ fi}})$$

$$\text{wobei } b' = \overline{\underline{m}}_{j_1} x_{j_1} = e_{j_1}, \dots, \underline{m}_{j_k} x_{j_k} = e_{j_k}, \underline{b}_{\underline{j}},$$

$$\{j_1, \dots, j_k\} = \{i, 1 \leq i \leq n: \text{occurs}(x_i \text{ in } b)\},$$

$$\text{und } \forall i, i', 1 \leq i < i' \leq k: j_i < j_{i'},$$

$$e'_i = e_i \underline{\text{bon}} \text{time}(b').$$

$$\text{time}(\underline{m}_1 x_1 = e_1, \dots, \underline{m}_n x_n = e_n, x_i) = \text{def}$$

$$\max\{\text{time}(e_1), \dots, \text{time}(e_n)\}$$

Betrachten wir unser Beispiel aus Abschnitt 2.1.2, so liefert das Programm $e \cong$

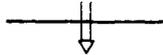
$$\overline{\underline{m}} y = f(x), \underline{m}_1 y_1 = h_1(x), \underline{m}_2 y_2 = h_2(x), (g_1(y, y_1), g_2(y, y_2))_{\underline{j}}$$

für alle drei Effizienzmaße die günstigsten Werte.

2.2.5 Zyklisches Warten und Verklemmung

Die durch UNFOLD-Schritte aus Funktionsaufrufen entstehenden Paralleldeklarationen haben die Eigenschaft, daß die parallel deklarierten Bezeichnungen in den definierenden Ausdrücken nicht auftreten. Betrachtet man jedoch die Transformationsregel (die sich aus der Komposition von Funktionen ergibt):

$$\underline{m} \ x = \overline{m}_1 \ x_1 = E_1, \dots, \underline{m}_n \ x_n = E_n, \underline{E}$$

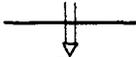


$$\underline{m}_1 \ x_1 = E_1, \dots, \underline{m}_n \ x_n = E_n, \underline{m} \ x = E$$

so können die geschachtelten Paralleldeklarationen, die geschachtelten Funktionsaufrufen entsprechen, in eine einzige Paralleldeklaration verwandelt werden. Nun besteht aber eine implizite Hierarchie unter den parallel angeordneten Deklarationen, die statisch feststellbar ist. Man beachte, daß auch hier wieder vorausgesetzt wird, daß keinerlei Bezeichnungskonflikte auftreten.

Die Reihenfolge der einzelnen Deklarationen in der Paralleldeklaration ist unerheblich, wie die Ablaufdefinition in 2.2.1 zeigt. Somit gilt die Transformationsregel

$$\underline{m} \ x = E, \underline{n} \ y = R$$



$$\underline{n} \ y = R, \underline{m} \ x = E$$

Außer den auf Funktionsaufrufe zurückführbaren Paralleldeklarationen gibt es auch andere z. B. der Form

$$\begin{aligned} \overline{m} \ x &= \underline{\text{if } B \text{ then } E \text{ else } T(y) \text{ fi}}, \\ \underline{n} \ y &= \underline{\text{if } B' \text{ then } E' \text{ else } T'(x) \text{ fi}}, \underline{g(x,y)} \end{aligned}$$

In diesem Ausdruck ist statisch keine Hierarchie erkennbar. Allerdings läßt sich die parallele Berechnungsregel (P) ohne Probleme auf solche Ausdrücke anwenden, ebenso die Definition von Abläufen in 2.2.1. Gilt beispielsweise $V(B) = \underline{\text{true}}$ und $V(B') = \underline{\text{false}}$, so tritt "dynamisch" eine Hierarchie auf.

Tritt jedoch bei der Abarbeitung solcher Ausdrücke auch "dynamisch" keine Hierarchie auf, d.h. endet die Berechnung mit einem Ausdruck z.B. der Form

$$\overline{m} x = y, \underline{m} y = x, g(x, y);$$

so spricht man von einer zyklischen Wartesituation oder von einer Verklemmung.

Definition: Ein Block E der Form $\overline{m}_1 x_1 = E_1, \dots, \underline{m}_n x_n = E_n, E_0$ heißt verklemmungsfrei in der Umgebung ENV, wenn gilt

$$\exists i, 1 \leq i \leq n: \forall e \in A(E_i, ENV \cup \{x_1 = \text{error}, \dots, x_n = \text{error}\}): \\ \forall j, 1 \leq j \leq n: \neg \text{occurs}(x_j \text{ in } e)$$

und

$$\overline{m}_1 x_1 = E_1, \dots, \underline{m}_{i-1} x_{i-1} = E_{i-1}, \\ \underline{m}_{i+1} x_{i+1} = E_{i+1}, \dots, \underline{m}_n x_n = E_n, E_j$$

verklemmungsfrei in der Umgebung

$$ENV \cup \{x_j = e \notin ENV\} \text{ ist.}$$

oder $n = 0$ und E_0 ist verklemmungsfrei in ENV.

Jede durch UNFOLD aus einem Funktionsaufruf entstandene Paralleldeklaration ist trivialerweise verklemmungsfrei und bleibt es auch nach Anwendung der Transformationsregeln am Anfang dieses Abschnitts.

Bei der parallelen Berechnungsregel kann man sich von der Vorstellung leiten lassen, daß die Folge der teilweise parallel stattfindenden Aktionen zur Auswertung von Ausdrücken durch den Fluß der erarbeiteten Werte für die Objektbezeichnungen bestimmt wird. Deshalb kann man auch von "Data Flow Programming" sprechen, wie das in /Kosinski 73/, /Dennis 74/, /Kosinski 77/ und /Arvind et al. 77/ geschieht. Dabei werden die "Data Flow Programs" häufig tatsächlich in Form von Datenflußplänen dargestellt. Ein Ansatz für eine denotationelle Semantikdefinition für solche Programme findet sich in /Plotkin 79/.

2.3 Parallele Berechnung und Programmentwicklung

Nun sollen Anwendung und Bedeutung der im letzten Abschnitt eingeführten Sprachelemente in der Programmentwicklung diskutiert werden. Insbesondere werden Fragen der parallelen Berechnung für die Kollektivdeklaration, die sequentielle Deklaration und die Paralleldeklaration untersucht. Auch die Frage der parallelen Verarbeitung der bewachten Ausdrücke wird noch einmal aufgegriffen. Abschließend wird ein Beispiel für einen Ablauf eines Programms mit Paralleldeklaration gegeben.

2.3.1 Sequentielle, kollektive und parallele Deklaration

Um die Unterschiede zwischen den verschiedenen Möglichkeiten der Deklarationen klar herauszuarbeiten, betrachten wir die Programmausdrücke:

- (P1: Parallel) $\overline{\overline{m_1} x_1 = E_1, \dots, \overline{m_n} x_n = E_n, E_j}$
 (P2: Kollektiv/Parallel) $\overline{\overline{m_1} x_1, \dots, \overline{m_n} x_n} = (E_1, \dots, E_n), E_j$
 (P3: Kollektiv) $\overline{\overline{m_1} x_1, \dots, \overline{m_n} x_n} = (E_1, \dots, E_n); E_j$
 (P4: Sequentiell) $\overline{\overline{m_1} x_1 = E_1; \dots; \overline{m_n} x_n = E_n; E_j}$

und die Kontextbedingungen (wobei (K2) aus (K1) folgt):

- (K1) $\forall i, j, 1 \leq i, j \leq n: \neg \text{occurs}(x_j \text{ in } E_i)$
 (K2) $\forall i, j, 1 \leq i \leq j \leq n: \neg \text{occurs}(x_j \text{ in } E_i)$

Für die syntaktische Korrektheit von Programm (P2) und Programm (P3) ist die Kontextbedingung (K1) notwendig, für die syntaktische Korrektheit von Programm (P4) die Kontextbedingung (K2). Gilt (K2), so ist für die Paralleldeklaration in Programm (P1) das Hierarchiekriterium erfüllt und die Paralleldeklaration ist verklemmungsfrei.

Nehmen wir nun an, daß die Bedingung (K1) erfüllt ist. Dann sind alle vier Programme mathematisch äquivalent und können bzgl. ihrer Effizienz verglichen werden.

Sei $t_i = \text{time}(E_i)$ und $t = \text{time}(E // \{x_1=e_1, \dots, x_n=e_n\})$ wobei $e_i = V(E_i)$, die Ausdrücke E_i seien als determiniert vorausgesetzt, dann gilt

$$\text{time}(P1) = \max \{t_1, \dots, t_n, \text{time}(E')\}$$

$$\text{time}(P2) = \max \{t_1, \dots, t_n, \text{time}(E'')\}$$

$$\text{time}(P3) = \max \{t_1, \dots, t_n\} + t$$

$$\text{time}(P4) = t_1 + \dots + t_n + t$$

wobei

$$E' \hat{=} E[E_1/x_1, \dots, E_n/x_n]$$

$$E'' \hat{=} E[\overline{D}; x_1/x_1, \dots, \overline{D}; x_n/x_n]$$

$$D \hat{=} (\underline{m}_1 x_1, \dots, \underline{m}_n x_n) = (E_1, \dots, E_n)$$

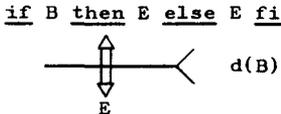
Die Prozessorengesamtzeit ptime bleibt für alle vier Programme gleich.

Da die Paralleldeklaration die liberalste Kontextbedingung hat, ist der Übergang von der sequentiellen oder kollektiven Deklaration ohne Einschränkung trivial möglich. Umgekehrt kann von der Paralleldeklaration zur sequentiellen oder kollektiven Deklaration nur nach sorgfältiger Prüfung der Kontextbedingungen (K2) bzw. (K1) übergegangen werden. Die Bedingung (K2) kann in manchen Fällen durch mehrmaliges Anwenden der Transformationsregel aus Abschnitt 2.2.5, also durch Umordnen der Deklarationen erreicht werden. Existiert jedoch keine statisch erkennbare Hierarchie für die Ausdrücke auf der rechten Seite der Paralleldeklaration, so ist ein Übergang höchstens nach einer Reihe komplexerer Transformations-schritte möglich.

2.3.2 Parallele Berechnung und bewachte Ausdrücke

Die Definition der Abläufe in 1.1.1 unterstellt eine streng sequentielle Abarbeitung bewachter Ausdrücke in einer nicht determinierten Reihenfolge im folgenden Sinn. Es wird einer der Wächter ausgewählt und ausgewertet. Liefert die Auswertung true, dann wird mit der Auswertung der dazugehörigen Alternative fortgefahren. Ergibt die Auswertung false, dann wird ein anderer Wächter gewählt und ausgewertet. Terminiert die Auswertung nicht oder sind alle Wächter zu false ausgewertet, so terminiert die Auswertung des bewachten Ausdrucks nicht.

In der nichtdeterministischen Berechnungsregel (P) kann zumindest die Auswertung der Wächter eines bewachten Ausdrucks parallel vorgenommen werden. Liefert bei dieser parallelen Auswertung einer der Wächter true, können alle anderen Berechnungen der Wächter abgebrochen werden, und es kann mit der Auswertung der dazugehörigen Alternative fortgefahren werden. Aber selbst die Auswertung der Alternativen kann parallel zu der Auswertung der Wächter erfolgen. Ein solches Vorgehen wird zum Beispiel in /Friedman, Wise 78/ vorgeschlagen, um die in /McCarthy 63/ diskutierte schwache Äquivalenz



in eine starke Äquivalenz, d.h. in eine unbedingte Äquivalenz, zu verwandeln.

Gibt es auch wenig Motivation von der Programmentwicklung für die Sicherstellung der Korrektheit der obigen Regel ohne die Nebenbedingung, so kann im Zusammenhang mit Echtzeitproblemen eine parallele Auswertung von Wächtern und Alternativen von Bedeutung sein, um eine minimale Auswertungszeit zu gewährleisten. Ein entsprechendes Sprachelement wird im Abschnitt 3.3.4 behandelt.

2.3.3 Teilberechnungen

Anhand eines einfachen Programmschemas, das durch "Einbettung" aus bestimmten, linear rekursiven Programmen gewonnen werden kann (vgl. /Bauer et al. 78d/), wollen wir nun noch einmal einige Aspekte der Parallelverarbeitung rekursiver Programme erörtern. Wir betrachten:

funct f = (m x, n y) r: if B(x) then f(h(x), g(y,x))
else T(y) fi

Terminierende Aufrufe f(E1,E2) besitzen Abläufe der Form:

m x₁ = E1, n y₁ = E2,
if B(x₁) then m x₂ = h(x₁), n y₂ = g(y₁,x₁),
.
.
if B(x_{n-1}) then m x_n = h(x_{n-1}), m y_n = g(y_{n-1},x_{n-1}),
if ¬ B(x_n) then T(y_n) fi fi ... fi,

falls wir uns auf die Parallelerdeklaration abstützen. Die Elimination der Wächter führt auf:

m x₁ = E1, n y₁ = E2,
m x₂ = h(x₁), n y₂ = g(y₁,x₁),
.
.
m x_n = h(x_{n-1}), n y_n = g(y_{n-1},x_{n-1}), T(y_n) fi,

und nach UNFOLD sämtlicher Objektdeklarationen auf:

T(g(g(... g(g(E2, E1), h(E1)) ... , hⁿ⁻²(E1)), hⁿ⁻¹(E1))).

Die Berechnung der x_i kann unabhängig von der Berechnung der y_i erfolgen. Dagegen gehen die x_i in die Berechnung der y_i ein. Die parallele Auswertung von Aufrufen von f entspricht dem klassischen Verhalten des Erzeuger/Verbraucher-Problems. Der eine Prozeß erzeugt die "n Einheiten" x₁, der andere Prozeß verbraucht sie und berechnet das Resultat des Aufrufs (vgl. dazu Abschnitt 5.5).

Man beachte, daß in der Funktion f die Form der Abläufe, d.h. der Wert von n, nur vom Parameter x abhängt. Dies erlaubt auch eine besondere Form der Teilberechnung bei "Teilinstantiierung" (vgl. "Mixed Computations" in /Ershov 78/ und /Bauer, Wössner 79/).

3. Kommunikation zwischen parallel ablaufenden Programmen

In diesem Kapitel werden komplexere Möglichkeiten zur Kommunikation zwischen parallel ablaufenden, applikativen Programmen definiert. In Kapitel 2 ist nur eine sehr eingeschränkte Möglichkeit für eine solche Kommunikation gegeben, denn eine Kommunikation findet genau dann statt, wenn der Sender der Information (der Prozeß, der das zu deklarierende Objekt berechnet) bereits seine Aktivität beendet hat, d.h. wenn der Prozeß der Berechnung des Objekts abgeschlossen ist.

Jetzt werden Sprachelemente eingeführt, die die Deklaration der Bezeichnung und die Bindung an ein bestimmtes Objekt aufspalten. Diese Sprachelemente lassen sich verallgemeinern zu "Kommunikationstabellen", die eine unbeschränkte Anzahl von Objekten aufnehmen und für Kommunikationszwecke bereithalten können. Auch dieses Sprachelement ist von applikativer Natur.

Darüber hinaus werden nichtstrikte Sprachelemente zur Beschreibung multipler Wartesituationen als Modifikation der endlichen Auswahl betrachtet. Abschließend wird eine Reihe von Beispielen behandelt, um die Mächtigkeit der definierten Mechanismen zu demonstrieren und ebenso die Verwendung applikativer Sprachen für die parallele Programmierung zu zeigen.

3.1 Resultatbezeichnungen

Trennt man die Deklaration einer Objektbezeichnung von der Bindung der Bezeichnung an ein Objekt, so entsteht eine Resultatbezeichnung. Es wird dadurch möglich, die Bindung an ein Objekt im Lauf einer Berechnung vorzunehmen, und trotzdem in anderen, parallel dazu ablaufenden Teilen auf das Objekt mit Hilfe der Resultatbezeichnung Bezug zu nehmen. Wird auf die Resultatbezeichnung Bezug genommen, bevor sie eine Bindung an ein Objekt erfahren hat, so wird die Auswertung des entsprechenden Ausdrucks zurückgestellt, bis die Bindung erfolgt ist. Erfährt eine Resultatbezeichnung keine oder mehrere Bindungen, so ist das Programm fehlerhaft und liefert keinen definierten Wert.

Für Resultatbezeichnungen ist somit, ähnlich wie bei der Paralleldeklaration, eine Hierarchie der Definitionen erforderlich, um ein definiertes Ergebnis zu erzeugen. Die Werte der Resultatbezeichnungen ergeben sich aus der Hierarchie und sind vom Standpunkt des Benutzers eines der Werte konstant, wenn sie sich auch erst "dynamisch" aufbauen. Damit sind Resultatbezeichnungen ein applikatives Sprachelement.

Das Konzept der Resultatbezeichnung findet sich bereits - eingeschränkt auf Prozeduren und sequentielle Verarbeitung - in ALGOL 58. Auf Grund der Probleme, die im Zusammenhang mit Programmvariablen (vgl. Abschnitt 4.2.3) in parallelen Programmen entstehen, wurde in /Tesler, Enea 68/ eine Beschränkung auf sogenannte "Single-Assignment-Variables" vorgeschlagen (vgl. auch /ONERA CERT 78/).

Wir wollen die Semantik von Resultatbezeichnungen auf zwei unterschiedliche, aber natürlich konsistente Weisen erklären. Einmal wählen wir eine deskriptive Möglichkeit durch Komprehension über eine Menge von Abläufen, zum anderen verwenden wir definierende Transformationsregeln.

3.1.1 Syntax und informelle Beschreibung von Resultatbezeichnungen

Wie bereits im Abschnitt 2.2 angedeutet, ist der Übergang von der Kollektivdeklaration zur Paralleldeklaration (siehe die Transformationsregel in Abschnitt 2.2.1) nicht ohne weiteres möglich, wenn auf der rechten Seite der Kollektivdeklaration statt der expliziten Angabe eines Tupels eine Funktionsapplikation oder ein bewachter Ausdruck mit einem Tupel als Wert auftritt. Ist ein solcher Ausdruck auch noch nicht determiniert, so ist z.B. ein Aufbrechen der bewachten Ausdrücke in ein Tupel von bewachten Ausdrücken nicht korrekt.

Um dieses Dilemma zu beseitigen und um in einem Prozeß, der mehrere Objekte (ein Tupel von Objekten) berechnet, Wert für Wert unmittelbar nach Abschluß seiner Berechnung zur Verfügung stellen zu können (unabhängig davon, ob alle anderen Werte bereits vorliegen), trennen wir Einführung einer Bezeichnung und Bindung der Bezeichnung an ein spezielles Objekt:

<u>result</u> <u>m</u> x	deklariert eine Bezeichnung x für ein Objekt der Art <u>m</u>
x <u>is</u> E	bindet die Bezeichnung x unauflöslich an ein Objekt aus B(E) oder an undefiniert, falls $\neg d(E)$; im zweiten Fall ist die Aktion äquivalent zu <u>abort</u> .

Solche Bezeichnungen nennen wir Resultatbezeichnungen. Sie können wie andere Objektbezeichnungen verwendet werden. Eine Bindung darf an eine Resultatbezeichnung genau einmal erfolgen. Anderenfalls ist das gesamte Programm nicht definiert.

In Kommunikationsfunktionen, abgekürzt comfunct, können Resultatbezeichnungen auch als Parameter auftreten. Dann müssen natürlich die aktuellen Parameter auch Resultatbezeichnungen sein. Kommunikationsfunktionen liefern kein Ergebnis ab, sondern nehmen Bindungen an Resultatbezeichnungen vor (vgl. Resultatparameter in /Bauer, Wössner 79/).

Diese Entscheidung steht im Einklang mit der folgenden Vereinbarung:

Konvention: (Kontextbedingung)

In einem Ausdruck dürfen keine Bindungen an global deklarierte Resultatbezeichnungen erfolgen.

Durch diese Konvention wird weiterhin die Freiheit der Ausdrücke von Effekten garantiert.

Wir wollen nun zuerst ein Beispiel betrachten, um die Verwendung von Resultatbezeichnungen zu demonstrieren, bevor wir die Semantik der Resultatbezeichnungen formal definieren.

Beispiel: Ganzzahlige Division mit Rest (vgl. /Bauer, Wössner79//)

```

┌ funct modiv = ( nat a, nat b : b > 0 ) ( nat, nat ):
  if a < b then ( 0, a )
                else ( nat d, nat m ) = modiv( a-b, b ),
                    ( d+1, m )                                     fi,
  ( nat d, nat m ) = modiv( E1, E2 ),   E                       └

```

Geht über in¹⁾

```

┌ comfunct comodiv = ( nat a, nat b : b > 0, result nat d, result nat m ):
  if a < b then ( d, m ) is ( 0, a )
                else result nat dd,
                    comodiv( a-b, b, dd, m ),
                    d is dd+1                                     fi,
  result nat d, result nat m,   comodiv( E1, E2, d, m ),   E └

```

In der zweiten Version steht das Objekt unter der Bezeichnung m, d.h. der Rest der Division, im Ausdruck E sofort nach Erreichen der Terminierungsbedingung zur Verfügung, in der ersten muß man erst noch das "Zurückklappern" der Rekursion abwarten, bis der bereits vorliegende Wert für m tatsächlich verwendet werden kann.

¹⁾ In /Bauer, Wössner 79/ findet sich die Notation

funct comodiv = (nat a, nat b : b > 0) ───► (nat d, nat m)

3.1.2 Formale Definition der Semantik von Resultatbezeichnungen

Wir ordnen nun Programm(ausdrück)en, in denen Resultatbezeichnungen verwendet werden, eine Semantik zu, indem wir Abläufe für solche Programme definieren. Da sich die Definition von Abläufen für "bewachte Aktionen" und Blöcke von Aktionen praktisch nicht von der Definition von Abläufen für bewachte Ausdrücke und Blöcke, die Ausdrücke verkörpern, unterscheidet, verzichten wir hier auf eine explizite Definition.

Eine deklarierte Resultatbezeichnung der Art result m erfährt erst im Lauf der weiteren Rechnung eine Bindung an ein Objekt aus $U(\underline{m}) \cup \{\ast\}$. Falls keine Bindung vorgenommen wird, ist der Wert des gesamten Ausdrucks, in dem die Resultatbezeichnung deklariert ist, äquivalent zu error, bzw. die gesamte Aktion ist äquivalent zu abort.

Um die Semantik des Blocks

result m x, R

zu definieren, betrachten wir zuerst die Menge aller denkbaren Werte e aus $U(\underline{m}) \cup \{\ast\}$, an die x gebunden werden könnte. Für jeden Wert e ist eine Abarbeitung von R möglich. Jeder durch die Annahme eines bestimmten Wertes e für die Resultatbezeichnung x entstehende Ablauf, genannt "Pseudoablauf", kann überprüft werden, ob er

- wertkonsistent ist, d.h. ob der für x angenommene Wert e auch in einer Aktion "x is E" an x gebunden wird;
- zeitkonsistent ist, d.h. ob eine Ablaufordnung existiert, so daß der Wert von x erst nach seiner Bindung an x verwendet wird;
- keine mehrfachen Bindungen von x an Werte enthält.

Wir definieren nun die Menge $A'(E, ENV)$ von Pseudoabläufen. Sei A' für alle bisher betrachteten Sprachelemente genauso definiert, wie für A .

$$A'(\overline{\text{result } m \ x, R}, ENV) =_{\text{def}} \bigcup_{\substack{e \in U(\underline{m}) \cup \{\omega\} \\ r \in A'(R, ENV \cup \{x = e\})}} \{\overline{\text{result } m \ x \ll e \gg, r}\}$$

$$A'(\overline{x \text{ is } D, R}, ENV) =_{\text{def}} \bigcup_{\substack{d \in A'(D, ENV) \\ r \in A'(R, ENV) \\ e = V(d/ENV)}} \{\overline{x \text{ is } d \ll e \gg, r}\}$$

Aufrufe von Kommunikationsfunktionen seien analog zur UNFOLD-Regel (Regel (7) im folgenden Abschnitt) jeweils durch ihre Rümpfe ersetzt.

Ein Pseudoablauf p in $A'(E, ENV)$ heißt

wertkonsistent, falls für jede auftretende Resultatbezeichnung x mit Vereinbarung result $m \ x \ll e \gg$ eine Bindung $x \text{ is } d \ll e' \gg$ existiert mit $e = e'$ oder falls $e = \omega$;

zeitkonsistent, falls eine totale Ablaufordnung existiert, so daß für jede Aktion a , in der die Resultatbezeichnung x auftritt, gilt: es existiert eine Aktion $\text{act}(x \text{ is } d)$ und $\text{act}(x \text{ is } d) \stackrel{E_z}{\leq} a$

fehlerhaft, falls er wertkonsistent ist, jedoch für eine Resultatbezeichnung x mehr als eine Bindung auftritt, oder falls für eine Resultatbezeichnung keine Bindung auftritt, obwohl für alle auftretenden Objektbezeichnungen die angenommenen Werte definiert sind.

Falls $p \in CA'(E, ENV)$ wertkonsistent, zeitkonsistent und fehlerfrei ist, dann nennen wir p zulässig.

Die Semantik eines zulässigen Pseudoablaufs p , mit $p \hat{=} \langle \text{result } m \ x \ll e \rangle, r \rangle$, ist erklärt durch

$$V(p) =_{\text{def}} V(r' [e/x])$$

wobei r' aus r entsteht, indem man die Aktion " x is $d \ll e \rangle$ " durch nop ersetzt.

Als Menge der Abläufe eines Programmterms, der Resultatzeichnungen enthält, definieren wir:

$$A(E, ENV) =_{\text{def}} \{p : p \in CA'(E, ENV) : p \text{ ist zulässig und } p \hat{=} \text{normalize}(p')\}$$

$$U \left\{ \begin{array}{ll} \{\underline{\text{error}}\} & \text{falls }] p \in CA'(E, ENV) : \\ & p \text{ ist fehlerhaft} \\ \emptyset & \text{sonst} \end{array} \right.$$

Dabei entfernt $\text{normalize}(p)$ in p lediglich alle Terme der Form $\ll e \rangle$. Für jeden zulässigen Ablauf $p \in CA'(E, ENV)$ gilt demnach:

$$A(\text{normalize}(p'), ENV) = \{p\}$$

Wie die Definition zeigt, tritt bei einer Resultatzeichnung keinerlei Überschreiben von Information auf. Wird eine Resultatzeichnung angewendet, d.h. wird der Wert einer Resultatzeichnung verwendet, so kann die Resultatzeichnung wie eine gewöhnliche Objektzeichnung benutzt und verstanden werden. Die obige Definition zeigt, daß eine Resultatzeichnung für jeden (zulässigen) Ablauf genau einen Wert besitzt. Damit können Resultatzeichnungen der applikativen Ebene zugerechnet werden.

Zur Erläuterung der Definition der Pseudoabläufe und Abläufe betrachten wir nun einige einfache Beispiele:

Beispiel 1: Sei $E_1 \hat{=} \overline{\text{result bool } x, x \text{ is true, if } x \text{ then } 1 \text{ else } 2 \text{ fi}} _$

Es gilt (mit $U(\text{bool}) = \{T, F\}$): $A'(E_1, \emptyset) =$

$$\left\{ \overline{\text{result bool } x \ll T \gg, x \text{ is true} \ll T \gg, \text{if } x \text{ then } 1 \text{ fi}} _ , \right. \\ \left. \overline{\text{result bool } x \ll F \gg, x \text{ is true} \ll T \gg, \text{if } \neg x \text{ then } 2 \text{ fi}} _ , \right. \\ \left. \overline{\text{result bool } x \ll \omega \gg, x \text{ is true} \ll T \gg \text{ if } x \text{ then error fi}} _ \right\}$$

Nur der erste Ablauf ist wertkonsistent, und da er gleichzeitig zeitkonsistent und alle Abläufe fehlerfrei sind, gilt:

$$A(E_1, \emptyset) = \{ \overline{\text{result bool } x, x \text{ is true, if } x \text{ then } 1 \text{ fi}} _ \}$$

Beispiel 2: Sei $E_2 \hat{=} \overline{\text{result bool } x, \text{if } x \text{ then } x \text{ is true else } x \text{ is false fi, } x} _$

Es gilt: $A'(E_2, \emptyset) =$

$$\left\{ \overline{\text{result bool } x \ll T \gg, \text{if } x \text{ then } x \text{ is true} \ll T \gg \text{ fi, } x} _ , \right. \\ \left. \overline{\text{result bool } x \ll F \gg, \text{if } \neg x \text{ then } x \text{ is false} \ll F \gg \text{ fi, } x} _ , \right. \\ \left. \overline{\text{result bool } x \ll \omega \gg, \text{if } x \text{ then abort fi, } x} _ \right\}$$

Die beiden ersten Abläufe sind nicht zeitkonsistent. Nur der dritte Ablauf ist zeit- und wertkonsistent. Alle drei Abläufe sind fehlerfrei. Damit gilt:

$$A(E_2, \emptyset) = \{ \overline{\text{result bool } x, \text{if } x \text{ then abort fi, } x} _ \}$$

Beispiel 3: Sei $E_3 \hat{=} \overline{\text{result bool } x, s \text{ is true, } x \text{ is false, } x} _$

Es gilt: $A'(E_3, \emptyset) =$

$$\left\{ \overline{\text{result bool } x \ll T \gg, x \text{ is true} \ll T \gg, x \text{ is false} \ll F \gg, x} _ , \right. \\ \left. \overline{\text{result bool } x \ll F \gg, x \text{ is true} \ll T \gg, x \text{ is false} \ll F \gg, x} _ , \right. \\ \left. \overline{\text{result bool } x \ll \omega \gg, x \text{ is true} \ll T \gg, x \text{ is false} \ll F \gg, x} _ \right\}$$

Keiner der drei Abläufe ist fehlerfrei. Deshalb gilt:

$$A(E_3, \emptyset) = \{ \text{error} \}$$

3.1.3 Transformationsregeln für Resultatbezeichnungen

Die folgenden Transformationsregeln dienen zur Entwicklung von Programmen mit Resultatbezeichnungen. Sie können analog zu /Pepper 79/ auch als definierende Transformationen verstanden werden.

(1) Aufspalten einer Deklaration

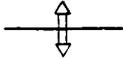
result m x, x is E



m x = E

(2) Vertauschbarkeit paralleler Bindungen

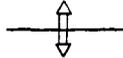
x is E, y is R



y is R, x is E

(3) Vertauschbarkeit von Deklarationen

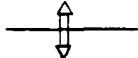
result m x, n y = E



n y = E, result m x

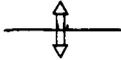
(4) Hereinziehen von Bindungen

x is $\overline{n_1} y_1 = E_1, \dots, \overline{n_k} y_k = E_k, \underline{E}$



$\overline{n_1} y_1 = E_1, \dots, \overline{n_k} y_k = E_k, x \underline{is} \underline{E}$

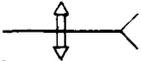
x is if B_1 then E_1 \square ... \square B_n then E_n fi



if B_1 then x is E_1 \square ... \square B_n then x is E_n else abort fi

(5) Elimination lokaler Resultatbezeichnungen

$\overline{\text{result } m \ x, D_1, x \text{ is } E, \quad y \text{ is } x, D_2}$

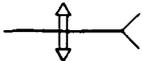


\neg occurs (x in E)

$\overline{D_1 [y/x], y \text{ is } E, D_2 [y/x]}$

(6) Auflösung von Kollektivbindungen

(x_1, \dots, x_n) is (E_1, \dots, E_n)



\neg occurs(x_i in E_j)

x_1 is E_1, \dots, x_n is E_n

(7) UNFOLD für Kommunikationsfunktionen

comfunct $f = (\underline{m}_1 y_1, \dots, \underline{m}_n y_n, \underline{\text{result}} \underline{q}_1 z_1, \dots, \underline{\text{result}} \underline{q}_k z_k) : A$

$f(E_1, \dots, E_n, x_1, \dots, x_k)$



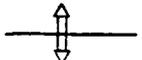
$\overline{\underline{m}_1 y_1 = E_1, \dots, \underline{m}_n y_n = E_n, A[x_1/z_1, \dots, x_k/z_k]}$

(8) Übergang von Funktions- zu Kommunikationsfunktionsaufrufen

funct $f = (\underline{m}_1 y_1, \dots, \underline{m}_n y_n)(\underline{q}_1, \dots, \underline{q}_k) : E$

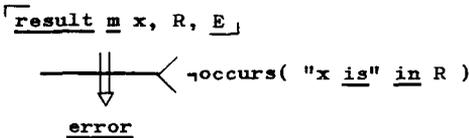
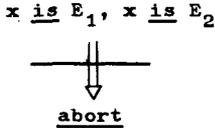
comfunct $cf = (\underline{m}_1 y_1, \dots, \underline{m}_n y_n, \underline{\text{result}} \underline{q}_1 z_1, \dots, \underline{\text{result}} \underline{q}_k z_k) :$
 $(z_1, \dots, z_k) \text{ is } E$

(x_1, \dots, x_k) is $f(E_1, \dots, E_n)$



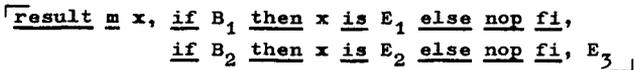
$cf(E_1, \dots, E_n, x_1, \dots, x_k)$

Die angegebenen Regeln reichen, zusammen mit Induktionsmethoden, aus, um definierte Programme zu entwickeln. Um gewisse Fehlersituationen auszudrücken, müssen noch weitere Regeln hinzugenommen werden, wie



Insbesondere stellt sich die Frage, ob nicht eine Reihe von Fehlersituationen statisch, d.h. durch restriktive Kontextbedingungen, ausgeschlossen werden sollte.

So kann man Programme, wie z.B.:



als unsicher verbieten, indem man jeweils in nur einer der parallel auszuführenden Aktionen Bindungen an die deklarierte Resultatbezeichnung x zuläßt. Die Ableitung eines Programms der obigen Form ist nur möglich, wenn man die Aufspaltung bewachter Aktionen in parallel auszuführende bewachte Aktionen als Transformationsregel zuläßt.

Resultatbezeichnungen lassen sich problemlos auch als Komponenten von Verbunden verwenden. Felder von Resultatbezeichnungen werfen die von Programmvariablen her bekannten Probleme (vgl. /Bauer, Wössner 79/) des "Aliasverbots" auf. Eine dazu alternative Konstruktion wird im folgenden Kapitel eingeführt.

3.1.4 Ableitung des Beispiels "modiv"

Mit Hilfe der eben eingeführten Transformationsregel sind wir nun in der Lage, das Beispiel aus 3.1.1 formal zu entwickeln. Sei "modiv" definiert wie in 3.1.1. Wir starten mit der Deklaration

(nat d, nat m) = modiv (A, B)

Regel (1) gibt:

result nat d, result nat m, (d, m) is modiv(A, B)

Regel (8) gibt:

result nat d, result nat m, comodiv (A, B, d, m)

where

cofunct comodiv = (nat a, nat b : b > 0, result nat d, result nat m):
(d, m) is modiv(a, b)

dabei gilt:

(d, m) is modiv (a, b)

ist nach UNFOLD für modiv und Regel (4) äquivalent zu

if a < b then (d, m) is (0, a)
else (nat dd, nat mm) = modiv(a-b, b),
(d, m) is (dd+1, mm) fi

FOLD für comodiv in der else-Alternative und Regel (5) liefert die Version in Abschnitt 3.1.1.

3.2 Kommunikationstabellen

Die im vorangegangenen Abschnitt eingeführten Resultatbezeichnungen erlauben genau einen "Kommunikationsvorgang" pro definierte Resultatbezeichnung. Nun werden Kommunikationstabellen eingeführt, die konsekutiv aufgebaut und komponentenweise über Indexzugriff gelesen werden können. Allerdings kann eine einmal eingetragene Information weder gelöscht, noch überschrieben werden. Kommunikationstabellen dürfen von parallel ablaufenden Programmen beschrieben werden. Konkurrierende Aktionen, die Einträge in die gleiche Tabelle vornehmen möchten, werden in nicht vorherbestimmter Weise sequentialisiert. Damit ist die Reihenfolge solcher Einträge in der Tabelle nicht determiniert.

In einer Kommunikationstabelle wird also ein bestimmter Indexbereich [1..n] mit Einträgen belegt. Jede Komponente dieses Bereichs kann als eine Resultatbezeichnung aufgefaßt werden, da ein Leseversuch des Wertes einer solchen Komponente zum Warten führt, bis der entsprechende Eintrag in der Tabelle vorgenommen worden ist.

Da die Belegung der Komponenten mit Einträgen unauflöslich ist, kann keinerlei Überschreiben von Information auftreten. Deshalb können Kommunikationstabellen noch der applikativen Ebene zugerechnet werden. Für die formale Definition von Kommunikationstabellen verwenden wir einerseits abstrakte Typen, die die Objekte "Kommunikationstabelle" beschreiben, andererseits setzen wir, wie eben schon für Resultatbezeichnungen, Pseudoabläufe ein, für die dann über bestimmte Prädikate die Konsistenz eines solchen Ablaufs charakterisiert wird. Wieder läßt sich so durch Komprehension über die Menge der Pseudoabläufe die Menge der tatsächlich möglichen Abläufe auszeichnen.

Die angewandte Technik läßt sich natürlich auch auf andere abstrakte Typen übertragen. Der Typ der Kommunikationstabellen wurde gewählt, da er, obwohl kein Überschreiben von Information auftritt, zum Beispiel schon ausreicht, um das Problem der "fünf dinierenden Philosophen" zu formulieren.

3.2.1 Informelle Beschreibung von Kommunikationstabellen

Eine Tabelle kann durch den abstrakten Typ TABLE charakterisiert werden. In der Notation folgen wir /Partsch, Broy 78/.

```
type TABLE = ( mode m ) table, vac, put, get:

  mode table,
  funct table vac,
  funct (table, m)table put,
  funct (table t, pnat n: length(t)  $\geq$  n )m get,
  funct (table)nat length,

  law: length( vac ) = 0,
  law: length( put( t, x) ) = length( t ) + 1,
  law: get(put(t, x), n) = if length(t) = n-1 then x
                                     else get(t,n) fi
                                     end of type
```

Der Typ TABLE ist hinreichend vollständig ("sufficiently complete") beschrieben. Er kann als Mischform aus Stapel und Feld verstanden werden (vgl. STACK, GREX und FLEX in /Bauer, Wössner 79/).

Eine Kommunikationstabelle ähnelt einer Resultatbezeichnung der Art table, die jedoch nicht durch eine einzige Bindungsoperation t is E an einen Wert der Art table gebunden wird, sondern komponentenweise durch konsekutive put-Aktionen aufgebaut wird. Eine Kommunikationstabelle wird vereinbart durch:

comtable m t

und um einen Elementeintrag verlängert durch:

put E on t.

Eine solche Aktion führt zum Übergang von t zu put(t,E). Ein Leseversuch get(t,n) führt im Falle length(t) < n zum Warten, bis der entsprechende Eintrag in der Tabelle vorliegt. Eine Kommunikationstabelle kann von parallel ablaufenden Programmen gelesen und mit Einträgen versehen werden und auch als Parameter auftreten. Die Funktion length steht als "hidden Function" nicht zur Verfügung, da length(t) eine Variable ist.

Um die parallele Ausführung von Aktionen auch in der Notation stärker hervorzuheben, werden wir nun häufig das Symbol "//" an Stelle des Kommas verwenden. Als Begrenzungssymbole gebrauchen wir $\overline{\quad}$ für "cobegin" oder "parbegin" und das Symbol $\underline{\quad}$ für "coend" oder "parend". Die folgende Transformationsregel definiert die Semantik dieser Symbole:

$$\frac{\overline{A_1}, \overline{A_2}, \dots, \overline{A_n}}{\underline{\quad} \overline{\quad} \underline{\quad}} \quad \underline{\quad} A_1 \quad \underline{\quad} A_2 \quad \underline{\quad} \dots \quad \underline{\quad} A_n \quad \underline{\quad}$$

wobei die A_i Aktionen und keine Ausdrücke seien. Trivialerweise sind damit $\overline{\quad}$ - und $\underline{\quad}$ -Klammern kommutativ und assoziativ.

Um die Verwendung von Kommunikationstabellen zu demonstrieren, betrachten wir nun das Erzeuger/Verbraucher-Problem in seiner einfachsten Form.

Beispiel: Erzeuger/Verbraucher (mit unendlichem Puffer)

```

comtable m buffer,
comfunct consumer = ( nat n, comtable m b ):
    consume(get(b,n)); consumer(n+1,b),
comfunct producer = ( comtable m b ):
    put product on b; producer(b),

 $\overline{\quad}$  consumer( 1, buffer )  $\underline{\quad}$  producer( buffer )  $\underline{\quad}$ 

```

Dabei sei product eine Funktion mit Wert der Art m und consume eine Aktion. Man beachte, daß das obige Programm nur unendliche Abläufe besitzt. Man kann das Beispiel auch auf endliche Abläufe umschreiben, indem man den producer die Anzahl der erzeugten Einheiten mitzählen läßt und an einer gewissen oberen Schranke den Prozeß abbricht.

Um auch weiterhin die Freiheit von Ausdrücken von Seiteneffekten garantieren zu können, was für die parallele Auswertung von Ausdrücken zwingend notwendig erscheint, fordern wir die Gültigkeit der folgenden - statisch nachprüfbaren - Konvention.

Konvention: In einem Ausdruck dürfen keine put-Aktionen auf eine global vereinbarte Kommunikationstabelle auftreten.

Kommunikationstabellen können als applikatives Gegenstück zu Warteschlangen, Kommunikationssequenzen oder Strömen (vgl. /Kahn 74/, /Kahn, McQueen 77/ und auch Abschnitt 5.1) aufgefaßt werden. Sie dokumentieren alle über sie stattfindenden "sendenden" Kommunikationsvorgänge und können deshalb als Sprachelemente bezeichnet werden, bei denen (im Gegensatz zu Programmvariablen) kein Überschreiben von Information auftritt.

Im Zusammenhang mit "Data Flow Programming Languages" werden auch unendliche Sequenzen von übertragenen Objekten betrachtet (vgl. /Plotkin 79/). Solche unendlichen Folgen treten in Verbindung mit nichtterminierenden Programmsystemen auf (vgl. Abschnitt 5.4).

Kommunikationstabellen können auch als Spezialfall eines Monitors (vgl. /Hoare 74/) oder eines Managers (vgl. /Jammel, Stiegler 77/) angesehen werden. Beispielsweise kann man Kommunikationstabellen durch einen Monitor implementieren.

Der in /Reed, Kanodia 79/ beschriebene Ansatz weist Ähnlichkeiten mit Kommunikationstabellen auf. So kann der dort vorgeschlagene "Eventcounter" als Spezialfall einer Kommunikationstabelle, die "Ticket"-Konstruktion als Sonderfall der nichtdeterministischen Sequentialisierung von parallel stattfindenden put-Aktionen interpretiert werden.

3.2.2 Formale Definition der Semantik von Kommunikationstabellen

Wir wenden die gleiche Technik an wie in Abschnitt 3.1.2. Für jede in einem Programm auftretende Kommunikationstabelle betrachten wir die Menge aller Kommunikationstabellen und prüfen, ob die entsprechenden Abläufe konsistent sind.

Sei dazu $U(\text{table } m)$ die Menge der Tabellen vom Typ $\text{TABLE}(m)$. Wir definieren wie in Abschnitt 3.1.2 Pseudoabläufe durch A' :

$$A'(\overline{\text{comtable}} \ c, \underline{R}_j, \text{ENV}) =_{\text{def}} \bigcup_{e \in U(\text{table } m)} \{ \overline{\text{comtable}} \ c \ll e \gg, \underline{r}_j \}$$

$$r \in A'(\overline{R}_j, \text{ENV} \cup \{c = e\})$$

$$A'(\text{get}(c, I), \text{ENV}) =_{\text{def}} \bigcup_{\substack{i \in A'(I, \text{ENV}) \\ j = V(i/\text{ENV})}} \text{get}(c, i) \ll j \gg$$

$$A'(\underline{\text{put}} \ E \ \underline{\text{on}} \ c, \text{ENV}) =_{\text{def}} \bigcup_{e \in A'(E, \text{ENV})} \begin{cases} \{ \underline{\text{put}} \ e \ \underline{\text{on}} \ c \ll y \gg \} & \text{falls } y \neq \ast, \text{ wobei } y = V(e/\text{ENV}) \\ \{ \underline{\text{abort}} \} & \text{sonst} \end{cases}$$

Im übrigen sei A' definiert wie A . Für Kommunikationsfunktionen verwenden wir die UNFOLD-Regel aus dem folgenden Abschnitt.

Sei im weiteren

$$I_n =_{\text{def}} \{1, \dots, n\},$$

$$F_n =_{\text{def}} \{f: I_n \rightarrow I_n, \text{ wobei } f \text{ bijektiv ist}\}.$$

Für jeden Pseudoablauf b , der die Kommunikationstabelle c enthält, können wir die put-Aktionen durch Zahlen aus I_n eindeutig indizieren, wenn b genau n put-Aktionen bzgl. c enthält. Sei $\{ap_i\}_{i \in I_n}$ die Folge der so indizierten put-Aktionen in b bzgl. c und sei $\{vp_i\}_{i \in I_n}$ die Folge der eingetragenen Werte, d.h. mit $ap_i = \text{act}(\underline{\text{put}} \ e \ \underline{\text{on}} \ c \ll y \gg)$ sei $vp_i =_{\text{def}} y$.

Der Ablauf b heißt dann wertkonsistent bzgl. der Kommunikationstabelle c und $f \in F_n$, wenn $\exists \text{act}(\text{comtable } m \ c \ll e \gg)$
 $\in \text{actions}(b): \text{length}(e) = n \wedge \forall i \in I_n: \text{get}(e,i) = \nu p_{f(i)}$

Der Ablauf b heißt zeitkonsistent bzgl. der Kommunikationstabelle b , $f \in F_n$ und einer Ablaufordnung, wenn

$$\forall i, j \in I_n: f(i) \leq f(j) \Leftrightarrow ap_i \stackrel{E}{z} ap_j \quad \wedge$$

$$\forall e \in \text{actions}(b) \text{ der Form } \text{get}(c,i) \ll j \gg \text{ gilt:}$$

$$ap_{f(j)} \stackrel{E}{z} e$$

Damit lassen sich nun die Mengen von Abläufen für Programm-
 ausdrücke, in denen Kommunikationstabellen auftreten, definieren.

$$A(P, ENV) =_{\text{def}} \{ b: \mid b' \in A'(P, ENV):$$

es existiert eine totale Ablaufordnung,
 bzgl. der b' für jede Kommunikationstabelle in b' wert- und zeitkonsistent
 ist; weiter gilt $b = \text{normalize}(b')$ } $\}$

Dabei entferne $\text{normalize}(b')$ alle Ausdrücke der Form " $\ll e \gg$ ".

Daß Kommunikationstabellen nichtdeterministisch aufgebaut
 werden können, zeigt das folgende einfache Beispiel:

```
comtable nat c, // put 1 on c // put 2 on c // ; get(c,1)
```

Dieses Programm ist äquivalent zu dem Ausdruck

$$1 \parallel 2.$$

Dies zeigt, daß bei Kommunikationstabellen im Gegensatz zu
 Resultatbezeichnungen die Belegung mit Werten von der nicht
 determinierten zeitlichen Verzahnung parallel ablaufender
 Aktionen abhängen kann.

3.2.3 Programmentwicklung mit Kommunikationstabellen

Kommunikationstabellen können in der Programmentwicklung auf zwei unterschiedliche Weisen verwendet werden. Einmal können Algorithmen, die auf Sequenzen oder Feldern arbeiten, oder z.B. auf Rechenstrukturen wie in /Broy 78/ oder /Partsch, Broy 78/, in stärker parallel ablaufende Programme umgeformt werden. Für die Klassen solcher Programme sind Transformationsregeln zu entwickeln, die sich insbesondere aus einem "Wechsel der Rechenstruktur", also einem "Theorie-morphismus", ergeben.

Andererseits können nichtdeterministische Algorithmen, die z.B. mit Mengen als Objekten arbeiten (vgl. /Broy, Wirsing 79a/), einer parallelen Verarbeitung zugänglich gemacht werden.

Beispiel: Erzeugen und Verarbeiten einer Menge

```

funct gen = ( m x ) set r : if B(x) then t(x)
                                     else gen(h1(x)) ∪ gen(h2(x)) fi,
funct con = ( set r s ) n : if s = ∅ then C
                                     else r x = some r y : y ∈ s ;
                                     g(w(x), con(s \ {x})) fi,
con(gen(E))

```

Ein Übergang zu Kommunikationstabellen liefert: ¹⁾

```

comtable ( r | empty ) s, result n r,
comfunct cogen = ( m x, comtable ( r | empty ) s ) :
    if B(x) then put t(x) on s
    else // cogen(h1(x), s) // cogen(h2(x), s) // fi,
comfunct cocon = ( comtable ( r | empty ) s, nat i, n y, result n r ) :
    if empty :: get(s, i) then r is y
    else cocon(s, i+1, g(w(get(s, i)), y), r) fi,
// cogen(E, s) ; put empty on s // cocon(s, 1, C, r) //,
r

```

¹⁾ Dieser Übergang kann durch die Regel in Abschnitt 3.2.4 formal vollzogen werden.

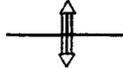
Die Äquivalenz der beiden obigen Programme kann durch Induktion über $|\text{gen}(E)|$ bewiesen werden.

Als wichtigste Regel für Kommunikationsfunktionen, die mit Kommunikationstabellen arbeiten, führen wir wieder die UNFOLD-Regel an. Sei

$$\text{comfunct } f = (\underline{m}_1 x_1, \dots, \underline{m}_n x_n, \\ \underline{\text{result}} p_1 y_1, \dots, \underline{\text{result}} p_i y_i, \\ \underline{\text{comtable}} q_1 z_1, \dots, \underline{\text{comtable}} q_j z_j): A$$

Dann gelte die Regel:

$$f(E_1, \dots, E_n, r_1, \dots, r_i, c_1, \dots, c_j)$$



$$\underline{m}_1 x_1 = E_1, \dots, \underline{m}_n x_n = E_n, A[r_1/y_1, \dots, r_i/y_i, c_1/z_1, \dots, c_j/z_j]$$

Die Verallgemeinerung dieser Regel auf beliebig permutierte Parameterlisten ist trivial. Durch die Regel werden die Abläufe von Programmen, die Kommunikationsfunktionen mit Kommunikationstabellen als Argumente enthalten, definiert.

Das oben betrachtete Beispiel beschreibt Programme, deren Kommunikationsbeziehung gerichtet ist. Eine allgemeinere Klasse von Programmen erhält man bei der Betrachtung von interaktiven Systemen. Solche Systeme bestehen aus einer Familie parallel ablaufender Prozesse, die untereinander wechselseitig kommunizieren. Beispiele für solche Systeme finden sich in Abschnitt 3.4 und - was die prozedurale Ebene betrifft - in den Abschnitten 5.1 und 6.2.

Natürlich sind Kommunikationstabellen nur ein Beispiel für aus Rechenstrukturen abgeleitete Kommunikationsstrukturen. Andere Rechenstrukturen ergeben andere (evtl. auch "applikative") Kommunikationsstrukturen. Durch die Wahl der Struktur wird eine Darstellung der Programme in ganz spezifischen Sprachelementen angestrebt. Wir nähern uns damit in gewisser Weise den Aufgabenstellungen der Systemprogrammierung.

3.2.4 Eine Regel für die gerichtete Kommunikation

Programme, die über Kommunikationstabellen kommunizieren, lassen sich aus Programmen entwickeln, die mit Objekten der Art table arbeiten. Hier wollen wir eine Regel für einen solchen Übergang angeben. Wir betrachten die Funktionen

```

funct fill = (table t, n x)table:
    if B(x) then fill(put(t,g(x)),h(x)) else t fi,
funct read = (table t, g y)r: E.

```

Falls in E nur lesend auf t zugegriffen wird, d.h. falls in E nur get-Operationen auf t angewendet werden, gilt die Transformationsregel:

```

read(fill(vac, E1), E2)

```



```

comtable m c, comfill(c, E1), read(c, E2)

```

wobei

```

comfunct comfill = (comtable m c, n x):
    if B(x) then put g(x) on c; comfill(c, h(x)) fi.

```

Die Korrektheit dieser Regel läßt sich durch Induktion über length(fill(vac, E₁)) beweisen.

Durch die Regel wird von einem rein sequentiellen Erzeugen einer Tabelle, die erst nach ihrer vollständigen Fertigstellung in read gelesen wird, zu einer sequentiellen Erstellung der Tabelle übergegangen, wobei nun jede Komponente unmittelbar nach ihrem Eintrag in die Tabelle gelesen werden kann, d.h. wobei das Erstellen und das Lesen der Tabelle parallel erfolgen.

Die beiden parallel ablaufenden Prozesse des Lesens und Erstellens der Tabelle kommunizieren über die Kommunikationstabelle.

3.3 Nichtstrikte Auswahl und disjunktives Warten

Die Definition der Semantik eines (teilweise) nichtstrikten Auswahloperators erfordert die (quasi-)parallele Auswertung beider Operanden. Die grundlegende Bedeutung eines solchen Sprachelements für die parallele Programmierung läßt sich auch aus der Tatsache ersehen, daß mit seiner Hilfe Sprachelemente für disjunktives Warten eingeführt werden können. Darüberhinaus ergeben sich Zusammenhänge mit nichtstrikten logischen Ausdrücken und der parallelen Auswertung bewachter Ausdrücke.

3.3.1 Der nichtstrikte Auswahloperator

Das einzige nichtstrikte Sprachelement der Breitbandsprache sind bewachte Ausdrücke (bzw. bewachte Anweisungen). Für alle übrigen Sprachelemente impliziert die Nichtdefiniiertheit von Teilausdrücken die Nichtdefiniiertheit des gesamten Ausdrucks. So gilt beispielsweise für die Auswahl:

$$d(E_1 \sqcup E_2) = d(E_1) \wedge d(E_2).$$

In /McCarthy 63/ wird unter der Bezeichnung "amb" ("Ambiguity Operator") ein nichtdeterministischer Auswahloperator eingeführt und folgendermaßen beschrieben:

"We define a basic ambiguity operator $\text{amb}(x,y)$, whose possible values are x or y when both are defined, otherwise, whichever is defined."

Definiert man dementsprechend:

$$B(\text{amb}(E_1, E_2)) = B(E_1) \cup B(E_2), \quad d(\text{amb}(E_1, E_2)) = d(E_1) \vee d(E_2),$$

so erweist sich der Auswahloperator amb unglücklicherweise als nicht monoton in der Egli-Milner Ordnung, wie das folgende Beispiel zeigt:

Es gilt $\text{amb}(1, \omega) \not\sqsubseteq \text{amb}(1, 2)$, obwohl $1 \sqsubseteq 1$ und $\omega \sqsubseteq 2$ gelten.

Allerdings ist die mit dem nichtstrikten Auswahloperator verwandte, nichtstrikte logische Disjunktion " \heartsuit " monoton, wobei die Bedeutung von \heartsuit definiert ist, wie folgt:

$$\begin{aligned}
 B(E_1 \heartsuit E_2) &= \text{def } (B(E_1) \wedge B(E_2) \wedge \{\underline{\text{false}}\}) \cup ((B(E_1) \cup B(E_2)) \wedge \{\underline{\text{true}}\}), \\
 d(E_1 \heartsuit E_2) &= \text{def } (d(E_1) \wedge B(E_1) = \{\underline{\text{true}}\}) \vee (d(E_2) \wedge B(E_2) = \{\underline{\text{true}}\}) \\
 &\quad \vee (d(E_1) \wedge d(E_2)).
 \end{aligned}$$

Mit dieser Definition erfordert der Operator \heartsuit die parallele Auswertung beider Operanden mit Abbruch der einen Auswertung, falls die andere Auswertung mit dem Resultat true beendet wird. Mit Hilfe dieses Operators lassen sich natürlich sofort weitere nichtstrikte logische Operatoren definieren, wie zum Beispiel die nichtstrikte logische Konjunktion \spadesuit :

$$E_1 \spadesuit E_2 = \text{def } \neg(\neg E_1 \heartsuit \neg E_2).$$

Wenden wir uns nun wieder dem Auswahloperator zu. Anstelle des nichtstrikten Auswahloperators nach McCarthy führen wir einen "etwas strikteren" Auswahloperator ein, der semantisch zwischen dem strikten Auswahloperator \square und dem nichtstrikten Operator \heartsuit liegt. Informell läßt sich die Semantik von \square in folgender Weise beschreiben:

Das Objekt x ist Element der Breite $B(E_1 \square E_2)$, wenn es Element der Breite $B(E_1)$ oder der Breite $B(E_2)$ ist. Die Definiiertheit $d(E_1 \square E_2)$ ist gegeben, wenn weder für E_1 , noch für E_2 ein unendlicher Ablauf existiert, und $d(E_1)$ oder $d(E_2)$ gilt.

Zur formalen Definition der mathematischen Semantik des Auswahloperators \square ist damit eine sorgfältige Unterscheidung zwischen nichtterminierenden Berechnungen und in endlicher Zeit auf einen Fehler ("Exception") führenden Berechnungen nötig. Dazu zerlegt man das Definierheitsprädikat in:

$$d(E) = \text{terminates}(E) \wedge \text{exceptionfree}(E).$$

Das Prädikat "terminates" übernimmt dann die Rolle von "d" bzgl. der Definition der Egli-Milner Ordnung, wohingegen "endliche Fehler" wie definierte Objekte behandelt werden. Wir verzichten hier auf die explizite Durchführung dieser Definition einer mathematischen Semantik und beschreiben die Semantik des Auswahloperators "operativ" über Abläufe.

3.3.2 Abläufe für den Auswahloperator

Analog zur informellen Beschreibung definieren wir:

$$A(E_1 \square E_2, ENV) = \text{def } \bigcup_{e_1 \in A(E_1, ENV)} \bigcup_{e_2 \in A(E_2, ENV)} \text{choice}(e_1, e_2, ENV)$$

$$\text{choice}(e_1, e_2, ENV) = \begin{cases} \{e_1\} & \text{falls } d(e_1') \wedge \neg d(e_2') \wedge \neg \text{infinite}(e_2) \\ \{e_2\} & \text{falls } d(e_2') \wedge \neg d(e_1') \wedge \neg \text{infinite}(e_1) \\ \{e_1, e_2\} & \text{sonst} \end{cases}$$

wobei $e_1' = e_1 // ENV$ und $e_2' = e_2 // ENV$.

Damit ist natürlich auch die mathematische Semantik für den Auswahloperator \square festgelegt. Durch die obige Definition wird erstmals das Prinzip der eins-zu-eins Korrespondenz zwischen den Teilausdrücken eines Ablaufs und den bei Auswertung eines Ausdrucks anfallenden Aktionen durchbrochen, da gewisse, auf undefinierte Situationen führende Abläufe nicht repräsentiert werden.

3.3.3 Disjunktives Warten

Ein erstes Beispiel für disjunktives Warten stellt der Ausdruck

$$\text{get}(c_1, i) \square \text{get}(c_2, j)$$

dar, wobei c_1 und c_2 Bezeichnungen für Kommunikationstabellen seien und $i, j \in \mathbf{N}$. Wird in nur einer der Tabellen der entsprechende Wert belegt, so liefert der Ausdruck diesen Wert, auch wenn in der anderen Tabelle der entsprechende Wert nie belegt wird.

Ein Sprachelement für bedingtes, disjunktives Warten kann durch eine definierende Transformationsregel auf den \square -Operator zurückgeführt werden:

$$\begin{array}{c} \underline{\text{if } P_1 \text{ then } E_1 \square \dots \square P_n \text{ then } E_n \text{ fi}} \\ \Downarrow \\ \underline{\text{nat } i = \text{if } P_1 \text{ then } 1 \text{ fi} \square \dots \square \text{if } P_n \text{ then } n \text{ fi},} \\ \underline{\text{if } i = 1 \text{ then } E_1 \square \dots \square i = n \text{ then } E_n \text{ fi}} \end{array}$$

Der Zusammenhang zwischen disjunktivem Warten und bewachten Ausdrücken wird durch die folgende Transformationsregel deutlich:

$$\begin{array}{c} \underline{\text{if}} \ B_1 \ \underline{\text{then}} \ E_1 \ \underline{\text{fi}} \ \parallel \ \dots \ \parallel \ B_n \ \underline{\text{then}} \ E_n \ \underline{\text{fi}} \\ \downarrow \\ \underline{\text{if}} \ B_1 \ \underline{\text{then}} \ E_1 \ \underline{\text{fi}} \ \parallel \ \dots \ \parallel \ B_n \ \underline{\text{then}} \ E_n \ \underline{\text{fi}} \end{array}$$

Im Gegensatz zu bewachten Ausdrücken werden in Ausdrücken für disjunktives Warten die Wächter parallel und nicht sequentiell ausgewertet. Liefert mindestens ein Wächter nach endlicher Zeit ein definiertes Resultat mit Wert true, so beeinflussen andere, nichtdefinierte Wächter das Resultat nicht. Deshalb kann disjunktives Warten als u.U. stärker definierte Implementierung für bewachte Ausdrücke verwendet werden.

Treten in den Wächtern B_i beim disjunktiven Warten Resultatbezeichnungen oder Aufrufe von "get" bzgl. global vereinbarter Kommunikationstabellen auf, so kann stets derjenige Zweig ausgewählt werden, für den die Auswertung von B_i zuerst oder überhaupt mit true endet. Eine Anwendung zeigt das folgende Beispiel.

Beispiel: Das Erzeuger/Verbraucher-Problem mit zwei Verbrauchern

```

[comtable m buffer1, comtable m buffer2,
 comtable bool request1, comtable bool request2;
 comfunct consumer = (comtable m b, comtable bool r, nat n):
   [put true on r; consume(get(b,n)); consumer(b,r,n+1)],

comfunct pro = (nat n1, nat n2):
  if get(request1,n1) then put product on buffer1; pro(n1+1,n2)
  fi get(request2,n2) then put product on buffer2; pro(n1,n2+1)
  fi

// pro(1,1) // consumer(buffer1,request1,1) //
                consumer(buffer2,request2,1)//
    ]
    
```


3.4 Programmbeispiele

Wie bereits angekündigt, sollen nun einige einfache Beispiele für die Verwendung von Kommunikationstabellen gegeben werden, die demonstrieren, wie Koordinationsprobleme mit Kommunikationstabellen gelöst werden können. Eine etwas ausführlichere Programmentwicklung enthält das Kapitel 6.

3.4.1 Das Erzeuger/Verbraucher Problem

In Abschnitt 3.2.1 wird das Erzeuger/Verbraucher Problem mit unbeschränktem Puffer durch Kommunikationstabellen beschrieben. Auch die etwas komplexere Aufgabe, bei der der Erzeuger maximal n Einheiten mehr erzeugen darf, als der Verbraucher verbraucht hat, läßt sich mit Kommunikationstabellen formulieren.

```
comtable bool in, comtable bool out,  
comfunct erzeuge = ( nat i ):  
    if get(out,i) then erzeuge; put true on in; erzeuge(i+1) fi,  
comfunct verbrauche = ( nat j ):  
    if get(in,j) then verbrauche; put true on out;  
                                     verbraucher(j+1) fi,  
for k to n do put true on out od;  
// erzeuge( 1 ) // verbraucher( 1 ) // └
```

Dieses Programm beschreibt ein nichtterminierendes System von Kommunikationsfunktionen. Eine formale Behandlung solcher Systeme findet sich in Abschnitt 5.4.

Im Unterschied zu Formulierungen durch Semaphore und P-/V-Operationen ist die der P-Operation entsprechende get-Operation eine reine Leseaktion.

3.4.2 Die 5 dinierenden Philosophen

Ein mittlerweile schon klassisches Problem der parallelen Programmierung verkörpert das Beispiel der 5 dinierenden Philosophen. Die Aufgabe läßt sich knapp in folgender Weise beschreiben. Die 5 Philosophen sitzen um einen runden Tisch und denken und essen abwechselnd. Es muß jedoch dafür Sorge getragen werden, daß benachbarte Philosophen nie gleichzeitig essen.

Eine zu restriktive Lösung läßt immer nur genau einen Philosophen essen.

```

mode philo = nat [1..5]; comtable philo in, comtable philo out,
    // put 1 on in; live(1,1,r1) //
        :
    put 5 on in; live(5,1,r5) //

```

wobei

comfunct live = (philo p, nat i, nat r):

philo x = get(in,i),

```

if p = x then eating(p); put p on out; thinking(p);
    if r > 0 then put p on in; live(p,i+1,r-1) fi
[] p ≠ x then if get(out,i) = x then live(p,i+1,r) fi fi

```

Dabei beschreibt r_p die Anzahl der EB/Denk-Phasen des Philosophen p. Man kann die Wirkungsweise des obigen Programms folgendermaßen beschreiben:

Der Philosoph p hat die ersten i-1 Einträge in der Tabelle in bereits gelesen. Verspürt er Hunger, so trägt er sich in die Tabelle in ein. Anschließend liest er die Einträge in der Tabelle in. Ist der von ihm gerade gelesene Eintrag p, so ist er und trägt sich anschließend in der Tabelle out wieder aus, um anderen Philosophen das Ende seiner EBphase mitzuteilen. Ist der gelesene Eintrag verschieden von p, so überprüft er ob der eingetragene Philosoph sein Essen beendet hat und liest danach den nächsten Eintrag in in.

Zwar ist bei dieser Vorgehensweise sichergestellt, daß benachbarte Philosophen niemals gleichzeitig essen, aber auch nicht benachbarte Philosophen hindern sich nun am Essen, obwohl dies garnicht notwendig ist. Deshalb ändern wir nun das Programm eines Philosophen etwas. Nach dem Eintrag in die Tabelle in, um seine Essensbereitschaft anzukündigen, liest der Philosoph p alle Einträge in der Tabelle in bis zu seinem eigenen Eintrag. Nur Einträge seiner Nachbarn merkt er sich. Nach Erreichen seines Eintrags überprüft er, ob alle von ihm gefundenen Einträge von Nachbarn auch durch Einträge in der Tabelle out schon abgeschlossene Essensphasen bezeichnen.

Zum Speichern der Einträge verwenden wir "Multisets" ("Bags"), die durch den folgenden abstrakten Datentyp beschrieben werden.

type MULTISET = (mode m) mset, empty, iselem, isempty, join, delete:

mode mset,
funct mset empty,
funct (mset, m)mset join,
funct (mset, m)mset delete,
funct (mset)bool isempty,
funct (mset, m)bool iselem,

law: iselem(empty,x) = false,

law: iselem(join(s,y),x) = if x = y then true
else iselem(s,x) fi,

law: delete(empty,x) = empty,

law: delete(join(s,y),x) = if x = y then s
else join(delete(s,x),y) fi,

law: isempty(empty) = true,

law: isempty(join(s,x)) = false

end of type

Damit lassen sich die essenden Philosophen folgendermaßen beschreiben:

```
comfunct live = (philo p, nat i, nat j, mset philo s, nat r):  
  [philo x = get(in,i),  
  if p = x ^ isempty(s) then eating(p); put p on out; thinking(p);  
    if r = 0 then put p on in;  
      live(p,i+1,j,empty,r-1) fi  
  [p = x ^ ¬isempty(s) then live(p,i,j+1,delete(s,get(out,j)),r)  
  [p ≠ x then if isneighbour(p,x) then  
    live(p,i+1,j,join(s,x),r)  
    else  
      live(p,i+1,j,s,r)  
    fi  
  fi
```

Der initiiierende Aufruf lautet nun

```
live(p,1,1,empty,rp)
```

Das Prädikat isneighbour sei die symmetrische, irreflexive Nachbarschaftsrelation.

Um die Korrektheit des obigen Programms zu beweisen verwenden wir die Funktion number auf Multisets:

```
funct number = (mset m s, m y)nat:  
  if ¬iselem(s,y) then 0  
  else number(delete(s,y),y)+1 fi
```

in dem Prädikat:

$$\begin{aligned} \forall \text{ philo } x, \text{ philo } y: \text{ isneighbour}(x,y) = & \\ & |\{\underline{\text{nat}}[1..i-1] k: \text{get}(\text{in},k) = y\}| = \\ & |\{\underline{\text{nat}}[1..j-1] k: \text{get}(\text{out},k) = y\}| + \\ & \text{number}(s,y) \end{aligned}$$

Dabei gibt $\text{number}(s,y)$ die Anzahl der y in s wieder. Dieses Prädikat ist für jeden Prozeß invariant. Da zusätzlich bei jedem Aufruf von $\text{live}(x,\dots)$ gilt:

"Anzahl der Einträge von x in 'in' " =

"Anzahl der Einträge von x in 'out' " + 1

wird sowohl eine Kollision, d.h. gleichzeitiges Essen zweier Nachbarn, als auch eine Verklemmung ausgeschlossen, wie der folgende Widerspruchsbeweis zeigt:

Annahme: Philosoph p und q essen gleichzeitig, obwohl sie benachbart sind. Dann gilt $s_p = s_q = \emptyset$. Sei ohne Beschränkung der Allgemeinheit $i_p < i_q$, dann gilt

$$|\{\underline{\text{nat}}[1..i_q-1] k: \text{get}(\text{in},k) = p\}| >$$

$$|\{\underline{\text{nat}}[1..j_q-1] k: \text{get}(\text{out},k) = p\}|$$

das steht jedoch im Widerspruch zur obigen Invarianten.

Das obige Programm kann auch auf beliebige Nachbarschaftsbeziehungen übertragen werden.

Teil II

Paralleler Ablauf prozeduraler Programme

Die für die applikative Teilsprache entwickelten Begriffe der parallelen Programmierung lassen sich auf die prozedurale Ebene übertragen. So läßt sich die operative Semantik von prozeduralen Programmen ebenfalls algebraisch durch Abläufe definieren. Abläufe von Anweisungen sind dabei selbst wieder Programme, die nur noch Deklarationen, Zuweisungen, Strichpunkte und einfach bewachte Anweisungen mit genau einer Alternative enthalten. Alle in solchen Abläufen auftretenden Ausdrücke sind ihrerseits wieder Abläufe im Sinne von Abschnitt 1.1. Im Gegensatz zur applikativen Ebene benötigt man für die Definitionen der Abläufe von Anweisungen eine etwas kompliziertere Behandlung der Umgebungen, da mit ihrer Hilfe die Wirkung von Zuweisungen ausgedrückt werden muß.

Die Einführung von Begriffen wie Aktion, Prozeß, Ablaufordnung und Effizienzmaß kann dann jedoch völlig analog zur applikativen Ebene vorgenommen werden. Die Erweiterung der Sprache um Anweisungen für die parallele Ausführung und die Kommunikation könnte mit den gleichen Methoden wie im Teil I erfolgen. Wir verzichten hier jedoch vorerst auf die Weiterverfolgung des Ansatzes der Abläufe und wenden uns stärker den Möglichkeiten der Transformationssemantik zu, da neben der formalen Definition der Semantik hier gleichzeitig auch eine formale Grundlage für die Entwicklung paralleler Programme durch Transformationen geschaffen wird.

War für applikative Programme die Funktionsapplikation der Ausgangspunkt für unsere Überlegungen zur parallelen Verarbeitung, so findet sich in der im Sinn der Transformationssemantik daraus entstehenden Kollektivzuweisung ein erster Einstiegspunkt für die parallele Verarbeitung von Anweisungen. Ist nämlich für eine Kollektivzuweisung die "Bernsteinbedingung" erfüllt, so kann sie in einen Satz parallel ausführbarer Zuweisungen umgewandelt werden (vgl. die Paralleldeklaration in Teil I).

Die Einbeziehung nicht konfliktfreier Zuweisungen kann mit Hilfe bewachter kritischer Bereiche erfolgen. Über axiomatische Transformationsregeln wird ihre Bedeutung formal beschrieben. Mit Hilfe dieser Regeln lassen sich sowohl Beweise über parallele Programme führen, als auch parallele Programme aus (nichtdeterministischen) sequentiellen Programmen entwickeln.

Parallele, prozedurale Programme können mit Hilfe der bewachten, kritischen Bereiche auf das Eintreten bestimmter Bedingungen warten und unter dem Schutz der Bereiche über gemeinsame Variable kommunizieren. Dieser Ansatz wird erweitert zu mehrfach bewachten Bereichen mit kritischer und unkritischer Phase. Damit umfaßt das entstehende Sprachelement sowohl disjunktives Warten, als auch die Schachtelung paralleler Anweisungen bzw. den "dynamischen" Aufbau von Systemen von parallelen Prozessen.

Ein direkt auf die Kommunikation paralleler Programme der prozeduralen Ebene ausgerichtetes Sprachelement stellen (Kommunikations-)Ströme dar. Sie können formal mit Hilfe algebraischer Datentypen spezifiziert und über Programmvariable und kritische Bereiche auf der Ebene der prozeduralen Programmierung eingeführt werden. Ströme und verwandte Konzepte erscheinen als angemessene Sprachelemente z.B. für die Beschreibung und Programmierung des Kommunikationsflusses in Rechnernetzen. Somit können sie als klassisches Beispiel für Sprachelemente dienen, wie wir sie auch in der Systemprogrammierung finden.

Ein Strom besteht dabei aus einer (evtl. leeren) Schlange von Objekten. Das vorderste Element des Stroms kann gelesen werden und wird dabei gleichzeitig entfernt, am Ende des Stroms können neue Elemente angefügt werden. Ist ein Strom leer, dann führt eine Leseversuch zu einem Wartezustand. Ströme bezeichnen wir im Gegensatz zu Kommunikationstabellen nicht als applikativ, da Information beim Lesen zerstört wird.

Eine einfache Erweiterung des algebraischen Datentyps der Ströme führt zu verschlüsselten Strömen. In verschlüsselten Strömen werden die Objekte unter Schlüsseln abgelegt und unter den gleichen Schlüsseln wieder gelesen. Solche Ströme gleichen "Feldern von einfachen Strömen", wobei die Schlüssel die Rolle der Selektoren spielen. Verschlüsselte Ströme können sowohl den Mißbrauch von Informationen durch unbefugte Benutzer verhindern helfen, als auch die unabhängige Verwendung eines Stroms durch mehrere Benutzer erlauben.

Man kann Ströme und die dazugehörigen Operationen natürlich auch als primitive Sprachelemente einführen und ihnen direkt eine Semantik zuordnen. Durch Ströme lassen sich simultan benutzbare Programmvariable simulieren und umgekehrt. Die Ausnutzung dieser Dualität in der formalen Definition der Ströme durch Transformationsregeln gestattet den formalen Übergang von Programmen mit Strömen zu Programmen mit gemeinsamen Variablen und umgekehrt.

Durch die in der Semantikbeschreibung nicht festgelegte Sequentialisierung konkurrierender Kommunikationsaktionen entsteht Nichtdeterminismus in der parallelen Programmierung. Durch die definierenden Transformationsregeln wird dieser globale Nichtdeterminismus auf den lokalen Nichtdeterminismus bewachter Anweisungen zurückgeführt. Damit wird beispielsweise Dijkstra's Kalkül der schwächsten Vorbedingungen auf parallele Programme anwendbar. Wenn die Terminierung eines Programms von der nichtdeterminierten Auswahl abhängt, sprechen wir von unbeschränktem Nichtdeterminismus. Für solche Programme gestatten die schwächsten Vorbedingungen keinerlei Aussagen mehr über das Programm - außer daß es eben möglicherweise nicht terminiert. Dies entspricht der Auffassung, daß nur Aussagen über garantiert terminierende Programme sinnvoll sind.

Gerade bei gewissen parallelen Programmen tritt jedoch das Problem des unbeschränkten Nichtdeterminismus auf, wenn beispielsweise die Terminierung eines Systems paralleler Programme von der nichtdeterminierten Verzahnung der Kommunikationsaktionen abhängt.

So können Systeme paralleler Programme, die abwechselnd kritische und unkritische Phasen durchlaufen und genau dann terminieren, wenn ein bestimmtes Programm P einmal seine kritische Phase durchlaufen hat, möglicherweise nicht terminieren, wenn P durch die anderen abwechselnd in ihre kritischen Phasen eintretenden Prozesse an der Ausführung seiner kritischen Phase gehindert wird ("Dynamic Blocking", "Starvation"). Solche Systeme sind damit unbeschränkt nichtdeterminiert.

Um Abhilfe zu schaffen, werden häufig allgemeine Fairnessforderungen erhoben, wie "ein wartender Prozeß darf von einem anderen Prozeß nur endlich oft überholt werden". Diese Beschränkung der Auswahlmöglichkeit durch die Forderung nach allgemeiner Fairness kollidiert jedoch mit den Stetigkeitsbedingungen der Sprache bzgl. der für die Fixpunktsemantik grundlegenden Egli-Milner Ordnung. Genauer gesagt, würden sich unter der Annahme der Gültigkeit allgemeiner Fairnessbedingungen Programme formulieren lassen, die in der Egli-Milner Ordnung nicht stetig sind, d.h. für die der schwächste Fixpunkt nicht mit dem Grenzwert der Funktionaliteration übereinstimmt. Diese Diskrepanz kann auch nicht - wie an einem einfachen Beispiel demonstriert werden kann - durch die Ersetzung der Egli-Milner Ordnung durch eine andere partielle Ordnung beseitigt werden. Vielmehr scheint die Forderung nach Fairness nicht mit dem zugrunde gelegten

Berechenbarkeitsbegriff vereinbar. Darüberhinaus ist eine Fairnessbedingung, die einem Benutzer keine oberen Schranken für seine Wartezeit garantiert, für praktische Zwecke kaum zu gebrauchen. Deshalb verzichten wir auf jegliche Fairnessforderungen bzgl. der verwendeten Sprachelemente.

In der parallelen Programmierung werden häufig nichtterminierende Systeme paralleler Programme betrachtet, um in der Praxis unbeschränkt lange laufende Prozesse wie z.B. in Betriebssystemen zu beschreiben. Die übliche Fixpunktsemantik versagt bei der formalen Beschreibung dieser Systeme, da sie solche Programme stets mit undefiniert oder abort gleichsetzt. Über (unendliche) Abläufe kann solchen Systemen jedoch formal eine Bedeutung zugeordnet werden. Die auf dieser formalen Grundlage verfügbaren Induktionsmethoden ergeben zusammen mit den definierenden Transformationsregeln einen Kalkül, der die formale Entwicklung unendlich lange laufender Programmsysteme erlaubt. So können wohlbekannte Versionen paralleler Programme aus nichtdeterministischen, sequentiellen, nichtterminierenden Programmen hergeleitet werden.

Als klassisches Beispiel für solche nichtterminierenden Systeme können Petrinetze betrachtet werden. Sie können trivial in nichtterminierende Systeme paralleler Programme umgesetzt werden. Da die so entstehenden Programme natürlich durch Transformationen weiterentwickelt werden können, ist ein Vergleich der unterschiedlichen (Formulierungen von) Lösungen gewisser Aufgabenstellungen aus dem Bereich paralleler Programme durch Petrinetze und durch bewachte, kritische Bereiche möglich.

Der klassische Übergang von applikativen (repetitiver Rekursion) zu prozeduralen Programmen entspricht (im Sinne operativer Semantik) einer konsequenten Einschränkung auf "Innermost"-Berechnungsregeln. Eventuell auf der applikativen Ebene gegebene, weitergehende Möglichkeiten paralleler Auswertung werden zugunsten einer Speicheroptimierung eingeschränkt. Jedoch erlauben die bis hierhin entwickelten Transformationstechniken, die in der nichtdeterministischen Berechnungsregel enthaltenen Möglichkeiten paralleler Auswertung rekursiver Programmschemata auf die prozedurale Ebene zu übertragen, wodurch man Übergänge von applikativen zu prozeduralen, parallelen Programmen erhält.

4 Grundelemente prozeduraler, paralleler Programme

In applikativen Sprachelementen treten durch das Fehlen von Seiteneffekten keinerlei Konflikte bei der parallelen Abarbeitung von Ausdrücken auf. Selbst die Konflikte zwischen konkurrierenden Einträgen in Kommunikationstabellen lassen sich problemlos durch einen kontrollierten Nichtdeterminismus auflösen.

Prozedurale Sprachelemente, also Anweisungen, können, soweit sie konfliktfrei sind, ebenfalls ohne Probleme parallel ausgeführt werden. Treten jedoch Konflikte auf, so führt ein nichtdeterministisches Sequentialisieren im allgemeinen auf unkontrollierbare Effekte.

Abhilfe versprechen Sprachelemente wie bewachte kritische Bereiche, die konfliktträchtige Anweisungen besonders kennzeichnen und eine unkontrollierte Mischung verhindern. Die Bedeutung solcher Sprachelemente kann durch definierende Transformationen formal beschrieben werden.

Zunächst aber sollen die im Kapitel 1 eingeführten Begriffe auf die prozedurale Ebene übertragen werden.

4.1 Abläufe der prozeduralen Ebene

In Analogie zu den in Abschnitt 1.1 definierten Abläufen definieren wir nun Abläufe für Anweisungen. Wieder werden dabei ablaufbestimmende Elemente eliminiert, bis ein gegebenenfalls unendliches, deterministisches Programm vorliegt, das nur noch Deklarationen, Zuweisungen und bewachte Anweisungen mit genau einer Alternative enthält. Alle noch auftretenden Ausdrücke sind ihrerseits wieder Abläufe. In Abläufen werden also Prozeduraufrufe (nichtprimitiver Prozeduren), Wiederholungsanweisungen und bedingte Anweisungen mit mehreren Alternativen eliminiert. Auch Sprunganweisungen können durch ein solches Vorgehen eliminiert werden, so daß auch Abläufe von Programmen mit Sprunganweisungen definiert sind.

4.1.1 Abläufe für Anweisungen

Da Anweisungen die Umgebungen verändern können, definieren wir Abläufe für Anweisungen über die Funktion

$$AU: STA \times \mathcal{E} \longrightarrow \mathcal{P}(STA \times \mathcal{E})$$

wobei \mathcal{E} die Menge der Umgebungen bezeichnet. AU liefert also für jedes Paar von Anweisungen und Umgebungen eine Menge von Paaren aus Abläufen und Umgebungen. In Abläufen von Anweisungen treten nur noch Deklarationen, Zuweisungen Strichpunkte und bedingte Ausdrücke mit genau einer Alternative auf. Alle auftretenden Ausdrücke sind Abläufe von Ausdrücken.

(1) Leere und nichtterminierende Anweisung

$$AU(\underline{\text{nop}}, ENV) =_{\text{def}} \{(\underline{\text{nop}}, ENV)\},$$

$$AU(\underline{\text{abort}}, ENV) =_{\text{def}} \{(\underline{\text{abort}}, ENV)\}.$$

(2) Initialisierende Deklaration, Zuweisungen

$$AU(\underline{\text{var } m_1 v_1, \dots, \text{var } m_n v_n} := E, ENV) =_{\text{def}}$$

$$\{(\underline{\text{var } m_1 v_1, \dots, \text{var } m_n v_n} := e, ENV') :$$

$$\begin{aligned} & e \in A(E, ENV) \wedge ENV' = \text{upd}(ENV, \{v_1 = e_1, \dots, v_n = e_n\}) \\ & \wedge d(e) \Rightarrow (e_1, \dots, e_n) = V(e // ENV) \\ & \wedge \neg d(e) \Rightarrow (e_1, \dots, e_n) = (\underline{\text{error}}, \dots, \underline{\text{error}}) \end{aligned} \}$$

wobei $\text{upd}(ENV, S) =_{\text{def}}$

$$(ENV \setminus \{ "x = d" \in ENV : \exists d' : "x = d'" \in S \}) \cup S$$

Für Zuweisungen gilt exakt die gleiche Definition, nur daß die Modeangaben "var m_i" auf der linken und rechten Seite weggelassen werden. Man beachte, daß wieder davon ausgegangen wird, daß keine Überlagerung von Variablenbezeichnungen auftritt.

(3) Sequentialisierung

$$\begin{aligned}
 & \text{AU}(\overline{S_1}; \dots; \underline{S_n}, \text{ENV}) =_{\text{def}} \\
 & \{(\overline{s_1}; \underline{R}, \text{ENV}') : (s_1, \text{ENV}') \in \text{AU}(S_1, \text{ENV}) \wedge \\
 & (\underline{R}, \text{ENV}') \in \left\{ \begin{array}{ll} \text{AU}(\overline{S_2}; \dots; \underline{S_n}, \text{ENV}) & \text{falls } \neg \text{occurs}(\underline{\text{abort}}, s_1) \\ & \vee \exists "x = e" \in \text{ENV}' : d(e) \\ \{(\underline{\text{abort}}, \text{ENV}')\} & \text{sonst} \end{array} \right\}
 \end{aligned}$$

(4) Bedingte und bewachte Anweisungen

$$\text{AU}(\underline{\text{if}} B_1 \underline{\text{then}} S_1 \parallel \dots \parallel B_n \underline{\text{then}} S_n \underline{\text{fi}}, \text{ENV}) =_{\text{def}}$$

$$\text{AU}(\underline{\text{if}} B_1 \underline{\text{then}} S_1 \parallel \dots \parallel B_n \underline{\text{then}} S_n \underline{\text{else}} \underline{\text{nop}} \underline{\text{fi}}, \text{ENV});$$

Falls $n \geq 2$:

$$\text{AU}(\underline{\text{if}} B_1 \underline{\text{then}} S_1 \parallel \dots \parallel B_n \underline{\text{then}} S_n \underline{\text{else}} S_{n+1} \underline{\text{fi}}, \text{ENV}) =_{\text{def}}$$

$$\bigcup_{1 \leq i \leq n} \text{AU}(\underline{\text{if}} B_i \underline{\text{then}} S_i \underline{\text{else}} R_i \underline{\text{fi}}, \text{ENV}),$$

$$\begin{aligned}
 \text{wobei } R_i = & \underline{\text{if}} B_1 \underline{\text{then}} S_1 \\
 & \quad \vdots \\
 & \parallel B_{i-1} \underline{\text{then}} S_{i-1} \\
 & \parallel B_{i+1} \underline{\text{then}} S_{i+1} \\
 & \quad \vdots \\
 & \parallel B_n \underline{\text{then}} S_n \\
 & \quad \underline{\text{else}} S_{n+1} \underline{\text{fi}}
 \end{aligned}$$

Damit sind die Abläufe bewachter Anweisungen auf die Abläufe bedingter Anweisungen zurückgeführt.

$$\text{AU}(\underline{\text{if}} B \underline{\text{then}} S_1 \underline{\text{else}} S_2 \underline{\text{fi}}, \text{ENV}) =_{\text{def}}$$

$$\begin{aligned}
 & \{(\underline{\text{if}} b' \underline{\text{then}} s \underline{\text{fi}}, \text{ENV}') : b' \in \text{AU}(B, \text{ENV}) \wedge b \hat{=} b' // \text{ENV} \wedge \\
 & \quad d(b) \wedge \forall(b) \Rightarrow (s, \text{ENV}') \in \text{AU}(S_1, \text{ENV}) \wedge b' \hat{=} b \quad \wedge \\
 & \quad d(b) \wedge \neg \forall(b) \Rightarrow (s, \text{ENV}') \in \text{AU}(S_2, \text{ENV}) \wedge b' \hat{=} \neg b \quad \wedge \\
 & \quad \neg d(b) \Rightarrow s = \underline{\text{abort}} \wedge b' \hat{=} b' \wedge \text{ENV}' = \text{ENV} \quad \}
 \end{aligned}$$

(5) Wiederholungsanweisungen

AU(while B do S od , ENV) =_{def}

AU(if B then S; while B do S od fi , ENV).

(6) Prozeduraufrufe

Sei p eine (nichtprimitive) Prozedur, vereinbart durch

proc p = (var m₁ v₁ , ... , var m_i v_i , n₁ x₁ , ... , n_j x_j) : S

Dann gilt:

AU(p(y₁ , ... , y_i , E₁ , ... , E_j) , ENV) =_{def}

AU([n₁ x₁ , ... , n_j x_j] = (E₁ , ... , E_j); S [y₁/v₁ , ... , y_i/v_i] , ENV)

Die Verallgemeinerung auf beliebig gemischte Parameterlisten liegt auf der Hand.

Auch für Programme, die Marken und Sprunganweisungen enthalten, können Abläufe definiert werden. Dabei werden wie bei rekursiven Prozeduren Marken und Sprünge schrittweise eliminiert, indem man anstelle der Sprünge die entsprechenden Programmtexte ein-kopiert. Technisch kann man das entweder durch die Aufnahme von Programmtexten und Marken in die Umgebungen in Analogie zu den Continuations der denotationellen Semantik durchführen. Oder man benutzt explizit die Techniken der Textexpansion wie in /Broy, Wirsing 79b/ oder man führt Sprünge auf parameterlose, rekursive Prozeduren zurück. Aus Platzgründen wollen wir jedoch auf die explizite Durchführung der Definition von Abläufen für Sprunganweisungen verzichten.

4.1.2 Reduzierte Abläufe von Anweisungen

Abläufe enthalten als Sprachelemente nur noch

- Blöcke mit Deklarationen und durch Strichpunkte getrennte Anweisungen, bzw. Zuweisungen,
- bewachte Anweisungen mit genau einer Alternative.

Alle auftretenden Ausdrücke sind ihrerseits Abläufe.

Wie in Abschnitt 1.1.3 Abläufe von Ausdrücken lassen sich auch Abläufe von Anweisungen reduzieren. Durch entsprechende Transformationsregeln (vgl. /Pepper 79/) lassen sich bewachte Anweisungen, lokale Programmvariable und mehrfache Zuweisungen an globale Programmvariable in Abläufen eliminieren, bis nur noch Zuweisungen an globale Programmvariable zurück bleiben. Treten die Anweisungen im Kontext mit einem Ergebnisausdruck auf, so können alle Zuweisungen schließlich durch eine Substitution der rechten Seite der Zuweisung für die Programmvariablen im Ergebnisausdruck beseitigt werden und wir erhalten wie in 1.1.3 einen Term der Termalgebra der zugrunde liegenden abstrakten Datentypen.

Damit lassen sich applikative und prozedurale Programme bzgl. ihrer reduzierten Abläufe vergleichen. Der in 1.1.3 definierte Begriff algorithmischer Äquivalenz kann somit unabhängig vom Sprachstil auch zum Vergleich applikativer und prozeduraler Programme verwendet werden. Die Kongruenzrelation "algorithmisch äquivalent" liegt zwischen Ablaufäquivalenz (operativer Äquivalenz) und mathematischer Äquivalenz.

4.1.3 Aktionen, Prozesse, Ablaufordnungen prozeduraler Programme

Die in Kapitel 1 für applikative Programme durchgeführten Begriffsbildungen und Definitionen wie Aktion, Prozeß und Ablaufordnung lassen sich völlig analog für Anweisungen vornehmen. Wir wollen deshalb auf die explizite Durchführung verzichten. Da in prozeduralen Programmen Sequenzen von Anweisungen und Zuweisungen das beherrschende Sprachelement bilden, sind die Ablaufordnungen bedeutend dichter, d.h. viel mehr Aktionen sind sequentiell.

4.2 Parallele Abläufe der prozeduralen Ebene

Wie bei applikativen Abläufen treten in prozeduralen Programmen parallele Abläufe auf, wenn ein Tupel (z.B. auf der rechten Seite einer kollektiven Zuweisung) erarbeitet wird. Die Ausdrücke für die einzelnen Komponenten können parallel ohne Probleme berechnet werden, in ihnen auftretende prozedurale Sprachelemente, wie Zuweisungen, können nur lokale Variable verändern, da Effekte auf globale Variable ausgeschlossen sind.

Um auch parallele Ausführung von Zuweisungen an globale Variable formulieren zu können, wird nun aus der kollektiven Zuweisung ein Sprachelement für die parallele Ausführung von Anweisungen abgeleitet, ähnlich wie in Kapitel 2 für die kollektive Objektvereinbarung.

4.2.1 Parallele Ausführung von Zuweisungen

Betrachten wir die kollektive Zuweisung

$$(v_1, \dots, v_n) := (E_1, \dots, E_n)$$

so ergibt sich durch das Besetzungstabu die Forderung nach der Verschiedenheit der Variablen v_i (bzw. ihrer Bezeichnungen, wenn ein Aliasverbot die Verschiedenheit der Variablen bei unterschiedlichen Bezeichnungen garantiert). Wir definieren:

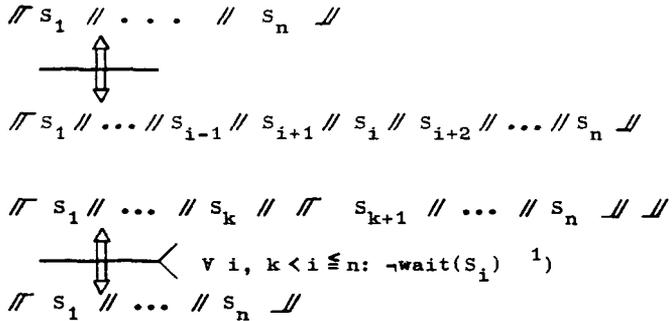
$$\mathcal{F}(v_1, \dots, v_k) := (E_1, \dots, E_k) \quad \mathcal{F}(v_{k+1}, \dots, v_n) := (E_{k+1}, \dots, E_n) \quad \mathcal{F}$$



$$v_i, j, 1 \leq i \leq k < j \leq n: \neg \text{occurs}(v_i \text{ in } E_j) \wedge \neg \text{occurs}(v_j \text{ in } E_i)$$

$$(v_1, \dots, v_n) := (E_1, \dots, E_n)$$

Zusammen mit dem Besetzungstabu ergibt die obige Anwendbarkeitsbedingung die sogenannte "Bernstein - Bedingung" (vgl. /Bernstein 66/). Da eine simultane Permutation der rechten und linken Seite einer kollektiven Zuweisung möglich ist, ergeben sich sofort die folgenden zwei Regeln



Die beiden obigen Transformationen zeigen die Assoziativität und Kommutativität der \parallel - und $\parallel\parallel$ -Klammern.

Als Abläufe definieren wir:

$$\text{AU}(\parallel S_1 \parallel \dots \parallel S_n \parallel, \text{ENV}) = \text{def}$$

$$\begin{aligned}
 & \{ (\parallel s_1 \parallel \dots \parallel s_n \parallel, \text{ENV}') : \exists \text{ENV}'_1, \dots, \text{ENV}'_n : \\
 & (s_1, \text{ENV}'_1) \in \text{AU}(S_1, \text{ENV}) \wedge \dots \wedge (s_n, \text{ENV}'_n) \in (S_n, \text{ENV}) \wedge \\
 & \text{ENV}' = \left(\bigcup_{1 \leq i \leq n} \text{ENV}'_i \right) \cup \left(\bigcup_{1 \leq i \leq n} (\text{ENV}'_i \setminus \text{ENV}) \right) \}
 \end{aligned}$$

Man beachte, daß die obige Ablaufdefinition natürlich nur für Anweisungen S_1, \dots, S_n gilt, die konfliktfrei sind, d.h. die die nachfolgende Bernsteinbedingung erfüllen.

¹⁾ vgl. Abschnitt 4.3

4.2.2 Die Bernsteinbedingung

Nach der Definition in 4.2.1 können wir nur Aussagen über die parallele Ausführung von Zuweisungen an globale Variable machen, die unmittelbar aus der kollektiven Zuweisung entstanden sind. Wendet man jedoch auf die "Komponenten" der Parallelanweisung weitere Transformationsregeln an, so können entsprechende, zu solchen Zuweisungen äquivalente Programme auftreten. Alle solchen parallel ausführbaren Programme, bzw. Anweisungen müssen dann zwangsläufig die Bernsteinbedingung erfüllen.

Definition: Die Parallelanweisung $\parallel S_1 \parallel \dots \parallel S_n \parallel$ erfüllt die Bernsteinbedingung, wenn gilt:

Es existiert keine Variable v , die in S_i eine Zuweisung erfährt und die in S_j auftritt ($i \neq j$); d.h. es gilt $\neg \text{occurs}(v \text{ in } S_j)$.

Solange eine Sprache das Aliasverbot garantiert, kann die Bernsteinbedingung statisch überprüft werden.

Lemma: Ein nach den Transformationen in Abschnitt 4.2.1 ableitbares Programm erfüllt die Bernsteinbedingung.

Beweis: Die Anwendbarkeitsbedingung der definierenden Transformationen garantieren die Bernsteinbedingung. Weitere lokale Transformationen können nur auf die vorhandenen globalen Variablen Bezug nehmen. □

4.2.3 Parallele Ausführung von nicht konfliktfreien Anweisungen

Die Beschränkung der parallelen Ausführung auf Anweisungen, die die Bernsteinbedingung erfüllen, ist sicherlich zu restriktiv. Eine völlige Aufhebung der Beschränkung führt jedoch auf eine Reihe semantischer Probleme.

Man kann z.B. wie in /Gries 77/, Seite 923, jede einzelne Anweisung als unteilbare Aktion definieren, und damit eine "quasi-parallele" Auswertung paralleler Anweisungen unterstellen. Eine solche Auffassung findet sich auch im ALGOL 68 Report,

wo die einzelnen Aktionen ("direct actions") einer kollateralen Aktion zeitlich verzahnt ausgeführt werden ("... are merged in time", vgl. /Wijngaarden et al. 75/, Seite 47).

Abgesehen davon, daß dies zu einer unbefriedigenden Begriffsbildung der parallelen Ausführung führt, wäre bei der Zulassung nicht konfliktfreier Anweisungen die Korrektheit lokaler Transformationen nicht mehr gesichert.

Betrachtet man die parallele Anweisung

$$\parallel x := x+2 \quad \parallel x := 2*x \quad \parallel$$

Dann liefert eine beliebige Sequentialisierung

$$x := x+2; x := 2*x$$

bzw.

$$x := 2*x; x := x+2$$

und die obige parallele Anweisung ist somit äquivalent zu

$$(x := 2*(x+2)) \parallel (x := 2*x+2).$$

Eine lokale Transformation der linken Anweisung liefert jedoch

$$\parallel x := x+1; x := x+1 \quad \parallel x := 2*x \quad \parallel$$

Diese parallele Anweisung gestattet jedoch die Sequentialisierung

$$x := x+1; x := 2*x; x := x+1$$

was äquivalent ist zu

$$x := 2*(x+1)+1$$

Die implizite Annahme, daß einzelne Anweisungen unteilbare Aktionen sind, kollidiert somit mit der Korrektheit lokaler Transformationen. Auch die maschinen- und übersetzerorientierte Vorstellung, daß jede Anweisung in eine Sequenz von elementaren Anweisungen aufgebrochen wird, steht im Gegensatz zu dieser Auffassung. Im folgenden Abschnitt wollen wir deshalb bewachte, kritische Bereiche durch definierende Transformationen einführen.

4.3 Transformationssemantik für bewachte, kritische Bereiche

Um in parallel ablaufenden Programmen die Benutzung gemeinsamer Programmvariablen ohne die eben beschriebenen Schwierigkeiten zu ermöglichen, wird in /Hoare 71/ vorgeschlagen, nicht konfliktfreie Anweisungen in "bewachte, kritische Bereiche" ("Conditional, Critical Regions") einzuschließen. In Erweiterung des Ansatzes von /Pepper 79/ kann die Bedeutung paralleler Programme mit bewachten, kritischen Bereichen durch definierende Transformationen erklärt werden. Sie erlauben parallele Programme auf äquivalente nicht-deterministische, sequentielle Programme zurückzuführen. Dadurch können sowohl Beweise über parallele Programme geführt werden, als auch parallele Programme aus sequentiellen entwickelt werden.

4.3.1 Bewachte, kritische Bereiche

Wir erweitern unsere Sprache um bewachte, kritische Bereiche der Form:

await B then S endwait

Dabei sei B ein boolescher Ausdruck und S eine Anweisung, in der keine bewachten, kritischen Bereiche auftreten. Angelehnt an die Bernsteinbedingung definieren wir:

Definition: Anweisungen oder Ausdrücke A_1 und A_2 heißen konfliktfrei (abgekürzt $\neg\text{conflict}(A_1, A_2)$), wenn in A_1 bzw. in A_2 keine Zuweisung an eine Programmvariable auftritt, die in A_2 bzw. in A_1 auftritt.

Eine Anweisung A heißt wartefrei (abgekürzt $\neg\text{wait}(A)$), wenn in A keine bewachten, kritischen Bereiche auftreten.

Beide Bedingungen sind statisch nachprüfbar. Wir fordern folgende Kontextbedingung für parallele Programme.

Konvention: Kontextbedingung

Für jede Parallelanweisung $\parallel S_1 \parallel \dots \parallel S_n \parallel$ gilt:

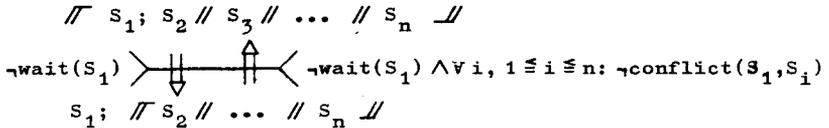
In S_i , $1 \leq i \leq n$, wird keine Zuweisung (auch nicht in einem kritischen Bereich) an eine Programmvariable vorgenommen, die in S_j , $1 \leq j \leq n$, $i \neq j$, außerhalb eines kritischen Bereichs auftritt.

4.3.2 Definierende Transformationsregeln für bewachte, kritische Bereiche

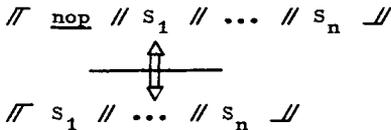
Für die Definition der Semantik bewachter, kritischer Bereiche geben wir folgende Transformationsregeln an:

(1) Konfliktfreie Anweisungen ohne kritische Bereiche

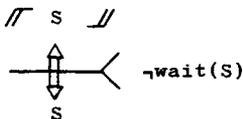
(a) Sequentialisierung/Parallelisierung



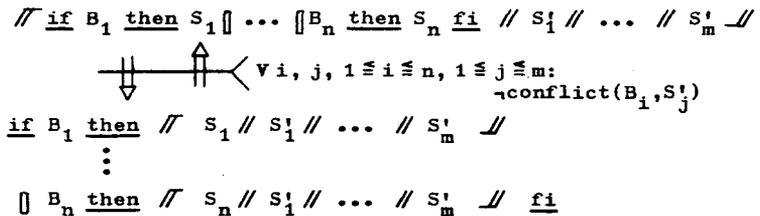
(b) Anfügen/Eliminieren leerer Anweisungen



(c) Eliminieren/Einführen überflüssiger $\parallel - \parallel$ -Klammern



(d) Herausziehen bewachter Anweisungen



Die Spezialisierung auf bedingte Anweisungen ergibt sich trivial.

(2) Bewachte, kritische Bereiche

Sei für $1 \leq i \leq n$:

$$W_i \hat{=}_{\text{def}} \underline{\text{await } B_i \text{ then } K_i \text{ endwait}},$$

wir definieren:

$$\parallel W_1; S_1 \parallel \dots \parallel W_n; S_n \parallel$$



$$\text{if } B_1 \underline{\text{then}} K_1; \parallel S_1 \parallel W_2; S_2 \parallel \dots \parallel W_n; S_n \parallel$$

\vdots

$$\begin{aligned} & \parallel B_n \underline{\text{then}} K_n; \parallel W_1; S_1 \parallel \dots \parallel W_{n-1}; S_{n-1} \parallel S_n \parallel \\ & \underline{\text{else abort}} \end{aligned} \quad \underline{\text{fi}}$$

(3) Lokale Variable

$$\parallel \underline{\text{var}} \underline{m} \ v := E; S_1 \parallel S_2 \parallel \dots \parallel S_n \parallel$$



$\neg \text{occurs}(v \text{ in } S_1, \dots, S_n)$

$$\underline{\text{var}} \underline{m} \ v; \parallel v := E; S_1 \parallel S_2 \parallel \dots \parallel S_n \parallel$$

Durch diese Transformationsregeln und Transformationsinduktion ist die mathematische Semantik paralleler Programme erklärt.

Beispiel: Aus /Owicki 75/, Seiten 52-56:

$$\begin{aligned} x := 0; \parallel & \underline{\text{await true then}} x := x+1 \underline{\text{endwait}} \parallel \\ & \underline{\text{await true then}} x := x+2 \underline{\text{endwait}} \parallel \end{aligned}$$

Die Regel (2) liefert:

$$\begin{aligned} x := 0; \text{if } \underline{\text{true then}} & x := x+1; \parallel \underline{\text{nop}} \parallel \underline{\text{await true then}} x := x+2 \\ & \underline{\text{endwait}} \parallel \\ \parallel \underline{\text{true then}} & x := x+2; \parallel \underline{\text{await true then}} x := x+1 \underline{\text{endwait}} \\ & \parallel \underline{\text{nop}} \parallel \\ & \underline{\text{else abort}} \end{aligned} \quad \underline{\text{fi}}$$

Die Anwendung der Regeln (1b), (2) und (1c) ergibt zusammen mit der Regel für bewachte Ausdrücke mit true als Wächter und der Regel für nop:

$x := 0; \overline{x} := x+1; x := x+2, \square \overline{x} := x+2; x := x+1,$

Triviale Umformungen führen auf:

$x := x+3$

Im Gegensatz zu /Owicki 75/ werden keinerlei Hilfsvariablen ("Auxiliary Variables") benötigt, um die Bedeutung des Programms "auszurechnen".

4.3.3 Beweisen durch Transformationen

Um über Programme mit Wiederholungsanweisungen und rekursiven Prozeduren mit Hilfe von Transformationen Beweise führen zu können, setzen wir Induktionsmethoden ein. Neben Transformationsinduktion kann man auch ganz gewöhnliche Induktionsmethoden einsetzen:

Beispiel: Parallele Berechnung der Summe $\sum_{i=1}^m a(i)$

Sei die Funktion padd definiert durch

```

funct padd = (nat n, int j1, int j2, rat y1, rat y2)rat:
  ( var int i, var int i1, var int i2, var rat x1, var rat x2 ) :=
    ( n, j1, j2, y1, y2 );
  // sum(i1,x1) // sum(i2,x2) //;    x1 + x2

  where
    proc sum = (var nat k, var rat x):
      ( await true then ( i, k ) := ( i-1, i ) endwait;
        if k  $\geq$  1 then x := x + a(k);    sum( k, x ) fi
    )

```

Lemma: $\text{padd}(n, j1, j2, y1, y2) = y1 + y2 + \sum_{i=1}^n a(i)$

Beweis: Induktion über n

Falls $n = 0$ liefert triviales Ausrechnen die Behauptung.

Falls $n \geq 0$ und die Behauptung für n zutrifft, dann erhalten wir durch UNFOLD, Regel (2) und Regel (1d) für $\text{padd}(n+1, j1, j2, y1, y2)$:

$$\overline{(\text{var int } i, \text{var int } i1, \text{var int } i2, \text{var rat } x1, \text{var rat } x2) := (n+1, j1, j2, y1, y2);}$$

$$\overline{\{ i, i1 \} := (i-1, i); x1 := x1 + a(i1) \} \parallel}$$

$$\overline{\{ i, i2 \} := (i-1, i); x2 := x2 + a(i2) \} \parallel};$$

$$\overline{\parallel \text{sum}(i1, x1) \parallel \text{sum}(i2, x2) \parallel};}$$

$$x1 + x2 \quad \quad \quad _$$

Eine Anwendung des Distributivgesetzes für die endliche Auswahl, ein UNFOLD für die Zuweisungen und ein FOLD für die Funktion padd ergibt:

$$\text{padd}(n, n+1, j2, y1+a(n+1), y2) \parallel \text{padd}(n, j1, n+1, y1, y2+a(n+1))$$

Nach Induktionsvoraussetzung ist dieser Ausdruck äquivalent zu

$$(y1+a(n+1)) + y2 + \sum_{i=1}^n a(i) \parallel y1 + (y2+a(n+1)) + \sum_{i=1}^n a(i)$$

Also gilt die Behauptung. □

Genaugenommen haben wir gezeigt, daß die Funktion padd der Rekursionsgleichung

$$\text{funkt padd} = (\text{nat } n, \text{int } j1, \text{int } j2, \text{rat } y1, \text{rat } y2) \text{rat:}$$

$$\text{if } n = 0 \text{ then } y1 + y2$$

$$\text{else padd}(n-1, n, j2, y1+a(n), y2) \parallel$$

$$\text{padd}(n-1, j1, n, y1, y2+a(n)) \quad \text{fi}$$

genügt. Über diese Version kann eine Programmentwicklung für padd führen. Die dazu benötigten Transformationsregeln stimmen mit den im Beweis verwendeten Regeln überein.

Dies zeigt, wie eng Verifikation und Programmentwicklung verwandt sind. Die Transformationsregeln aus dem Abschnitt 4.3.2 reduzieren Programme mit parallelen Sprachelementen auf nichtdeterministische, sequentielle Programme. So läßt sich damit auch Dijkstras Kalkül der schwächsten Vorbedingungen (vgl. /Dijkstra 76/) auf parallele Programme übertragen. Der Zusammenhang zwischen Transformationsregeln und Verifikationsregeln wird in /Pepper 79/ behandelt. Verifikationsregeln für eine eingeschränkte Sprache für die parallele Programmierung enthält /Flon, Suzuki 78a/.

4.3.4 Abgeleitete Transformationsregeln

Aus den definierenden Transformationsregeln lassen sich durch Komposition und Induktion weitere Regeln ableiten. Wir wollen hier einige davon angeben und ihre Korrektheit beweisen.

Lemma: Die folgende Regel ist ableitbar:

$$\begin{array}{c} \parallel S_1; T_1 \parallel \dots \parallel S_n; T_n \parallel \\ \\ \begin{array}{c} \text{---} \\ \parallel \\ \downarrow \end{array} \begin{array}{c} \diagdown \\ \diagup \end{array} \neg \text{wait}(S_1) \wedge \dots \wedge \neg \text{wait}(S_n) \\ \\ \parallel S_1 \parallel \dots \parallel S_n \parallel; \parallel T_1 \parallel \dots \parallel T_n \parallel \end{array}$$

Beweis: Nach der Kontextbedingung in 4.3.1 müssen die S_i paarweise konfliktfrei sein. n-maliges Anwenden von (1a) ergibt $S_1; \dots; S_n; \parallel T_1 \parallel \dots \parallel T_n \parallel$. Die Anwendung von (1c) auf S_n und sukzessives Anwenden von (1b) und (1a) in umgekehrter Richtung liefert die gewünschte Form.¹⁾ □

Man beachte, daß für konfliktfreie Zuweisungen in Abschnitt 4.2 schon die Zurückführung auf die Kollektivzuweisung erklärt ist. Beide Definitionen sind verträglich.

Lemma: Die folgende Regel ist ableitbar:

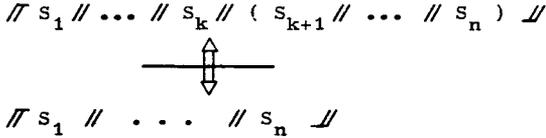
$$\begin{array}{c} \parallel S_1; S_2 \parallel \dots \parallel S_n \parallel \\ \\ \begin{array}{c} \text{---} \\ \parallel \\ \downarrow \end{array} \begin{array}{c} \diagdown \\ \diagup \end{array} \neg \text{conflict}(S_1, S_2) \wedge \neg \text{wait}(S_1) \\ \\ \parallel S_1 \parallel \dots \parallel S_n \parallel \end{array}$$

Beweis: Anwenden der Regeln (1a), (1b) und der Regel (1a) in umgekehrter Richtung.¹⁾ □

¹⁾ Wieder wird bei der Ableitung das "neutrale" Element nop und die Regeln dafür (vgl. Abschnitt 0) verwendet.

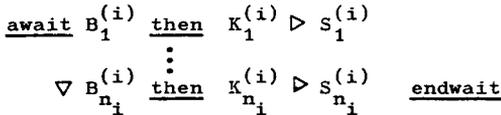
4.3.5 Erweiterungen bewachter, kritischer Bereiche

Wir erlauben nun mehrere parallele Anweisungen zu klammern:

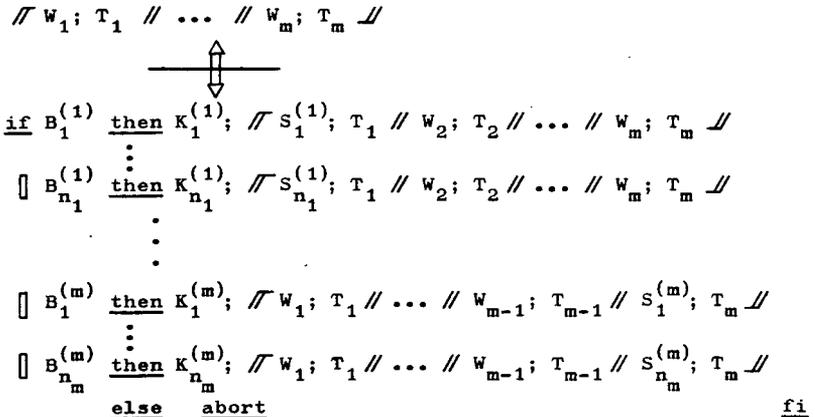


Man beachte, daß im Gegensatz zur Regel in Abschnitt 4.2.1 keine Nebenbedingungen für die S_{k+1}, \dots, S_n gefordert werden. Durch die obige Transformationsregel werden alle Regeln (1) aus Abschnitt 4.3.2 auf Anweisungen der Form $(S_1 // \dots // S_n)$ übertragen.

Die bewachten, kritischen Bereiche verallgemeinern wir zu $W_i = \text{def}$

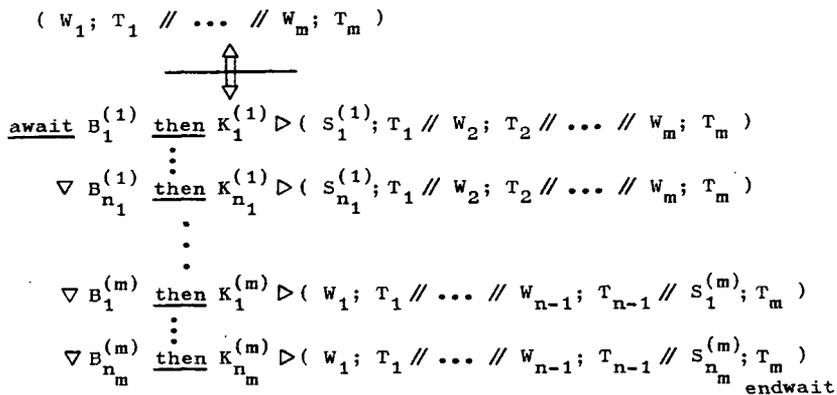


Dabei werden die Anweisungen $S_j^{(i)}$ nicht mehr "geschützt" ausgeführt, wie die folgende Transformationsregel erkennen läßt. Wir definieren:



Die mehrfach bewachten, kritischen Bereiche sind damit ein Sprach-element für disjunktives Warten mit kritischer und unkritischer Alternative.

Für mit runden Klammern zusammengefaßte, parallele Anweisungen ergibt sich hingegen:



Für Programme mit geschachtelter Parallelität muß die Kontextbedingung entsprechend erweitert werden.

Konvention: Kontextbedingung

In Parallelanweisungen $// S_1 // \dots // S_n //$ oder $(S_1 // \dots // S_n)$ muß stets gelten: Tritt eine Programmvariable v in S_i außerhalb eines bewachten, kritischen Bereichs, in einem bewachten kritischen Bereich nach dem \triangleright -Symbol oder innerhalb einer weiteren $// - //$ -Klammer auf, so darf in S_j mit $i \neq j$ keine Zuweisung an v auftreten.

Parallelanweisungen $(S_1 // \dots // S_n)$ mit $\text{wait}(S_i)$ dürfen nur im Inneren von $// - //$ -Klammern auftreten.

$\text{wait}(A)$ bedeutet nun: In A treten bewachte, kritische Bereiche außerhalb von $// - //$ -Klammern auf.

Die Konvention stellt sicher, daß bei der Anwendung der Transformationsregeln keine Programme entwickelt werden können, in denen nicht konfliktfreie Anweisungen ungeschützt parallel auftreten. Für Programme, die die Konvention erfüllen, gilt die Konvention auch nach der Anwendung von Transformationsregeln, da die Konvention unter Transformationen invariant ist.

Ein Beispiel für die Verwendung mehrfach bewachter, kritischer Bereiche findet sich in Abschnitt 6.2.

5 Spezielle Systeme paralleler Programme

Im folgenden wollen wir - neben speziellen Sprachelementen für die Kommunikation - Fragen des Nichtdeterminismus, der Fairness und unbeschränkt laufender Programme behandeln. Damit nähern wir uns klassischen Fragestellungen aus dem Bereich der Systemprogrammierung.

5.1 Kommunikation zwischen parallel ablaufenden Programmen

Kommunikation, d.h. der Austausch von Informationen, ist eines der zentralen Anliegen der parallelen Programmierung. Wie im Abschnitt 3.2 und 3.4 demonstriert wird, können Kommunikationsvorgänge auf der applikativen Ebene ohne den Gebrauch von Programmvariablen beschrieben werden. Andererseits können die Sprachelemente für die Kommunikation durch "gemeinsame" Programmvariable dargestellt ("implementiert") werden.

Für die Kommunikationstabellen der applikativen Ebene entwickeln wir Kommunikationsströme als ein prozedurales Gegenstück. Sie werden aus dem abstrakten Typ QUEUE (Warteschlangen) abgeleitet. Andere abstrakte Typen würden andere Kommunikationsmechanismen ergeben.

5.1.1 Resultatbezeichnungen und parallele Anweisungen

Resultatbezeichnungen können natürlich auch in prozeduralen Programmen verwendet werden. Da sie Programmvariablen stark ähneln (vgl. die Transformationsregeln in Abschnitt 3.1 und in /Pepper 79/), kann man sie auch über Programmvariablen implementieren:

<u>result</u> <u>m</u> <u>x</u>	→	<u>var</u> (<u>m</u> <u>atomic free</u>) <u>x</u> := <u>free</u>
<u>x is</u> <u>E</u>	→	<u>⌈m</u> <u>x'</u> = <u>E</u> ; <u>await</u> <u>x = free</u> <u>then</u> <u>x := x'</u> <u>endwait</u> ,
<u>x</u>	→	<u>⌈var</u> <u>m</u> <u>x'</u> ; <u>await</u> <u>x ≠ free</u> <u>then</u> <u>x' := x</u> <u>endwait</u> ; <u>x'</u>

Diese Korrespondenz zeigt die starke Verwandtschaft des Konzepts der Resultatbezeichnungen zu dem Konzept der "Single Assignment Variables" (vgl. /ONERA CERT 78/).

5.1.2 Kommunikationsströme

In Analogie zu den Kommunikationstabellen führen wir nun Kommunikationsströme ein. Ein Kommunikationsstrom entspricht einem "Kanal", der Objekte überträgt und auf Anforderung an gewisse Prozesse weitergibt.

Ein Strom wird vereinbart durch:

stream m s

Ein Objekt wird durch den Strom übertragen durch

send E on s

und empfangen durch

receive v from s

wobei v eine (lokale) Variable der Art m bezeichne. Sowohl die Sende- als auch die Empfangsanweisung üben einen Seiteneffekt auf den Strom aus. An die Stelle von v darf auch eine Resultatbezeichnung der Art m treten.

Wie Kommunikationstabellen lassen sich Ströme durch einen abstrakten Typ charakterisieren:

type QUEUE = (mode m) queue, empty, isempty, send, remove,
next:

mode queue,

funct queue empty,

funct (queue) bool isempty,

funct (queue, m) queue send,

funct (queue s: \neg isempty(s)) queue remove,

funct (queue s: \neg isempty(s)) m next,

law isempty(empty) = true,

law isempty(send(s,x)) = false,

law remove(send(empty,x)) = empty,

law next(send(empty,x)) = x,

law \neg isempty(s) \Rightarrow remove(send(s,x)) = send(remove(s),x),

law \neg isempty(s) \Rightarrow next(send(s,x)) = next(s)

end of type

Beispiel: Paralleldeklaration von Objektbezeichnungen

Die Paralleldeklaration einer Objektbezeichnung kann durch Ströme simuliert, bzw. implementiert werden. Seien dabei stream m cx und stream bool signal global vereinbart.

```
m x = E; var bool s; receive s from signal;  
while s do send x on cx; receive s from signal od
```

Dieses Programm berechnet den Wert von E parallel zu beliebigen anderen Programmen und stellt ihn auf die Anforderung

```
send true on signal
```

zur Verfügung, so daß er durch die Anweisung

```
receive y on cx
```

auf eine Variable oder Resultatbezeichnung übertragen werden kann. Eine Anweisung send false on signal beendet die Möglichkeit einer Übertragung.

Beispiel: Parallel benutzbare, gemeinsame Programmvariable

Läßt man in dem obigen Beispiel auch ein Ändern des Wertes von x zu, so erhält man eine Simulation bzw. Implementierung einer parallel benutzbaren Programmvariablen. Seien stream m cx, stream m vx und stream signal signal mit mode signal = atomic{put, get, stop} global vereinbart. Das Programm

```
var m v := E; var signal s; receive s from signal;  
while s ≠ stop do if s = get then send v on cx  
    [] s = put then receive v from vx fi  
    receive s from signal od
```

kann parallel zu beliebigen Programmen benutzt werden und stellt auf die Anforderung

```
send get on signal; receive y from cx
```

den Wert der lokalen Variablen v zur Verfügung. Auf die Anweisung

```
send put on signal; send E' on vx
```

wird der Wert von vx zu E' geändert. Die Anweisung send stop on signal beendet die Möglichkeit einer Änderung/Übertragung.

5.1.4 Kommunikation über Schlüssel

Beteiligen sich mehrere Prozesse an der Kommunikation über einen Strom, so kann die spezifisch für einen bestimmten Prozeß gesendete Information nicht gezielt durch diesen Prozeß abgerufen werden. Die Information ist weder vor dem unbeabsichtigten, noch vor dem unbefugten Zugriff durch konkurrierende Prozesse geschützt. Abhilfe verspricht ein individuellerer Kommunikationsmechanismus, der durch Verallgemeinerung gewöhnlicher Ströme zu Strömen mit individuellen Schlüsseln entsteht. Solch ein Strom mit "verschlüsselten" Informationen wird vereinbart durch:

key k stream m s

wobei k die Art der Schlüssel und m die Art der übertragenen Nachrichten bezeichnet. Das Senden einer Nachricht E unter dem Schlüssel K erfolgt durch:

send E key K on s,

das Empfangen einer Nachricht unter dem Schlüssel K durch die Variable (oder Resultatbezeichnung) v geschieht durch:

receive v key K from s.

Die Bedeutung dieser Anweisungen kann formal wieder mit Hilfe eines abstrakten Typs definiert werden.

```
type KEYSTREAM = (mode m, mode k, funct(k,k)bool eq) kstream,  
kisempty, kempty, ksend, kremove, knext:  
mode kstream,  
funct kstream kempty,  
funct(kstream,k)bool kisempty,  
funct(kstream,k,m)kstream ksend,  
funct(kstream s, k k: ~kisempty(s,k))kstream kremove,  
funct(kstream s, k k: ~kisempty(s,k))m knext,
```


5.2 Unbeschränkter Nichtdeterminismus

Da bei parallelen Programmen die zeitliche Abfolge der einzelnen Kommunikationsvorgänge nicht vollständig festgelegt ist, entsteht das Phänomen des Nichtdeterminismus. Da das Resultat eines Programms von der Reihenfolge der Kommunikation abhängen kann, können die Programme auch nichtdeterminiert sein, d.h. mehrere mögliche Resultate besitzen. Ist sogar die Terminierung eines Programms nicht determiniert, so bezeichnen wir es als unbeschränkt nichtdeterminiert.

5.2.1 Beschränkter und unbeschränkter Nichtdeterminismus

Analog zu /Broy et al. 78b/ bezeichnen wir ein Programm als deterministisch, wenn es keine nichtdeterministischen Sprach-elemente, wie Auswahl, bewachte Ausdrücke oder Anweisungen, Kommunikationstabellen und await-Anweisungen, enthält. Ein Programmausdruck heißt determiniert, wenn seine Auswertung nicht terminiert oder jede Auswertung terminiert und den gleichen Wert liefert. Trivialerweise ist jedes deterministische Programm determiniert. Da die Frage, ob ein Programm nicht-deterministisch ist, nach dieser Definition von syntaktischer Natur ist, ist sie statisch entscheidbar. Dagegen kann die Determiniertheit eines nichtdeterministischen Ausdrucks statisch nicht entschieden werden. Die Probleme verschärfen sich noch, wenn man unbeschränkt nichtdeterminierte Programme betrachtet.

Definition: Ein Ausdruck $E \in \text{EXP}$ heißt unbeschränkt nichtdeterminiert, wenn gilt

$$\neg d(E) \wedge |B(E)| > 0$$

Da schon für deterministische Programme das Terminierungsproblem nicht entscheidbar ist, ist auch allgemein nicht entscheidbar, ob ein Programm unbeschränkt nichtdeterminiert ist. Deshalb kann unbeschränkte Nichtdeterminiertheit nicht syntaktisch ausgeschlossen werden, sondern muß semantisch bewältigt werden.

Dementsprechend werden in /Plotkin 79/ bei der Definition einer denotationellen und einer operativen Semantik für Dijkstras Sprache unbeschränkt nichtdeterminierte Programme mathematisch mit abort gleichgesetzt. Operativ bedeutet das, daß im Falle einer Auswahlentscheidung nicht eine Alternative ausgewählt wird, sondern alle Alternativen baumartig durchlaufen werden. Existiert ein nichtterminierender Zweig, so terminiert konsequenterweise das gesamte Programm nicht. Damit stimmen axiomatische, operative und mathematische Semantik überein - der beabsichtigte Effekt wird erreicht. Allerdings ist man damit von der für die parallele Programmierung typische zufällige Auswahl eines Ablaufs abgewichen.

Kombiniert man die Technik schwächster Vorbedingungen mit Hoares Kalkül der Zusicherungen, so kann man auch in der axiomatischen Semantik unterschiedliche Eigenschaften unbeschränkt nichtdeterministischer und stets nicht terminierender Programme behandeln.

5.2.3 Lokaler und globaler Nichtdeterminismus

Resultiert die Nichtdeterminiertheit eines Programm(ausdruck)s aus einer bewachten Anweisung, einem bewachten Ausdruck oder einer endlichen Auswahl, so sprechen wir von lokalem Nichtdeterminismus. Ist jedoch die nicht bestimmte zeitliche Abfolge paralleler Aktionen die Ursache der Nichtdeterminiertheit, so sprechen wir von globalem Nichtdeterminismus.

Die Beschränkung der Wahlfreiheit bei globalem Nichtdeterminismus durch gewisse Nebenbedingungen bezeichnet man als Fairnessbedingungen. Da durch die Regeln im Abschnitt 4.3 globaler Nichtdeterminismus auf lokalen Nichtdeterminismus zurückgeführt wird, und für lokalen Nichtdeterminismus Fairnessbeschränkungen wenig Sinn haben, kann in diesem Ansatz Fairness nicht berücksichtigt werden. Unabhängig davon ergeben sich jedoch ohnehin theoretische Probleme mit allgemeinen Fairnessbeschränkungen.

5.3 Koordination und Fairness

Gewisse Aktionen parallel ablaufender Prozesse erfordern eine Koordinierung. Beispielweise müssen konkurrierende, nicht konfliktfreie Anweisungen in einer nichtdeterminierten Weise sequenzialisiert werden.

Konkurrieren in einem System nichtterminierender, paralleler Prozesse ständig mehrere Prozesse um die Durchführung gewisser "exklusiver" Aktionen, so kann, wenn über die Auswahl des jeweils nächsten Prozesses nichts vorgeschrieben ist, ein Prozeß immer wieder von anderen Prozessen überholt werden und somit selbst nie zum Zuge kommen ("starvation", "dynamic blocking"). Um solche Effekte auszuschließen werden häufig sogenannte Fairnessforderungen aufgestellt (z.B. "first-come-first-served"). Allerdings ergeben sich aus allgemeinen Fairnessforderungen gewisse Probleme für die formale Definition der Semantik.

5.3.1 Kritische Phasen, gegenseitiger Ausschluß, Synchronisation

Aktionen paralleler Prozesse, die nicht parallel ausgeführt werden können oder dürfen, nennt man kritische Phasen. Solche Aktionen benötigen gegenseitigen Ausschluß.

Definition: Aktionen a_1, a_2 eines Ablaufs heißen gegenseitig ausgeschlossen, wenn sie bzgl. jeder möglichen Ablaufordnung sequentiell sind, d.h. wenn stets gilt:

$$a_1 \text{ } \overset{c}{\text{t}} \text{ } a_2 \vee a_2 \text{ } \overset{c}{\text{t}} \text{ } a_1$$

Schon in frühen Arbeiten über Parallelität finden sich Lösungsvorschläge für das Synchronisierungsproblem der Sicherstellung des gegenseitigen Ausschlusses (vgl. /Dijkstra 65/). Neben der Vermeidung von letztlich "Hardware"-bedingten Konflikten ist man auch an der ununterbrechbaren ("konsistenten") Änderung gemeinsamer Variabler interessiert (vgl. Abschnitt 4.2.3).

Dies führt zur Begriffsbildung "unteilbare Aktion".

Definition: Sei P ein Programm. Eine Aktion $act(A) \in actions(P)$ heißt unteilbar, wenn

$$\begin{aligned} \forall act(B) \in actions(P): \\ act(B) \in actions(A) \vee act(A) \in actions(B) \vee \\ act(B) \stackrel{t}{\sqsubseteq} act(A) \vee act(A) \stackrel{t}{\sqsubseteq} act(B). \end{aligned}$$

Schon aus der Definition läßt sich erkennen, daß es sich wieder um das Problem des gegenseitigen Ausschlusses gewisser Aktionen handelt.

Aus dem bisher Gesagten ergibt sich, daß gerade Kommunikationsvorgänge eine Synchronisation benötigen, daß aber andererseits Synchronisation eine (eingeschränkte) Kommunikationsmöglichkeit voraussetzt. So stellen die ersten Vorschläge für Synchronisationsprimitive, wie "test-and-set" durch Koordinationsvariable oder Semaphore mit ihren P- und V-Operationen, stets eingeschränkte Kommunikationsmechanismen in den Vordergrund. Spätere Ansätze hingegen arbeiten eher mit allgemeineren Kommunikationsmechanismen, in denen man auch die obigen Vorschläge darstellen kann. Einen wenig kommunikationsorientierten Vorschlag stellen die "Pathexpressions" (vgl. /Campbell, Habermann 74/, /Lauer, Campbell 75/) dar, die in gewisser Weise auf Petrinetze zurückgeführt werden können.

Bei Kommunikationsmechanismen erhebt sich zusätzlich die Frage, ob man im Falle eines Senderversuchs den Sender warten läßt, bis der Empfänger empfangsbereit ist, und erst nach Abschluß des Kommunikationsvorgangs Sender und Empfänger fortsetzen läßt (vgl. /Hoare 78/), d.h. ob der Kommunikationsvorgang in Form eines "Rendezvous" stattfindet, oder ob der Sender die Empfangsbereitschaft des Empfängers nicht abwartet, sondern die Nachricht "deponiert". Im ersten Fall erhält man Warteschlangen sendewilliger Prozesse, im zweiten Fall Warteschlangen von Nachrichten.

5.3.2 Fairness

Der Begriff der Fairness wird in der Literatur mit leicht unterschiedlichen Bedeutungen gebraucht. Wir wollen Fairness immer im Sinne von "First-Come-First-Served" bezogen auf den globalen Nichtdeterminismus parallel ablaufender Prozesse verstehen.

Einen in Bezug auf lokalen Nichtdeterminismus manchmal anzutreffenden Fairnessbegriff "alle Auswahlentscheidungen sind gleichwahrscheinlich" wollen wir schon deshalb nicht verwenden, weil damit der lokale Nichtdeterminismus wahrscheinlichkeits-theoretischen Überlegungen zugänglich würde. Die für die Programmentwicklung bedeutungsvolle Verfeinerung durch Einengung der Auswahl wäre mit solchen Vorstellungen unverträglich.

Da in Abschnitt 4.3 globaler Nichtdeterminismus auf lokalen Nichtdeterminismus zurückgeführt wird, ist in diesem Zusammenhang die formale Definition von Fairness schwierig. Ein formaler Fairnessbegriff läßt sich jedoch mit Hilfe der Ablaufordnungen definieren. In Bezug auf die Kommunikationstabellen aus Kapitel 3 bedeutet das:

Definition: Sei P ein Programm der applikativen Ebene, das Kommunikationstabellen verwendet. Eine Ablauf b von P heißt "fair nach n Schritten" bzgl. einer gegebenen Ablaufordnung, wenn für jede Aktion a der Form put E on c bzgl. einer Kommunikationstabelle c gilt: Es gibt höchstens n Aktionen a_i , $1 \leq i \leq n$, der Form put E_i on c, so daß gilt:

$$\neg(\text{act}(E_i) \sqsubseteq_z \text{act}(E)) \wedge \text{act}(E) \sqsubseteq_z \text{act}(E_i) \wedge a_i \sqsubseteq_z a$$

Die Definition besagt, daß ein sendebereiter Prozeß höchstens von n Prozessen "überholt" werden darf. Wir sprechen von allgemeiner Fairness, wenn gesagt wird: Für jede Kommunikationstabelle existiert ein $n \in \mathbb{N}$, so daß jeder Ablauf fair nach n Schritten ist. Ähnliche Fairnessbedingungen können auch für Ströme oder bewachte, kritische Bereiche formuliert werden.

5.3.3 Fixpunktsemantik, Fairness und nichtstetige Funktionale

Betrachtet man die Aussagen in /Broy et al. 78b/, so ergibt sich (ganz in Übereinstimmung mit der intuitiven Vorstellung), daß alle in CIP-L als operativ bezeichneten Sprachelemente in der Egli-Milner-Ordnung stetig sind, und daß gerade die "unendlichen", als nichtoperativ bezeichneten Teile, wie Quantifizierung, Komprehension und unendliche Auswahl, nicht stetig sind (vgl. auch /Manna et al. 73/). Das deckt sich völlig mit der Vorstellung, daß mit stetigen Sprachelementen gebildete Funktionale zur Definition rekursiver Funktionen in ihrem schwächsten Fixpunkt mit dem Grenzwert der Funktionaliteration übereinstimmen, und somit "berechenbare" Funktionen charakterisieren. Für nichtstetige Funktionale stimmen Grenzwert der Funktionaliteration und schwächster Fixpunkt nicht zwangsläufig überein. Damit erhält man eine Diskrepanz zwischen Berechnungsregeln und Fixpunktsemantik.

Z. B. ist in CIP-L der Ausdruck some nat x: x > 0 kein operativer Ausdruck. Betrachtet man jedoch das folgende Programm (Beispiel nach /Park 79/):

```

proc q =: // await true then x := 1 await //
           while z = 0 do y:=y+1;
           await true then z:=x await od //,
  (var nat x, var nat y, var nat z) := ( 0, 0, 0);

q;      y      ─

```

so liefern die Ableitungsregeln in Abschnitt 4.3 für q:

```

if z = 0 then y:=y+1; x:=1; // nop // while ... od //
           z := x; q
           else nop                                     fi

```

Dies ist im Kontext des obigen Programms ($z = y = x = 0$) äquivalent zu:

```

y:=y+1; x:=1; y:=y+1; z:=x; z := x; q

```

Damit erkennt man, daß q einen unendlichen Ablauf besitzt.
Es gilt:

$$B(P) = \mathbb{N} \setminus \{0, 1\} \quad \text{und} \quad d(P) = \text{FALSE}$$

wobei P das obige Programm bezeichne. P ist folglich unbeschränkt nichtdeterministisch.

Erhebt man jedoch die allgemeine Forderung nach Fairness, so kommt der Prozeß, der x auf 1 setzt, nach endlicher Zeit zum Zug und es gilt $d(P) = \text{TRUE}$.

Damit erhält man jedoch einen Ausdruck P , der zu dem nicht-operativen, nichtstetigen Ausdruck some nat $x: x > 2$ äquivalent ist. Daß dieser Ausdruck nichtstetig ist, erkennt man aus dem Beispiel:

```
funct g = (nat x )nat: if x = 0 then g(some nat z: z > 0)
                    [] x = 1 then 1
                    [] x > 1 then g(x-1)+1           fi
```

Sei die obige rekursive Definition abgekürzt durch $g = T[g]$.
Sei nun g_0 die völlig undefinierte Funktion, d.h.

$$\forall x \in \mathbb{N}: B(g(x)) = \emptyset \wedge \neg d(g(x))$$

und $g_{i+1} = T[g_i]$. Es gilt

$$B(g_i(x)) = \begin{cases} \{1, \dots, i-1\} & \text{falls } x = 0 \\ \{x\} & \text{falls } 0 < x \leq i \\ \emptyset & \text{falls } i < x \end{cases}$$

$$d(g_i(x)) = \begin{cases} \text{FALSE} & \text{falls } (x = 0) \vee (x > i) \\ \text{TRUE} & \text{falls } 0 < x \leq i \end{cases}$$

folglich gilt für $g_\infty = \text{lub} \{g_i\}$

$$B(g_\infty(x)) = \begin{cases} \mathbb{N} \setminus \{0\} & \text{falls } x = 0 \\ \{x\} & \text{falls } x > 0 \end{cases}$$

$$d(g_\infty(x)) = \begin{cases} \text{FALSE} & \text{falls } x = 0 \\ \text{TRUE} & \text{falls } x > 0 \end{cases}$$

Die Funktion g_{∞} ist jedoch kein Fixpunkt von T , da zwar $B(g_{\infty}(x)) = B(T[g_{\infty}](x))$, aber $d(g_{\infty}(0)) = \text{FALSE}$ und $d(T[g_{\infty}](0)) = \bigwedge_{x \in \mathbb{N} \setminus \{0\}} d(g_{\infty}(x)) = \text{TRUE}$.

Also ist T nicht stetig, da sonst g_{∞} Fixpunkt von T sein müßte. Übrigens gilt $T[g_{\infty}] = T[T[g_{\infty}]]$, d.h. $T[g_{\infty}]$ ist Fixpunkt - genauer gesagt sogar schwächster Fixpunkt von T .

Da $d(g_{\infty}(0)) = \text{FALSE}$ terminiert ein Aufruf von $g_{\infty}(0)$ nach unserer Semantikdefinition nicht zwangsläufig. Wendet man jedoch auf einen Aufruf $g(0)$ Transformationsregeln an, so gilt:

$$g(0)$$

ist nach UNFOLD und Vereinfachungen äquivalent zu

$$g(\text{some nat } z: z > 0)$$

Dieser letzte Ausdruck ist jedoch auch für $g = g_{\infty}$ äquivalent zu

$$\text{some nat } z: z > 0$$

Das zeigt, daß eine Anwendung von Transformationsregeln für Funktionsaufrufe von g zum schwächsten Fixpunkt $T[g_{\infty}]$ von T und nicht zum Grenzwert g_{∞} der Funktionaliteration führen kann.

5.3.4 Bemerkungen zur Stetigkeit und Fairnessbedingungen

Das eben behandelte Beispiel zeigt, daß bei Annahme einer allgemeinen Fairnessbedingung nichtstetige Funktionale mit "operativen" Sprachelémenten formulierbar werden. Die Diskrepanz zwischen Grenzwert der Funktionaliteration und schwächstem Fixpunkt läßt sich zwar z.B. durch transfiniten Funktionaliteration lösen, aber dieser Ansatz entfernt sich wohl zu weit von dem intuitiven Begriff der "Berechenbarkeit". Außerdem läßt sich zu jedem festen Verfahren transfiniten Funktionaliteration ein Funktional angeben, so daß das Verfahren für dieses Funktional nicht den (schwächsten) Fixpunkt des Funktionals liefert.

Man beachte, daß das hier angesprochene Problem der Nichtstetigkeit gewisser Fairnessbedingungen nicht einfach durch das Ersetzen der Egli-Milner Ordnung durch eine andere Ordnung gelöst werden kann. Betrachtet man nämlich neben der Funktion g aus Abschnitt 5.3.3 die Funktion g' :

```
funct  $g' = (\text{nat } x) \text{ nat} : \text{if } x = 0 \text{ then } g'(\text{some nat } z : \text{true})$   
     $\square$   $x = 1 \text{ then } 1$   
     $\square$   $x > 1 \text{ then } g'(x-1)+1$            fi
```

so erkennt man, daß g' exakt die gleiche Folge von Funktionen g_i bei der Funktionaliteration besitzt, und daß g_∞ aus Abschnitt 5.3.3 schwächster Fixpunkt des obigen Funktionals ist, durch das g' rekursiv definiert wird. Dies zeigt, daß jede Ordnung, unter der das obige Funktional monoton ist, nur g_∞ als Grenzwert der Funktionaliterationsfolge g_i liefern kann, da dann stets g_∞ schwächster Fixpunkt der obigen Gleichung bleibt.

Es bleibt die Möglichkeit, einen komplexeren Begriff für die rekursiven Funktionen einzuführen und damit eine feinere Äquivalenzrelation auf der Menge der rekursiv definierbaren Funktionen. Damit bewegt man sich in die Richtung der operativen Semantik und der operativen Äquivalenz (vgl. Kapitel 1 und /Back 79/).

Eine andere Alternative ist, ganz auf Fairnessbedingungen zu verzichten und so den auftretenden Problemen völlig aus dem Weg zu gehen. Eine faire Synchronisation läßt sich nämlich unter gewissen, sehr schwachen Annahmen bereits durch Semaphore formulieren (vgl. /Morris 79/). So sind in den Kapiteln 1 - 4 keinerlei Fairnessforderungen in die Semantikdefinition eingegangen.

Zu einem ähnlichen Schluß - wenn auch über eine völlig andere Argumentation - kommt auch /Hoare 78/: "The question arises: Should a programming language definition specify that an implementation must be fair? Here, I am fairly sure that the answer is NO."

5.4 Unendliche Abläufe und nichtterminierende Prozeduren

Im Zusammenhang mit rekursiven Prozeduren treten keine spezifischen Problem für Parallelanweisungen auf. Betrachtet man jedoch nichtterminierende, rekursive Prozeduren, wie sie zur Modellierung "unendlich" lang laufender Dienstleistungsprogramme in Betriebssystemen auftreten, so liefert die übliche Fixpunktsemantik keine befriedigende Bedeutung, da solche Programme stets mit undefiniert bzw. mit abort identifiziert werden.

Deshalb ordnen wir gewissen, nichtterminierenden Prozeduren, die von der Form "endständiger Rekursion" (vgl. Tail-Rekursion in /Bauer, Wössner 79/) sind, unendliche Abläufe zu. Wie demonstriert wird, können solche Programme auch durch Transformationen entwickelt werden. Auch Petrinetze lassen sich durch solche Prozeduren als Programme repräsentieren.

5.4.1 Bedeutung unendlicher Abläufe

Betrachten wir eine Aktion, bzw. Anweisung, die nicht terminiert, so kann sie bei Ausübung eines globalen Effekts durchaus von Interesse für die praktische Programmierung sein:

Beispiel: Sei

$$\text{comfunct } g = (\text{comtable } m \text{ } c, \text{ nat } i) : \\ \quad \text{put } f(i) \text{ on } c; \quad g(c, i+1)$$

Für jede Kommunikationstabelle c der Art comtable m und jede Funktion f der Art funct(nat) m generiert $g(c, 1)$ eine (unendliche) Tabelle der Funktionswerte von f .

Das Beispiel zeigt, daß unendliche Abläufe, bzw. nichtterminierende Prozeduren und Kofunktionen, zum Beispiel zur Beschreibung unendlicher Sequenzen (vgl. /Plotkin 79/) und unendlicher Folgen von Anweisungen verwendet werden können, bzw. ganz allgemein zur Behandlung unendlicher Objekte (vgl. /Bauer 78/).

Für die formale Behandlung nichtterminierender Systeme rekursiv definierter Prozeduren durch unendliche Abläufe führen wir eine partielle Ordnung auf Abläufen ein.

Definition: Seien a und b Abläufe; wir sagen "a ist Anfang von b", im Zeichen $a \sqsubseteq_s b$, wenn

$$\begin{aligned} a \hat{=} b \quad \text{oder} \quad a \hat{=} \underline{\text{nop}} \quad \text{oder} \quad (\text{für } i < n) \\ a \hat{=} \overline{s_1; \dots; s_i; s'_{i+1}} \quad \text{und} \quad b \hat{=} \overline{s_1; \dots; s_n} \quad \text{bzw.} \\ a \hat{=} \underline{\text{if } p \text{ then } s_1; \dots; s_i; s'_{i+1} \text{ fi}} \quad \text{und} \quad b \hat{=} \underline{\text{if } p \text{ then } s_1; \dots; s_n \text{ fi}} \\ \text{und} \quad s'_{i+1} \sqsubseteq s_{i+1} . \end{aligned}$$

Diese Ordnung läßt sich von Abläufen auf beliebige Anweisungen übertragen.

Definition: Seien S_1 und S_2 Anweisungen; wir sagen " S_1 ist Anfang von S_2 ", im Zeichen $S_1 \sqsubseteq_s S_2$, wenn für jede Umgebung ENV gilt:

$$\forall (a, \text{ENV}_1) \in \text{AU}(S_1, \text{ENV}) \exists (b, \text{ENV}_2) \in \text{AU}(S_2, \text{ENV}) : a \sqsubseteq_s b$$

Mit dieser Definition ist nop kleinstes Element für \sqsubseteq_s .

Seien nun p und q rekursiv definierte Prozeduren:

$$\begin{aligned} \underline{\text{proc}} \ p &= (\underline{m}_1 \ x_1, \dots, \underline{m}_n \ x_n) : T_p[p, x_1, \dots, x_n], \\ \underline{\text{proc}} \ q &= (\underline{m}_1 \ x_1, \dots, \underline{m}_n \ x_n) : T_q[q, x_1, \dots, x_n]. \end{aligned}$$

Wir definieren für p und in Analogie für q zwei Iterationsfolgen:

$$\begin{aligned} \underline{\text{proc}} \ P_0 &= (\underline{m}_1 \ x_1, \dots, \underline{m}_n \ x_n) : \underline{\text{abort}}, \\ \underline{\text{proc}} \ P_{i+1} &= (\underline{m}_1 \ x_1, \dots, \underline{m}_n \ x_n) : T_p[p_i, x_1, \dots, x_n]; \\ \underline{\text{proc}} \ P'_0 &= (\underline{m}_1 \ x_1, \dots, \underline{m}_n \ x_n) : \underline{\text{nop}}, \\ \underline{\text{proc}} \ P'_{i+1} &= (\underline{m}_1 \ x_1, \dots, \underline{m}_n \ x_n) : T_p[p'_i, x_1, \dots, x_n]. \end{aligned}$$

Wir beschränken uns im folgenden auf Prozeduren, für die gilt:

$$\forall n \in \mathbb{N} : P'_i \sqsubseteq_s^* P_i,$$

wobei

$$p \sqsubseteq_s^* q \Leftrightarrow \text{def} \quad \forall y_1, \dots, y_n : p(y_1, \dots, y_n) \sqsubseteq_s q(y_1, \dots, y_n).$$

Dies gilt insbesondere für rekursive Prozeduren von der Form der

terminalen Rekursion ("Tail-Recursion" in /Bauer, Wössner 79/). Gilt die oben angegebene Beziehung für p , dann ist $\{p_i^!\}_{i \in \mathbb{N}}$ eine \subseteq_s -monotone Folge.

Definition: Die Prozedur p heißt Ablaufapproximation der Prozedur q , wenn gilt:

$$\forall i \in \mathbb{N} \exists j \in \mathbb{N}: p_i^! \subseteq_s q_j^!.$$

Ist zusätzlich auch q Ablaufapproximation von p , dann heißen p und q ablaufäquivalent.

Man beachte, daß für terminierende Aufrufe von p und q die Ablaufäquivalenz mit der Ablaufgleichheit zusammenfällt. In Analogie zu /Nivat 79/ definieren wir für unendliche Abläufe a und b , $a \subseteq_s b$ genau dann, wenn $a \subseteq b$.

Definition: Ein (nichtterminierender) Prozeduraufruf $p(y_1, \dots, y_n)$ heißt wirkungsapproximierend zum Aufruf $q(y_1, \dots, y_n)$, falls für jede Umgebung ENV gilt:

$$\forall i \in \mathbb{N} \forall (a, ENV_1) \in AU(p_i^!(y_1, \dots, y_n), ENV): \\ \exists j \in \mathbb{N} \exists (b, ENV_2) \in AU(q_j^!(y_1, \dots, y_n), ENV): \quad a \subseteq_I b.$$

Gilt zusätzlich, daß $q(y_1, \dots, y_n)$ wirkungsapproximierend zu $p(y_1, \dots, y_n)$ ist, so heißen beide Aufrufe wirkungsäquivalent oder funktional äquivalent.

Nichtterminierende Programme sind in der mathematischen Semantik äquivalent, können aber sehr wohl nicht wirkungsäquivalent sein. Setzen wir voraus, daß alle Transformationsregeln im Kapitel 4 nicht nur korrekt im Sinne der mathematischen Semantik, sondern auch korrekt im Sinne der Wirkungsäquivalenz sind, d.h. auch nichtterminierende Programme in wirkungsäquivalente Programme überführen, so wird der Begriff der Wirkungsäquivalenz auf parallele Programme übertragen, obwohl wir auf die Definition von Abläufen für Programme mit bewachten, kritischen Bereichen verzichtet haben. Mit Hilfe der Transformationsinduktion lassen sich dann Systeme paralleler Programme aus nichtdeterministischen, sequentiellen Programmen entwickeln.

Beispiel: Das Erzeuger-Verbraucher Problem

In /Bauer, Wössner 79/ findet sich ein sequentielles Programm für das Erzeuger/Verbraucher Problem:

```
( nat max, nat inventurbestand ) :  
  var nat a := inventurbestand; m  
  where  
    proc e := ⌈ a := a+1; erzeuge; ⌋ ,  
    proc v := ⌈ a := a-1; verbrauche; ⌋ ,  
    proc m := if a > 0 then v  
              ⌈ a < max then e fi ⌋
```

Zusätzlich gilt \neg conflict(erzeuge, verbrauche) und \neg occurs(a in erzeuge, verbrauche). Ein UNFOLD für e und v bringt m genau auf die Form von p_0 in unserer Transformationsregel. Auch die Anwendbarkeitsbedingungen sind erfüllt und wir erhalten:

```
( nat max, nat inventurbestand ) :  
  var nat a := inventurbestand; ma  
  where  
    proc ma := ⌈ er // ve ⌋ ,  
    proc er := ⌈ await a < max then a := a+1 endwait ;  
              erzeuge; er ⌋  
    proc ve := ⌈ await a > 0 then a := a-1 endwait ;  
              verbrauche; ve ⌋
```

Dies zeigt, wie sequentielle, nichtdeterministische Programme durch Transformationen in Parallelprogramme verwandelt werden können, selbst wenn es sich um nicht-terminierende Programme handelt.

5.4.3 Petri-Netze als nichtterminierende parallele Programme

Ein klassisches Beispiel für nichtterminierende Systeme sind Petrinetze (vgl. /Petri 62/, /Peterson 77/). Wir wollen hier eine Vorschrift angeben, wie Petrinetze in nichtterminierende Systeme parallel ablaufender Programme umgeformt werden können.

Sei Q ein Petrinetz mit $Q = (H, S, A, B)$, wobei H eine endliche Menge von Hürden, S eine dazu disjunkte, endliche Menge von Stellen sei, und $A \subseteq H \times S$, $B \subseteq S \times H$. Ferner sei i_s eine Anfangsbelegung der Stellen $s \in S$, $i_s \in \mathbb{N}$.

Für jede Hürde i mit n_i Kanten $B_i \subseteq B$, die in i münden und m_i Kanten $A_i \subseteq A$, die von i ausgehen, definieren wir eine Prozedur:

```

proc hi =: await s1(i) > 0 ∧ ... ∧ sni(i) > 0 then
    co Schalten der Hürde  $i$  co
    (s1(i), ..., sni(i), q1(i), ..., qmi(i)) :=
    (s1(i) - 1, ..., sni(i) - 1, q1(i) + 1, ..., qmi(i) + 1)
    endwait; hi

```

wobei mit jeder Stelle in S eine Variable von der Art nat verbunden wird und $B_i = \{s_1^{(i)}, \dots, s_{n_i}^{(i)}\} \times \{i\}$

$$A_i = \{i\} \times \{q_1^{(i)}, \dots, q_{m_i}^{(i)}\}$$

Damit ergibt sich als Programm für das Petrinetz Q

```

( var nat s1, ..., var nat sn ) := ( i1, ..., in );
// h1 // ... // hm //

```

wobei n die Zahl der Stellen und m die Zahl der Hürden wiedergibt.

Man beachte, daß sich damit die Ergebnisse der Theorie der Petrinetze auf Programme übertragen lassen.

Bei dieser Umsetzung von Petrinetzen in parallele Programme wird von einem streng sequentiellen Schalten der Hürden ausgegangen. Dies kann jedoch nicht voll befriedigen, da ja gerade Petrinetze als Modelle parallel ablaufender Prozesse Verwendung finden. Deshalb wollen wir nun von dieser "quasi-parallelen" Auffassung von Petrinetzen zu einer stärker parallelen Auffassung übergehen. Dazu definieren wir für jede Hürde i :

```

proc  $h_i^!$  := await  $s_1^{(i)} > 0 \wedge \dots \wedge s_{n_i}^{(i)} > 0$  then
     $(s_1^{(i)}, \dots, s_{n_i}^{(i)}) := (s_1^{(i)} - 1, \dots, s_{n_i}^{(i)} - 1)$ 
    endwait;

    co Schalten der Hürde  $i$  co

    await true then
     $(q_1^{(i)}, \dots, q_{m_i}^{(i)}) := (q_1^{(i)} + 1, \dots, q_{m_i}^{(i)} + 1)$ 
    endwait;  $h_i^!$ 

```

Durch diese Definition benötigt jedes Schalten einer Hürde eine gewisse Zeitspanne. Damit kann das Schalten der Hürden in ähnlicher Weise zeitlich bezüglich Anfangs- und Endzeiten geordnet sein, wie schon durch die Ablaufordnungen in Abschnitt 1.2 beschrieben wird.

Diese Umsetzung von Petrinetzen in parallele Programme ergibt zwar eine etwas realistischere Darstellung für parallel ablaufende Prozesse, aber die Komplexität der entstehenden Programme ist auch beträchtlich höher. Dies wird deutlich, wenn man die entstehenden Programmsysteme zu transformieren versucht, oder über sie Beweise führen möchte.

5.5 Von applikativen zu prozeduralen, parallelen Programmen

Die bisher beschriebenen Transformationsregeln gestatten die Transformation von applikativen zu applikativen, parallelen Programmen und die Transformation von prozeduralen zu prozeduralen, parallelen Programmen. Die Regeln für Übergänge von applikativen zu prozeduralen Programmen induzieren damit natürlich auch Transformationsregeln für den Übergang von applikativen (parallelen) Programmen zu prozeduralen, parallelen Programmen. Diese Übergänge sollen im folgenden genauer untersucht werden.

5.5.1 Zur parallelen Auswertung von Anweisungen

Wie im Kapitel 2 gezeigt wird, erlauben applikative Formulierungen von Algorithmen häufig eine weitgehend parallele Auswertung. Da der Übergang zu prozeduralen Versionen im allgemeinen eine Sequentialisierung mit sich bringt, werden die Möglichkeiten für eine parallele Auswertung durch den Übergang meistens eingeschränkt. Wir wollen das an zwei Beispielen verdeutlichen.

Beispiel: Prozedurale Fassung der Funktion `op` aus Abschnitt 2.2.3

Unter Ausnützung der Assoziativität der Konkatenation & erhalten wir nach Anwendung der Regel aus /Bauer et al. 78d/, Seite 280, die prozedurale Fassung:

```
funct op = (sequ a, sequ b: length(a) = length(b)) sequ:  
  ( var sequ va, var sequ vb, var sequ vc) := ( a, b, ());  
  while -isempty(a) do ( va, vb, vc) := ( rest( va ), rest( vb ),  
                                          vc & ( top(va) ; top(vb) ))  
  od;  
vc
```

Im Gegensatz zur applikativen Version in 2.2.3 erfolgt die Verknüpfung der einzelnen Komponenten der Sequenzen `a` und `b` durch die Operation `;` nun sequentiell. Die durch die nichtdeterministische Berechnungsregel gegebene Möglichkeit der parallelen, komponentenweisen Verknüpfung besteht für die prozedurale Version nicht mehr.

Beispiel: Prozedurale Fassung der Funktion f aus Abschnitt 2.3.3

Da f von repetitiver Form ist, ist ein Übergang zu einer prozeduralen Version unmittelbar möglich:

```
funct f = (m x, n y)r:  
  (var m vx, var n vy) := (x, y);  
  while B(vx) do ( vx, vy) := ( h( vx ), g( vy, vx ) ) od;  
  T(vy)
```

In der prozeduralen Version wird die Berechnung von h(vx) und g(vy,vx) zwar noch parallel ausgeführt, sind aber die Auswertungszeiten der beiden Ausdrücke stark unterschiedlich, so daß einmal der eine und dann wieder der andere Ausdruck eine erheblich längere Ausführungszeit benötigen, dann entstehen im Vergleich zur Auswertung in 2.3.3 unter Umständen erhebliche, unnötige Wartezeiten.

Beide Beispiele zeigen deutlich die charakteristischen Unterschiede in den Möglichkeiten für eine parallele Auswertung zwischen nahe verwandten, applikativen und prozeduralen Versionen eines Programms.

Bei applikativen Programmen bewirkt die Rekursion einen kellerartigen Aufbau der einzelnen Inkarnationen. Dadurch existieren die Argumentwerte der den Inkarnationen entsprechenden Aufrufe simultan (im Speicher oder in der Berechnung) und können so für eine parallele Berechnung benützt werden. In prozeduralen Versionen "kollabieren" die Argumentekeller zu Programmvariablen, die die Argumentwerte im Laufe der Rechnung nacheinander annehmen. Damit ist die Möglichkeit paralleler Auswertung zugunsten einer Optimierung des Speicherplatzbedarfs eingeschränkt worden.

In Begriffen der Berechnungsregeln ausgedrückt heißt das:

Der Übergang von repetitiver Rekursion ("Tail-Recursion") zu prozeduralen Formulierungen erfordert eine Einschränkung auf "Innermost"-Berechnungsregeln. Durch die Anwendung der nichtdeterministischen Berechnungsregel u.U. vorhandene Möglichkeiten paralleler Auswertung werden beschnitten.

5.5.2 Zur Erhaltung der Parallelität bei Übergängen

Will man die Möglichkeiten paralleler Auswertung applikativer Programme beim Übergang auf prozedurale Programme erhalten, so muß die "implizite" Möglichkeit paralleler Auswertung explizit gemacht werden. Wir wollen dies am zweiten Beispiel des vorangegangenen Abschnitts demonstrieren. Die parallele Auswertung der Funktion f aus Abschnitt 2.3.3 entspricht dem parallelen Programm pf:

```

funct pf = (m x, n y)r:
  [ proc pro =: if B(vx) then vx := h(vx);
                                     send vx on c;  pro  fi;

    proc con =: if B(vz) then vy := g(vy,vz);
                                     receive vz from c;  con  fi;

    (var m vx, var n vy, var m vz) := ( x, y, x);
    stream m c;

    // pro // con //;

  T(vy)
  ]

```

Die Äquivalenz des Programms pf und des Programms f beweisen wir durch Transformationen. Dazu geben wir erst eine nichtdeterministische Einbettung von f an:

```

funct ndf = (m x, n y, m z, queue q)r:
  if B(x)      then  ndf( h(x), y, z, send( q, h(x)) )
  [ -isempty(q) then  ndf( x, g( y, z ), next(q), remove(q) )
  [ ¬B(z)∧¬B(x) then  T(y)
  ]
  ]

```

Es gilt:

$$f(x, y) = ndf(x, y, x, \text{empty})$$

Die Korrektheit der Gleichung läßt sich durch Induktion beweisen. Wir beginnen unsere Transformationen mit dem Programm:

```
// pro // con //
```

Die Regeln in Abschnitt 4.3 und 5.1 liefern:

```

if B(vx) then vx := h(vx);
if B(vz) then vy := g(vy,vz);
    if true      then c := send(c,vx);
        // pro // receive vz from c; con //
    [] ¬isempty(c) then (c, vz) := (remove(c), next(c));
        // send vx on c; pro // con // fi
        else // send vx on c; pro // fi
    else
if B(vz) then vy := g(vy,vz);
    if ¬isempty(c) then (c, vz) := (remove(c), next(c)); // con //
        else abort fi
    else nop fi fi

```

Eine Umordnung der Wächter und Vertauschen der Anweisungen ergibt:

```

if B(vx)      then (vx,c) := (h(vx),send(c,h(vx))); // pro // con //
[] ¬isempty(c) then (vy,vz,c) := (g(vy,vz),next(c),remove(c));
        // pro // con //
[] ¬B(vx) ∧ ¬B(vz) then nop
        else abort fi

```

Das entspricht exakt der repetitiven Fassung der Funktion ndf. Damit ist die Äquivalenz von f und pf gezeigt.

Ist der Beweis der Äquivalenz des parallelen, prozeduralen Programms mit der applikativen Version aus Abschnitt 2.3.3 etwas mühsam, so haben wir damit eine allgemeine Transformationsregel für den Übergang von dem applikativen Programmschema zu parallelen, prozeduralen Programmen. Das applikative Programmschema ist häufig anwendbar, da sowohl die Entrekursivierung, die die Assoziativität der primitiven Operationen ausnützt, als auch die Entrekursivierung, die die Möglichkeit der Funktionsumkehr verwendet (für beide Fälle vgl. /Bauer et al. 78d/, Seite 280), auf exakt ein solches Programmschema führen. In der gleichen Weise wie eben können natürlich noch weitere Transformationsregeln für den Übergang von applikativen zu parallelen, prozeduralen Programmen entwickelt werden.

Teil III

Beispiele, abschließende Bemerkungen

Im Gegensatz zu den Programmbeispielen in den vorangegangenen Abschnitten, die höchstens als Fragmente aus Programmentwicklungen zu bezeichnen sind und jeweils zur Verdeutlichung bestimmter Aspekte dienen, soll nun an zwei einfachen Beispielen die Anwendung der eingeführten, formalen Methoden in ausführlicheren Programmentwicklungen demonstriert werden. Wie das Programm am Ende von Abschnitt 2.2.3 zeigt, liefert die parallele Anwendung der nichtdeterministischen Berechnungsregel schon eine befriedigende, parallele Auswertung für gewisse rekursive Funktionen. Treten jedoch bei nichtlinearer Rekursion während der Abarbeitung der rekursiven Aufrufe gleiche Teilausdrücke auf, so wird die nichtdeterministische Berechnungsregel ineffizient. Man kann den überflüssigen Aufwand durch Sequentialisierung vermeiden.

Soll aber die Auswertung parallel vorgenommen werden, um beispielsweise minimale Rechenzeiten zu erhalten, so wird eine Kommunikation (vgl. dazu Abschnitt 2.1.2) zwischen den parallel ablaufenden Auswertungen nötig. Wertverlaufsrekursion führt dabei auf Grund der auf dem Definitionsbereich gegebenen linearen Ordnung auf gerichtete Kommunikationsbeziehungen. Dies wird am Beispiel des Interpolationsalgorithmus nach Aitken-Neville demonstriert, der auf eine Form der Kommunikation über Kommunikationstabellen gebracht wird.

Bei komplizierten rekursiven Definitionen ist wechselseitige Kommunikation erforderlich, wie zum Beispiel beim parallelen Durchlaufen von Graphen zur Ermittlung der von einem Punkt aus erreichbaren Menge von Knoten. Der Graphdurchlaufalgorithmus wird zu einer Version mit bewachten, kritischen Bereichen entwickelt.

Nach den Beispielen folgt eine knappe zusammenhängende Übersicht über die in der Literatur beschriebenen Ansätze zur Theorie paralleler Programmierung. Entsprechend den Schwerpunkten der vorliegenden Arbeit wird dabei formalen Methoden besondere Aufmerksamkeit gewidmet. Dies umfaßt - neben den nur spärlichen Ansätzen zur abgesicherten Entwicklung paralleler Programme - Verfahren zur formalen Beschreibung der Semantik von parallelen Sprach-elementen und Verifikationsmethoden für parallele Programme.

Die Arbeit schließt mit Ausblicken auf offene Fragestellungen und mögliche zukünftige Untersuchungen.

Gilt aber $af(h_1(x)) \cap af(h_2(x)) \neq \emptyset$, sind sogar relativ viele Elemente im Durchschnitt beider Mengen, so empfiehlt es sich, die Werte $f(h_1(x))$ und $f(h_2(x))$ nicht unabhängig von einander zu berechnen.¹⁾Eine Abhilfe verspricht die Verwendung von Wertetabellen. Besonders bei Wertverlaufsrekursion, wie z.B. wenn $h_1(h_1(x)) = h_2(x)$ ist, wie bei der Fibonacci-Funktion, genügt häufig schon ein Tabellenausschnitt fester Länge. Etwas komplizierter wird die Situation, wenn beispielweise $h_1(h_2(x)) = h_2(h_1(x))$ gilt. Aber auch in diesen Fällen kann eine Tabellierung erfolgen und eine beträchtliche Effizienzverbesserung erreichen. Als Beispiel betrachten wir das Schema nach Aitken/Neville.

6.1.2 Das Schema nach Aitken/Neville

Eine kurze Programmentwicklung in /Broy 79/ führt auf ein Programm der Form:

```
funct nev = (rat x, index i, index j)rat:  
  if j = 0 then y[i]  
    else nev(x,i+1,j-1)+dd(x,i,i+j)*  
      (nev(x,i+1,j-1)-nev(x,i,j-1)) fi
```

dabei seien $y[0], \dots, y[n]$ die Stützwerte an den Stützstellen $x[0], \dots, x[n]$, die Art index steht für nat[0..n] und die Funktion dd bezeichnet die "dividierten Differenzen"
 $dd(x,i,k) = (x-x[k])/(x[k]-x[i])$.

6.1.3 Parallelisierung

In /Broy 79/ wird aus dem obigen Programm durch Tabellierung schließlich ein sequentielles Programm entwickelt, das auf einem eindimensionalen Feld als Tabelle der Länge n+1 arbeitet. Wir gehen jedoch zu einem parallelen Programm über, das die einzelnen "Schichten" parallel berechnet.

¹⁾ vgl. Abschnitt 2.1.2 und auch /Bauer, Wössner 79/

Wir wählen die in Abschnitt 3.2.4 beschriebene Vorgehensweise und entwickeln zuerst eine Version, die mit Tabellen der Art table arbeitet, wobei table die durch den abstrakten Typ TABLE(rat) in Abschnitt 3.2.1 spezifizierte Art sei. Wir gehen damit zum Tabellieren von nev über:

```
funct tab = (index j)table: enter( vac, 0, j ),  
funct enter = (table t, index i, index j)table:  
  if i+j  $\leq$  n then enter( put( t, nev( x, i, j ) ), i+1, j )  
  else t fi
```

Hier sind x und n unterdrückte Parameter. Für alle i, j mit i+j = n gilt:

```
get( tab(j), i+1 ) = nev( x, i, j ).
```

Nun können wir zu einer Berechnung mit Hilfe der tabellierten Werte übergehen:

```
funct tab = (index j)table:  
  if j = 0 then enter( vac, 0, 0 )  
  else enter'( tab(j-1), vac, 0, j ) fi,  
funct enter' = (table in, table out, index i, index j)table:  
  if i+j  $\leq$  n then enter'( in, put( out, get( in, i+2 ) + dd( x, i, i+j ) +  
    ( get( in, i+2 ) - get( in, i+1 ) ),  
    i+1, j )  
  else out fi
```

Es gilt:

```
get( tab( n ), 1 ) = nev( x, 0, n )
```

Wenn wir nur an diesem Wert interessiert sind, wie das bei der Interpolation im allgemeinen der Fall ist, ergibt mit

```
funct etab = (table in, nat j: j = tab(j-1) ) table:  
  if j  $\leq$  n then etab( enter'( in, vac, 0, j ), j+1 )  
  else in fi
```

der Aufruf get(etab(enter(vac, 0, 0), 0), 1) das gewünschte Resultat nev(x, 0, n).

Die Funktion `etab` entsteht durch einfache "Funktionsumkehr" aus der Funktion `tab` (vgl. /Bauer, Wössner 79/).

Beide Funktionen `etab` und `enter` haben genau die in Abschnitt 3.2.4 beschriebene Form. Die Anwendung der dortigen Regel liefert:

```
funct interpol = (rat x)rat:  
┌  
  get(out,1)  
  
  where  
  
    comtable rat in, comtable rat out,  
  
    comfunct cotab = (comtable rat in, comtable rat out, nat j):  
      if j ≤ n then coarray rat help,  
                      // coen(in,help,0,j) // cotab(help,out,j+1)//  
  
      fi,  
  
    comfunct coen = (comtable rat in, comtable rat out, index i, index j):  
      if i+j ≤ n then put get(in,i+2)+dd(x,i,i+j)*  
                        (get(in,i+2)-get(in,i+1)) on out;  
                        coen(in,out,i+1,j)  
  
      fi,  
  
      // for i from 0 to n do put y[i] on in od //  
      cotab(in,out,1) //
```

Dabei ist der Aufruf `enter`(`init`,0,0) durch eine for-Wiederholung ausgedrückt worden, weil die Rekursion genau der definierenden Transformation der for-Wiederholung in /Bauer et al. 77/ entspricht.

Ein ähnliches paralleles Programm läßt sich für Kommunikationsströme entwickeln. Dabei kann man ganz analog zuerst eine Version herleiten, die mit Objekten der Art queue arbeitet. Dann geht man zu einer prozeduralen Version und schließlich zu einer Formulierung mit Strömen über (vgl. Abschnitt 5.5).

6.2 Wechselseitige Kommunikation

Wie im vorangegangenen Abschnitt demonstriert wurde, führt die Parallelisierung von einer durch Wertverlaufsrekursion definierten Funktion auf gerichtete Kommunikation. Dabei wird die auf dem Definitionsbereich gegebene lineare Ordnung ausgenutzt. Nun betrachten wir ein Beispiel, für das eine lineare Ordnung auf dem Definitionsbereich nicht gegeben ist. Wir werden ein paralleles Programm für das Durchlaufen eines gerichteten Graphen entwickeln, das die Menge der von einem gegebenen Knoten aus erreichbaren Knoten berechnet.

6.2.1 Durchlaufen von Graphen

In /Broy et al. 78a/ wird eine Spezifikation für das Problem, in einem gegebenen Graphen für einen Knoten die Menge der erreichbaren Knoten zu berechnen, angegeben. Eine rekursive Lösung für das Problem stellt die folgende Funktion dar:

```
funcnt emb = (node x, set node s)set node
  if x ∈ s then s
  else  $\bigcup_{y \in \text{neighbours}(x)}$  emb(y, s ∪ {x}) fi
```

Dabei bezeichne node die Art der endlichen Menge der Knoten des gegebenen Graphen und neighbours(x) gebe für jeden Knoten x die Menge seiner benachbarten Knoten wider. Für jeden Knoten z liefert der Aufruf emb(z, ∅) die Menge der von z aus erreichbaren Knoten des Graphen.

Wie in /Broy et al. 78a/ demonstriert wird, werden Programme der obigen Form häufig auf die Gestalt verneisteter Rekursion gebracht, d.h. es wird eine Sequentialisierung vorgenommen, so daß das Durchsuchen des Graphen in einer vorgegebenen partiellen Ordnung erfolgt. So läßt sich durch entsprechende Sequentialisierung sowohl "Depth-First-Search" als auch "Breadth-First-Search" aus ein und demselben rekursiven Programm herleiten. Wir wollen jedoch nun ein Programm für das parallele Durchsuchen von Graphen entwickeln.

Der Einfachheit halber beschränken wir uns auf Graphen mit genau zwei ausgehenden Kanten pro Knoten. Für einen Knoten x bezeichnen $l(x)$ und $r(x)$ die beiden über diese Kanten erreichbaren Nachbarn von x . Damit erhält unser Programm die Form:

```
funct emb = (node x, set node s)set node:  
  if x ∈ s then s  
    else emb(l(x), s ∪ {x}) ∪ emb(r(x), s ∪ {x}) fi
```

Außer für Bäume ist diese Funktion hochgradig ineffizient, da jeder mehrfach von x aus erreichbare Knoten unnötigerweise tatsächlich mehrfach aufgesucht wird. Diese Ineffizienz kann durch eine Sequentialisierung der Funktionsaufrufe vermieden werden. Dazu verwenden wir die folgende, durch Induktion beweisbare Beziehung (vgl. /Broy et al. 78a/):

$$(i) \quad emb(x,s) \cup emb(y,s) = emb(x, emb(y,s)).$$

Die Kommutativität der Mengenvereinigung ergibt sofort auch:

$$(ii) \quad emb(x, emb(y,s)) = emb(y, emb(x,s)).$$

Die Gleichung (i) ergibt durch Anwenden auf emb :

```
funct emb = (node x, set node s)set node:  
  if x ∈ s then s  
    else emb(l(x), emb(r(x), s ∪ {x})) fi
```

Diese Version entspricht der "Depth-First-Search"-Strategie. Zwar werden nun keine Pfade mehrfach durchlaufen, aber die Sequentialisierung zerstört gleichzeitig die Möglichkeit der parallelen Auswertung beider Funktionsaufrufe im Rumpf von emb . Deshalb wollen wir nun auf die prozedurale Ebene übergehen und dort eine parallele Version des Algorithmus entwickeln. Den Übergang zur prozeduralen Ebene vollziehen wir mit der Prozedur `join`:

```
funct emb = (node x, set node s')set node:  
  proc join = (node x): s := emb(x,s),  
    var set node s := s'; join(x,s); s
```

Ein UNFOLD für emb und ein Aufbrechen der Aufrufe liefert für die Prozedur `join`:

```

proc join = (node x):
  if x ∈ s then s := s
    else s := s ∪ {x}; s := emb(l(x), s); s := emb(r(x), s) fi

```

Das Ersetzen von $s := s$ durch nop und ein FOLD der beiden letzten Zuweisungen ergeben:

```

proc join = (node x):
  if x ∈ s then nop
    else s := s ∪ {x}; join(l(x)); join(r(x)) fi

```

Im folgenden Abschnitt gehen wir zu einer parallelen Version von join über.

6.2.2 Paralleles Durchlaufen von Graphen

Die letzte Version von join legt folgende parallele Version pjoin von join nahe:

```

proc pjoin = (node x): // go(x) //
  where
  proc go = (node x):
    await x ∈ s then nop
      ∇ x ∉ s then s := s ∪ {x} ▷ ( go(l(x)) // go(r(x)) ) endwait

```

Die Korrektheit dieses Übergangs ergibt sich aus dem folgenden Lemma.

Lemma: Für beliebige Knoten x_1, \dots, x_n des Graphen gilt die Transformationsregel:

$$\begin{array}{c}
 // \text{go}(x_1) // \dots // \text{go}(x_n) // \\
 \hline
 \begin{array}{c}
 \updownarrow \\
 \text{join}(x_1); \dots ; \text{join}(x_n)
 \end{array}
 \end{array}$$

Beweis: Durch Induktion über a , die Anzahl der Knoten x mit $x \notin s$.

Sei $a = 0$, dann sind beide Anweisungen äquivalent zu nop. Sei $a > 0$ und die Behauptung richtig für $a-1$. Wir führen den Induktionsschluß durch Induktion über n .

Nun ist ein FOLD für join möglich. Die Vertauschbarkeit von Aufrufen von join liefert die gewünschte Form. Damit ist der Beweis der Korrektheit der Regel abgeschlossen. \square

Aus der parallelen Version von join lassen sich weitere Versionen ableiten.

6.2.3 Paralleles Durchlaufen mit fest vorgegebener Prozessoranzahl

Haben wir nur eine fest vorgegebene Anzahl von Prozessoren zur Verfügung, so können auch Programme für das Durchlaufen von Graphen entwickelt werden, die auf einer fest vorgegebenen Anzahl von Prozessoren ablaufen. Sind beispielsweise nur zwei Prozessoren vorhanden, so kann join(x) durch

```
if x  $\in$  s then nop  
      else s := {x} u s; // do(l(x)) // do(r(x)) // fi
```

ersetzt werden, wobei

```
proc do = ( node x ):  
  await x  $\in$  s then nop  
   $\nabla$  x  $\notin$  s then s := {x} u s  $\triangleright$  do(l(x)); do(r(x)) fi
```

Die Korrektheit dieses Übergangs läßt sich wieder durch Induktion beweisen, indem man zeigt, daß die obige Version ein "Abkömmling" der parallelen Version in Abschnitt 6.2.2 ist.

7 Historische Entwicklungen und abschließende Bemerkungen

Ergänzend zu den Literaturhinweisen in den vorangegangenen Abschnitten soll nun ein knapper Überblick über die in der Literatur zu findenden Ansätze zu einer Theorie der parallelen Programmierung gegeben werden. Abschließend werden offene Fragestellungen und Ziele möglicher, zukünftiger Untersuchungen kurz angesprochen werden.

7.1 Historischer Überblick

Neben einem kurzen Überblick über die Entwicklungen auf dem Gebiet der Sprachelemente für die parallele Programmierung sollen insbesondere die Ansätze zur formalen Beschreibung der Semantik und zur Verifikation paralleler Programme kurz erwähnt werden. Die aufgeführten Literaturzitate können keinerlei Anspruch auf Vollständigkeit erheben.

7.1.1 Sprachelemente für die parallele Programmierung

Ursprünglich waren Sprachelemente für die parallele Programmierung ausschließlich durch die in den Maschinen, der "Hardware", vorgegebenen Strukturen geprägt. So wurden die ersten Kommunikationsmechanismen zur Koordination paralleler Prozesse über den Begriff des gemeinsamen Speichers ("Common Store") eingeführt (vgl. /Dijkstra 65/). Später wurde dieser Ansatz zu einem für die Koordination spezifischen Sprachelement, den Semaphoren (vgl. /Dijkstra 68/), weiterentwickelt.

Dieses Sprachelement wurde in /Hoare 71/ zu "Conditional Critical Regions" verallgemeinert. Neben diesem Konzept zur dezentralisierten Koordination und Synchronisation, entstand das Monitorkonzept (vgl. /Hoare 74/) für die zentralisierte Koordination.

Überhaupt stehen Fragen der Synchronisation schon deshalb¹⁾ im Mittelpunkt der Interessen (vgl. /Brinch Hansen, Staunstrup 78/), weil der synchronisierte Zugriff auf gemeinsame Speicherabschnitte oder Programmvariable als das wesentliche Kommunikationsmittel erscheint.

¹⁾ neben Problemen der Betriebsmittelvergabe

So nehmen Probleme des koordinierten Lesen und Schreibens auf gemeinsamen Speicherabschnitten einen breiten Raum in der einschlägigen Literatur ein (vgl. /Courtois et al. 71/).

Um eine Reduzierung des Synchronisationsaufwands bemühen sich /Sintzoff, van Lamsweerde 75/ und /Sintzoff 79/, indem sie jede gemeinsame Variable genau einem Prozeß zuordnen, der dann diese Programmvariable ändern darf, während die übrigen Prozesse nur lesend auf diese Variable zugreifen dürfen.

In /Lampport 77a/ wird völlig auf die Koordination von Lese- und Schreibzugriffen verzichtet. Stattdessen wird durch entsprechende Abfragen die Konsistenz der gelesenen Werte getestet und gegebenenfalls wird die Leseaktion wiederholt.

Andere Ansätze befassen sich mit Transitionssystemen, bei denen in jedem Zustand eine festgelegte Menge von Übergängen möglich ist, aus denen dann einer oder mehrere konfliktfreie in nichtdeterministischer Weise ausgewählt werden (vgl. /Keller 75/, /Mayr 78/). Solche Systeme können in gewisser Weise als Verallgemeinerungen von Petrinetzen (vgl. /Petri 62/, /Peterson 77/) aufgefaßt werden. Eine andere Erweiterung des Ansatzes der Petrinetzen ersetzt die Belegungen ("Steine", "Token") durch Werte oder Objekte, also durch Nachrichten, die durch die Transitionen übertragen werden (vgl. dazu /Yoeli 79/).

Solche Systeme von kommunizierenden Prozessen, die über keinerlei gemeinsame Variable verfügen, werden auch in /Hoare 78/ und /Brinch Hansen 78/ beschrieben. Allerdings sind die einzelnen Prozesse dabei rein prozedural.

Die mit dem Konzept der Programmvariablen verbundenen Schwierigkeiten führten auch zu Versuchen einen applikativen Stil für die parallele Programmierung zu entwickeln. Die "Data Flow Programming Languages" (vgl. /Kosinski 73/) können in gewisser Weise als ein Schritt in diese Richtung angesehen werden. In /Kessel 77/ werden gerade die "Kommunikationsmodule" über lokale Variable erklärt. Ein weit entwickelter Ansatz findet sich in /Friedman, Wise 76a, 78a, 78b/.

7.1.2 Formale Beschreibungen der Semantik

Bereitet schon die formale Beschreibung sequentieller, imperativer Programmiersprachen große Schwierigkeiten, so daß es beispielweise ein Jahrzehnt gedauert hat, bis Sprachen wie ALGOL 60 einigermaßen vollständig formal beschrieben waren, so vervielfachen sich die Probleme bei den Abläufen paralleler Programme. Nichtdeterminismus, Synchronisationsprobleme und die präzise Definition paralleler Ausführung bereiten bei Ansätzen mit denotationeller Semantik große Schwierigkeiten, nicht zuletzt, weil die parallele Ausführung von Programmen in gewisser Weise eher durch eine operative, als durch eine mathematische Semantik beschrieben werden kann.

Die unterschiedlichsten Sprachelemente, die von den verschiedensten Seiten für die parallele Programmierung vorgeschlagen wurden, wurden in den seltensten Fällen von halbwegs formalen Semantikbeschreibungen begleitet, obwohl gerade solche Sprachelemente leicht falsch verstanden werden können. Für neu entwickelte Sprachen wie z.B. die ADA-Sprache GREEN, die vom amerikanischen Verteidigungsministerium in Auftrag gegeben wurde, konnten gerade die Elemente für parallele Programme nicht formal beschrieben werden. So heißt es in der Einleitung zur formalen Beschreibung von GREEN (vgl. /GREEN 79/, Seite 1-2):

"However, the State of the Art in formal semantics does not allow (as of Spring 1979), in the opinion of the authors, to give a mathematically meaningful semantics for parallelism ... "

Ein Ansatz zur formalen Beschreibung der in /Hoare 78/ vorgeschlagenen Sprachelemente findet sich in /Francez et al. 78/. Ebenso existieren Versuche (vgl. /Raulefs 79/) die Semantik von Actors (vgl. /Hewitt, Baker 78/) formal zu beschreiben.

Ein weit fortgeschrittener Ansatz wird in /Lauer, Campbell 75/ beschrieben. Hier wird, ausgehend von "Pathexpressions" (vgl. /Campbell, Habermann 74/), mit Hilfe von Petrinetzen die Bedeutung solcher "Pathexpressions" formalisiert. Eine Variante dazu enthält /Robert, Verjus 77/, wobei dort mit jeder Prozedur gewisse "Statuszähler" verknüpft werden, mit deren Hilfe dann Synchronisationsprädikate angegeben werden können. Beide Ansätze sind jedoch stark auf das Synchronisationsproblem ausgerichtet.

Anstrengungen, die theoretischen Grundlagen paralleler Programme zu durchleuchten, stützen sich im allgemeinen auf Arbeiten über Nichtdeterminismus. Zur Ausdehnung der Methoden denotationeller Semantik ist es insbesondere notwendig ein mathematisches Modell zur Darstellung der Prozesse als Objekte anzugeben. Ein auf ein solches Modell ausge richteter Ansatz enthalten die Arbeiten von /Milner 73/ und /Milne, Milner 77/. Versuche, die Methoden der denotationellen Semantik auf parallele Sprachelemente zu erweitern, finden sich in /Keller 78/ und /Plotkin 79/.

7.1.3 Verifikationsmethoden

Ein anderer Bereich, dem erstaunlicherweise mehr Aufmerksamkeit zuteil wurde als Methoden zur formal abgesicherten Konstruktion paralleler Programme, handelt von Verifikationsmethoden. Früh wurde die Notwendigkeit solcher Methoden auf Grund der "Nichtreproduzierbarkeit" von Fehlern beim Testen als besonders dringlich bezeichnet. Ein erster Vorschlag in dieser Richtung findet sich in /Ashcroft, Manna 71/, wo Programme durch Flußdiagramme beschrieben werden. In /Levitt 72/ werden für parallele Programme mit Semaphoren und P- und V-Operationen Beweismethoden beschrieben. In /Clint 73/ werden Verifikationsmethoden für Coroutinen angegeben. Die Reduktionsmethode in /Lipton 75/ versucht immer größere Teile des Programms als unteilbare Aktionen zu betrachten, und somit die kombinatorische Komplexität der zeitlichen Verschränkung für Beweise zu reduzieren.

Maschinenorientiert ausgedrückt besitzen parallele Programme nicht nur einen Kontrollzustand (oder Befehlszähler) sondern einen Vektor von Kontrollzuständen. Eine Zusicherung im Sinn von Floyd oder Hoare bezieht sich in einem sequentiellen Programm immer auf den "Kontrollzustand", der der Aufschreibung der Zusicherung im Programm entspricht. Diese einfache Beziehung ist bei parallelen Programmen nicht möglich. Abhilfe ergibt eine pauschale Einführung eines Vektors von Kontrollzählern, über die die Zustände der übrigen parallelen Prozesse in der Belegungen der Programmvariable sichtbar und damit durch Zusicherungen ansprechbar werden.

In /Owicki 75/, /Howard 76/ und /Gries 76/ werden statt der Einführung eines Vektors von Kontrollzählern Hilfsvariable eingeführt, die jeweils dem Programmsystem so angepaßt werden können, daß die Informationen über die Kontrollzustände der übrigen Prozesse, die für die Zusicherungen von Bedeutung sind, durch die Werte der Hilfsvariablen erkennbar sind.

Auch durch graphentheoretische Methoden beschriebene Programme (Petrietze) lassen sich mit Hilfe von Zusicherungen beweisen ("Placeassertions" in /Keller 76/). Eine stark auf Invarianten abgestützte Methode enthält /Ashcroft 75/.

In /Flon, Suzuki 78, 78a/ wird die Methode der schwächsten Vorbedingungen nach /Dijkstra 76/ auf gewisse parallele Sprachelemente übertragen. Eine ausführliche Darstellung und Diskussion von Verifikationsmethoden für parallele Programme enthält /Lampport 77b/.

7.2 Offene Probleme, mögliche, zukünftige Untersuchungen

Die vorliegende Arbeit versteht sich als der Versuch, die Konzepte und Begriffe der parallelen Programmierung im Rahmen einer im Prinzip sequentiellen Programmiersprache systematisch zu entwickeln, formal zu beschreiben und dabei Methoden für die abgesicherte Entwicklung paralleler Programme aufzuzeigen.

Inwieweit die in dieser Arbeit vorgestellten Ansätze den Erfordernissen der Praxis gerecht werden können, kann sicherlich erst nach einer größeren Zahl von Anwendungen entschieden werden. Von solchen Anwendungen ist auch eine Weiterentwicklung der Konzepte durch die zu erwartende Rückkoppelung zu erhoffen. Dabei stellt insbesondere die Behandlung von Programmsystemen mit einer hohen Zahl von parallel ablaufenden Komponenten eine Herausforderung an die Anwendbarkeit der Methoden dar. Gerade hier kann die Unterstützung durch ein Transformationssystem, wie es im Rahmen des Projekts CIP an der Technischen Universität München entwickelt wird, sich als außerordentlich hilfreich erweisen.

Ein weiterer Punkt, der in dieser Arbeit nicht behandelt werden konnte, betrifft die simultane Benutzbarkeit großer Datenstrukturen (vgl. /Bayer, Schkolnick 77/ und /Ramsperger 77/). Die Ausdehnung der in dieser Arbeit vorgeschlagenen Methoden auf solche Datenstrukturen stellt ebenfalls ein offenes Problem dar.

Als Ausgangspunkt für abgesicherte Programmentwicklungen sind formale Spezifikationen von zentralem Interesse (vgl. /Broy et al. 78a/). Bei Programmen, die eine bestimmte (mathematische) Funktion berechnen, kann von herkömmlichen Spezifikationen über sequentielle Programmversionen zu parallelen Programmen übergegangen werden (vgl. dazu Kapitel 6). Auch für "interaktive" Systeme ist ein Übergang von einer sequentiellen Version zu Systemen paralleler Programme oft möglich (vgl. Abschnitt 5.4.2). Die Spezifikation großer, interaktiver Programmsysteme stellt dabei jedoch noch ein Problem dar.

Literatur

/Ashcroft, Manna 71/

E.A. Ashcroft, Z. Manna: Formalization of Properties of Parallel Programs. Machine Intelligence 6, 1971, 17-41

/Ashcroft 75/

E.A. Ashcroft: Proving Assertions about Parallel Programs. Journal of Computer and Systems Sciences 10, 1975, 110-135

/Arvind et al. 77/

Arvind, K.P. Gostelow, W. Plouffe: Indeterminacy, Monitors and Dataflow. Proceedings of the Sixth ACM Symposium on Operating Systems Principles, November 1977, 159-169

/Arvind, Gostelow 78/

Arvind, K.P. Gostelow: Some Relationships between Asynchronous Interpreters of a DataFlow Language. In: /Neuhold 78/, 95-119

/Arnold 77/

A. Arnold: Schemes de programmes recursifs non deterministes avec appel synchrone. In: B. Robinet (ed.): Proceedings of the 3rd International Symposium on Programming, March, 28-30 1978, Dunod 1978, 28-30

/Back 79/

R.-J. Back: Semantics of Unbounded Nondeterminism. University of Helsinki, Computing Centre, Research Report No 8, 1979

/Bauer 77/

F.L. Bauer: Algorithmische Sprachen. Technische Universität München, Institut für Informatik, Skriptum zur Vorlesung (2 Teile) 1977

/Bauer 78/

F.L. Bauer: Lazy Evaluation and Pointer Representation. In: /MOD 78/, 406-420

/Bauer et al. 77/

F.L. Bauer, M. Broy, R. Gnatz, W. Hesse, B. Krieg-Brückner: Notes on the Project CIP: Towards a Wide Spectrum Language for Program Development by Transformations. Technische Universität München, Institut für Informatik, TUM-INFO-7716, July 1977

/Bauer et al. 78a/

F.L. Bauer, M. Broy, R. Gnatz, W. Hesse, B. Krieg-Brückner: Towards a Wide Spectrum Language for Program Specification and Program Development. In: K. Alber (Hrsg.): Programmiersprachen, 5. Fachtagung der GI, Braunschweig 1978, Informatik Fachberichte 12, Berlin-Heidelberg-New York: Springer 1978, 73-85

/Bauer et al. 78b/

F.L. Bauer, M. Broy, R. Gnatz, W. Hesse, B. Krieg-Brückner: A Wide Spectrum Language for Program Development. In: B. Robinet (ed.): Proceedings of the 3rd International Symposium on Programming, March 28-30, 1978, Paris: Dunod 1978, 1-16

/Bauer et al. 78c/

F.L. Bauer, M. Broy, R. Gnatz, W. Hesse, B. Krieg-Brückner, H. Partsch, P. Pepper, H. Wössner: Towards a Wide Spectrum Language To Support Program Specification and Program Development. SIGPLAN Notices 13:12, December 1978, 15-24

/Bauer et al. 78d/

F.L. Bauer, M. Broy, H. Partsch, P. Pepper, H. Wössner: Systematics of Transformation Rules. In: /MOD 78/, 273-289

/Bauer, Wössner 79/

F.L. Bauer, H. Wössner: Algorithmische Sprache und Programmentwicklung. Berlin-Heidelberg-New York: Springer 1980, im Erscheinen

/Bayer, Schkolnick 77/

R. Bayer, M. Schkolnick: Concurrency of Operations on B-trees. Acta Informatica 9, 1977, 11-26

/Bernstein 66/

A.J. Bernstein: Analysis of Programs for Parallel Processing. IEEE Transactions on Electronic Computers, EC-15:5, October 1966, 757-763

/Brinch Hansen 78/

P. Brinch Hansen: Distributed Processes: A Concurrent Programming Concept. Comm. ACM 21:11, November 1978, 934-941

/Brinch Hansen, Staunstrup 78/

P. Brinch Hansen, J. Staunstrup: Specification and Implementation of Mutual Exclusion. IEEE Transaction on Software Engineering, SE-4:5, September 1978, 365-370

/Broy 78/

M. Broy: A Case Study in Program Development: Sorting. Tech. Universität München, Institut für Informatik, TUM-INFO-7831, Dezember 1978

/Broy 79/

M. Broy: Transformations for Reliable Software. Proc. Conference on the Production and Assessment of Numerical Software, Liverpool, April 9-11, 1979 (to appear)

/Broy, Wirsing 79a/

M. Broy, M. Wirsing: From Enumeration to Backtracking. (Unpublished manuscript)

/Broy, Wirsing 79b/

M. Broy, M. Wirsing: Programming Languages as Abstract Data Types. Les Arbres en Algèbre et en Programmation, Lille, 21-23 Feb. 1980

/Broy et al. 78a/

M. Broy, R. Gnatz, M. Wirsing: Problemspezifikation - eine Grundlage für Programmentwicklung. Proceedings of the Workshop on Reliable Software, Bonn, September 22-23, 1978, Hanser:München 1979, 235-246

/Broy et al. 78b/

M. Broy, R. Gnatz, M. Wirsing: Semantics of Nondeterministic and Noncontinuous Constructs. In: /MOD 78/, 553-592

/Broy et al. 79/

M. Broy, W. Dosch, H. Partsch, P. Pepper, M. Wirsing: Existential Quantifiers in Abstract Data Types. H.A. Maurer (ed.): Proceedings of the Sixth Colloquium on Automata, Languages and Programming, Graz 1979, LNCS 71, Berlin-Heidelberg-New York: Springer 1979, 73-87

/Campbell, Habermann 74/

R.H. Campbell, N. Habermann: The Specification of Process Synchronisation by Path Expressions. Proc. Int. Symp. on Operating Systems Theory and Practice, April 1974, 91-102

/Clint 73/

M. Clint: Program Proving. Acta Informatica 2, 1973, 50-63

/Courtois et al. 71/

P.J. Courtois, F. Heymans, P.L. Parnas: Concurrent Control with Readers and Writers. Comm. ACM 14:10, October 1971, 667-668

/Dahl, Nygaard 66/

O.J. Dahl, K. Nygaard: SIMULA - an ALGOL-Based Simulation Language. Comm. ACM 9:9, September 1966, 671-687

/Dennis 74/

J.B. Dennis: First Version of a Data Flow Procedure Language. In: B. Robinet (ed.): Colloque sur la Programmation, LNCS 19, Berlin-Heidelberg-New York: Springer 1975, 362-376

/Dijkstra 65/

E.W. Dijkstra: Solution of a Problem in Concurrent Programming Control. Comm. ACM 8:9, September 1965, 569

/Dijkstra 68/

E.W. Dijkstra: Co-Operating Sequential Processes. In: F. Genuys (ed.): Programming Languages. Academic Press, 1968, 43-112

/Dijkstra 76/

E.W. Dijkstra: A Discipline of Programming. Prentice Hall, Englewood Cliffs N. J. 1976

/Ershov 78/

A.P. Ershov: On the Essence of Compilation. In: /Neuhold 78/, 391-418

/Feldman 79/

J.A. Feldman: High Level Programming for Distributed Computing. Comm. ACM 22:6, June 1979, 353-368

/Flon, Suzuki 78/

L. Flon, N. Suzuki: Nondeterminism and the Correctness of Parallel Programs. In: /Neuhold 78/, 589-605

/Flon, Suzuki 78a/

L. Flon, N. Suzuki: Consistent and Complete Proof Rules for the Total Correctness of Parallel Programs. Palo Alto Research Center, CSL-78-6, November 1978

/Francez et al. 78/

N. Francez, C.A.R. Hoare, D.J. Lehmann, W.P. de Roever: Semantics of Non-determinism, Concurrency and Communication. Tech. Report, August 1978

/Friedman, Wise 76a/

D.P. Friedman, D.S. Wise: CONS Should Not Evaluate its Arguments. In: S. Michaelson, R. Milne (eds.): Proc. of the Int. Symp. on Automata, Languages and Programming. Edinburgh 1976, 257-284

/Friedman, Wise 76b/

D.P. Friedman, D.S. Wise: The Impact of Applicative Programming on Multiprocessing. Proceedings of the Int. Conf. on Parallel Processing, IEEE 1976, 263-272

/Friedman, Wise 78/

D.P. Friedman, D.S. Wise: A Note on Conditional Expressions. Comm. ACM 21:11, November 1978, 931-933

/Friedman, Wise 78a/

D.P. Friedman, D.S. Wise: Applicative Multiprogramming. Indiana University, Computer Science Department, Technical Report 72, January 1978, revised December 1978

/Friedman, Wise 78b/

D.P. Friedman, D.S. Wise: Aspects of Applicative Programming for Parallel Processing. IEEE Transactions on Computers, C-27:4, April 1978, 289-296

/Green 79/

Formal Definition of the Green Programming Language. Preliminary Draft, Cii-Honeywell Bull, April 1979

/Greif 77/

I. Greif: A Language for Formal Specification. Comm. ACM 20:12, December 1977, 931-935

/Gries 77/

D. Gries: An Exercise in Proving Parallel Programs Correct. Comm. ACM 20:12, December 1977, 921-930

/Henderson, Morris 76/

P. Henderson, J.H. Morris: A Lazy Evaluator. University of New Castle upon Tyne, Computing Laboratory, 1976

/Hewitt 79/

C. Hewitt: Evolving Parallel Programs. Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Working Paper 164c, January 1979

/Hewitt, Baker 78/

C. Hewitt, H. Baker: Actors and Continuous Functionals. In: /Neuhold 78/, 367-387

/Hoare 71/

C.A.R. Hoare: Towards a Theory of Parallel Programming. In: C.A.R. Hoare, R.H. Perott (eds.): Operating Systems Techniques, Academic Press, New York 1972, 61-71

/Hoare 74/

C.A.R. Hoare: Monitors: An Operating Systems Structuring Concept. Comm. ACM 17:10, October 1974, 549-557

/Hoare 78/

C.A.R. Hoare: Communicating Sequential Processes. Comm. ACM 21:8, August 1978, 666-677

/Hoare 78a/

C.A.R. Hoare: Some Properties of Predicate Transformers. J. ACM 25:3, July 1978, 461-480

/Howard 76/

J.H. Howard: Proving Monitors. Comm. ACM 19:5, May 1976, 273-279

/Jammel, Stiegler 77/

A.J. Jammel, H.G. Stiegler: Managers versus Monitors. In: Proc. of the IFIP Congress 77, Amsterdam: North-Holland 1977, 827-830

/Kahn 74/

G. Kahn: The Semantics of a Simple Language for Parallel Programming. In: Proc. of the IFIP Congress 74, 471-475

/Kahn, MacQueen 77/

G. Kahn, D. MacQueen: Coroutines and Networks of Parallel Processes. In: Proc. of the IFIP Congress 77, Amsterdam: North-Holland 1977, 994-998

/Keller 75/

R.M. Keller: A Fundamental Theorem of Asynchronous Parallel Computation. In: T.Y. Feng (ed.): Parallel Processing. Berlin-Heidelberg-New York: Springer 1975, 102-112

/Keller 76/

R.M. Keller: Formal Verification of Parallel Programs. Comm. ACM 19:7, July 1976, 371-384

/Keller 78/

R.M. Keller: Denotational Models for Parallel Programs with Indeterminate Operators. In: /Neuhold 78/, 337-363

/Kessels 77/

J.L.W. Kessels: A Conceptual Framework for a Nonprocedural Programming Language. Comm. ACM 20:12, December 1977, 906-913

/Kosinski 73/

P.R. Kosinski: A Data Flow Language for Operating Systems Programming. SIGPLAN Notices 8:9, September 1973, 89-94

/Kosinski 77/

P.R. Kosinski: A Straightforward Denotational Semantics for Non-Determinate Data Flow Programs. Massachusetts Institute of Technology, Laboratory of Computer Science, Computation Structures Group Memo 157, December 1977

/Kott 78/

L. Kott: About a Transformation System: A Theoretical Study. In: Robinet (ed.): Proceedings of the Third International Symposium on Programming, March, 28-30, Paris: Dunod 1978, 232-247

/Lampert 77a/

L. Lampert: Concurrent Reading and Writing. Comm. ACM 20:11, November 1977, 806-811

/Lampert 77b/

L. Lampert: Proving the Correctness of Multiprocessor Programs. IEEE Transactions on Software Engineering, SE-3:2, March 1977, 125-143

/Landin 65/

P.J. Landin: A Correspondence Between ALGOL 60 and Church's Lambda Notation: Part I. Comm. ACM 8:2, February 1965, 89-101

/Lauer, Campbell 75/

P.E. Lauer, R.H. Campbell: Formal Semantics of a Class of High-Level Primitives for Coordinating Concurrent Processes. Acta Informatica 5, 1975, 297-332

/Levitt 72/

K.N. Levitt: The Application of Program-Proving Techniques to the Verification of Synchronization Processes. Fall Joint Computer Conference, 1972, 33-47

/Lipton 75/

R.J. Lipton: Reduction: A Method of Proving Properties of Parallel Programs. Comm. ACM 18:12, December 1975, 717-721

/Manna et al. 73/

Z. Manna, S. Ness, J. Vuillemin: Inductive Methods for Proving Properties of Programs. Comm. ACM 16:8, August 1973, 491-502

/Mayr 78/

E.W. Mayr: Kollateralität und Ablaufbeschreibungen. Tech. Universität München, Institut für Informatik, TUM-INFO-7802, Dezember 1978

/McCarthy 63/

J. McCarthy: A Basis for a Theory of Computation. In: P. Braffort, D. Hirschberg (eds.): Computer Programming and Formal Systems. Amsterdam: North-Holland 1963, 33-70

/Milne, Milner 77/

G. Milne, R. Milner: Concurrent Processes and their Syntax. University of Edinburgh, Department of Computer Science, Internal Report CSR-2-77, 1977

/Milner 73/

R. Milner: Processes: A Mathematical Model of Computing Agents. Proc. Logic Colloquium, Bristol, Amsterdam: North-Holland 1973, 157-173

/MOD 78/

F.L. Bauer, M. Broy (eds.): Program Construction. Lecture Notes of the International Summer School on Program Construction, Marktobendorf 1978, LNCS 69, Berlin-Heidelberg-New York: Springer 1979

/Morris 79/

J.M. Morris: A Starvation-Free Solution of the Mutual Exclusion Problem. IPL 8:2, February 1979, 76-80

/Neuhold 78/

E.J. Neuhold (ed.): Formal Descriptions of Programming Concepts. Amsterdam: North-Holland 1978

/Nivat 75/

M. Nivat: On the Interpretation of Recursive Program Schemes. Symposia Mathematica, Vol. XV, Instituto Nazionale di Alta Matematica, Italy 1975, 255-281

/Nivat 79/

M. Nivat: On the Synchronisation of Processes. Laboratoire Informatique Theoretique et Programmation, Universite Paris 7, Universite P. et M. Curie, June 1979

/ONERA CERT 78/

D. Comte, G. Durrien, O. Gelly, A. Plas, J.C. Syre: Parallelism, Control and Synchronisation in a Single Assignment Language. SIGPLAN Notices 13:1, January 1978, 25-33

/Owicki 75/

S. Owicki: Axiomatic Proof Techniques for Parallel Programs. Cornell University, Ph. D. Thesis 1975

/Park 79/

D. Park: Notes on Relational Semantics. Copenhagen Winter School on Abstract Software Specifications, 1979

/Partsch, Broy 78/

H. Partsch, M. Broy: Examples for Change of Types and Object Structures. In: /MOD 78/, 421-463

/Pepper 79/

P. Pepper: A Study on Transformational Semantics. Technische Universität München, Dissertation am Fachbereich Mathematik 1979

/Peterson 77/

J.L. Peterson: Petri Nets. Computing Surveys 9:3, September 1977, 223-252

/Petri 62/

C.A. Petri: Kommunikation mit Automaten. Technische Hochschule Darmstadt, Dissertation 1962

/Plickert 77/

H. Plickert: Ein Ansatz zur formalisierten Behandlung der Semantik von Speichern mittels eines Maschinenmodells. Technische Universität München, Dissertation am Fachbereich Mathematik 1977

/Plotkin 79/

G. Plotkin: Some Experiments in the Semantics of Parallelism. Copenhagen Winter School on Abstract Software Specifications, January 1979

/Ramsperger 77/

N. Ramsperger: Concurrent Access to Data. Acta Informatica 8, 1977, 325-334

/Raulefs 79/

P. Raulefs: Semantics of Concurrent Actor Systems. Abstract in: Bulletin of the EATCS 7, 1979, 83-84

/Reed, Kanodia 79/

D.P. Reed, R. K. Kanodia: Synchronization with Eventcounters and Sequencers. Comm. ACM 22:2, February 1979, 115-123

/Robert, Verjus 77/

P. Robert, J.P. Verjus: Toward Autonomous Descriptions of Synchronization Modules. Proc. of the IFIP Congress 77, 1977, 981-986

/Seegmüller 74/

G. Seegmüller: Einführung in die Systemprogrammierung. Reihe Informatik 11, Wissenschaftsverlag, Bibliographisches Institut Mannheim, Wien, Zürich, 1974

/Sintzoff, Van Lamsweerde 75/

M. Sintzoff, A. Van Lamsweerde: Constructing Correct and Efficient Concurrent Programs. Proc. of the Int. Conf. on Reliable Software, 1975, 319-326

/Sintzoff 79/

M. Sintzoff: Principles for Distributed Programs. Proc. of the Int. Symp. on Semantics of Concurrent Computation, Evian, July 2-4, 1979

/Tesler, Enea 68/

L.G. Tesler, H.J. Enea: A Language Design for Concurrent Processes. AFIPS Conference Proceedings 32, 1968, 403-408

/Vuillemin 74/

J. Vuillemin: Correct and Optimal Implementation of Recursion in a Simple Programming Language. J. Comp. Sci. 9:3, June 1974, 332-354

/Wadsworth 71/

C. Wadsworth: Semantics and Pragmatics of Lambda Calculus. Oxford, Ph. D. Dissertation, 1971

/Wijngaarden et al. 75/

A. van Wijngaarden (ed.), B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H.A. Lindsay, L.G.L.T. Mertens, R.G. Fisher: Report on the Algorithmic Language ALGOL 68. Acta Informatica 5, 1975, 1-236

/Yoeli 79/

M. Yoeli: A Structured Approach to Parallel Programming and Control. Proc. of the First European Conf. on Parallel and Distributed Processing, Toulouse, February 1979, 163-169