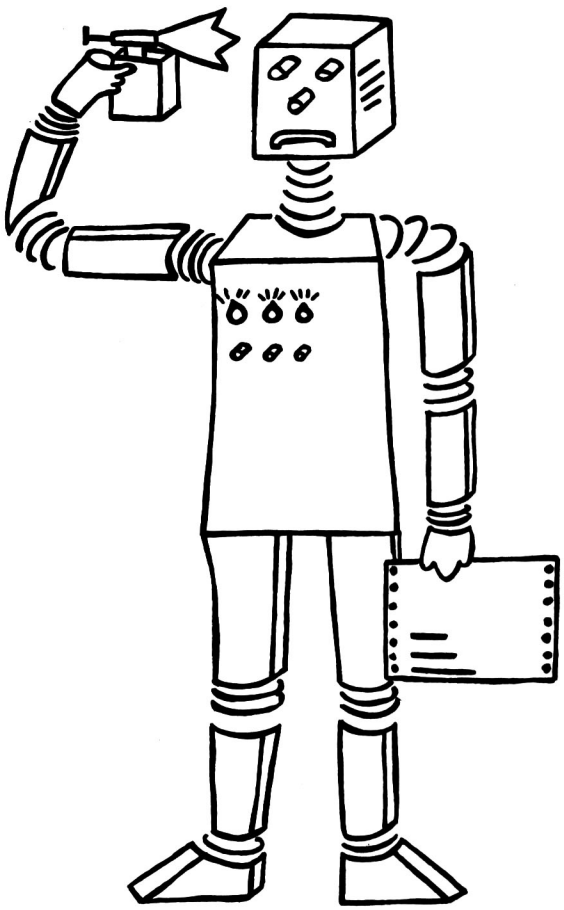
Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern 1, W. Germany

SEKI-Working Paper



Implementation logischer Sprachen auf Multiprozessor-Systemen

Volker Penner

SEKI-Working-Paper SWP-87-05 August 1987

Implementation logischer Sprachen auf Multiprozessor-Systemen

Volker Penner

SEKI-Working-Paper SWP-87-05 August 1987

Abstract

The main contribution of this paper consists in a proposal for implementing parallel logical languages on distributed systems using

* a parallel extension (PWAM) of Warren's abstract machine (WAM) in order to take advantage of approved techniques,

* an approach for modularizing logical programs in order to meet restricted resources, and

* integrating committed choice nondeterminism in order to allow backtrack-free implementations.

The text is divided as follows: first we give an overview over the main notions playing a dominent role in parallel languages, then we introduce the basic process-oriented semantics and discuss the way "or"-parallelism is handled. Starting point of the last chapters is a brief description of the WAM, which is followed by a discussion of the static structure and dynamic behavior of it's parallel counterpart (PWAM) stressing essentially the basic ideas and omitting technical details.

I n h a l t

1. Einleitung.

Logisches Programmieren hat seinen Ursprung im Automatischen Beweisen und der Künstlichen Intelligenz. Anfang der 70'er kam die Idee auf (Kow 74), (Col 73), die bislang ausschließlich für Spezifikationszwecke verwendete Prädikatenlogik als Programmiersprache zu verwenden.

Programmläufe haben dabei zum Ziel, für vorgelegte Behauptungen (Zielklauseln) Beweise zu finden. Als Ableitungsregel dient das Resolutionsverfahren (Rob 65), mit dessen Hilfe gezeigt werden kann, daß eine Menge von Klauseln (logisches Programm) über keine Herbrand-Modelle verfügt. Die Beweise sind konstruktiv, und die in ihrem Verlauf anfallenden Substitutionen definieren auf inkrementelle Weise Ergebnisse. Die Substitutionen werden durch Unifikationsverfahren gewonnen, welche sog. mgu's (most general unifier) liefern.

Nach der grundlegenden Idee von (Kow 79) gehören zur Angabe eines Algorithmus zwei Komponenten, nämlich: Logik und Kontrolle. Durch logische Sprachmittel erfolgt die Spezifikation dessen, was vom Algorithmus erwartet wird, und die Kontrolle beeinflußt, wie Resultate gewonnen werden.

Prolog hat die Horn-Klausel-Teilmenge der Prädikatenlogik als logische Grundlage und nutzt bekanntlich die Reihenfolge der Klauseln und je Klausel die Aufeinanderfolge der Literale im Klauselkörper zur Steuerung einer Top-Down-Depth-First-Strategie. Wesentliches Hilfsmittel ist dabei Backtracking: im Fall eines Fehlschlags (failure) wird die nächste alternative Programmklausel zur Unifikation herangezogen.

Bestehende logische Programmsysteme und dabei insbesondere Prolog-Implementationen sind berechtigter Kritik ausgesetzt. Die Kritikpunkte betreffen:

* Mangelnde Performance
* Ungeeignete Kontrollstrukturen
* Fehlende Konzepte zur Modularisierung

Ein großes Problem ist die mangelnde Effizienz bestehender Implementierungen. Dies wird besonders deutlich bei Problemlösungen, die Gebrauch von nichtdeterministischen Prozeduren machen. Diese führen nämlich i.a. zu zahlreichen Backtrack-Operationen.

Die oben für Prolog erwähnte Ableitungsstrategie und weitere Kontrollmittel wie z.Bsp. der "cut" setzen beim Programmierer eine genaue Kenntnis über die Art der Programmbearbeitung voraus. Dies steht im krassen Widerspruch zum deklarativen Anspruch der Sprache.

Konventionelle Programmiersprachen verfügen in der Regel über Konzepte zur Modularisierung und ermöglichen damit Zerlegungen einer Aufgabe in Teilprobleme, die unabhängig voneinander gelöst werden können. Prolog enthält kein geeignetes Konzept. Dies widerspricht dem Anspruch, eine Sprache zum Rapid Prototyping zu sein.

Eine Verbesserung der Situation wird erhofft durch Nutzung von Parallelität. Die Möglichkeiten dazu hängen wesentlich von der zugrunde liegenden Hardware ab. Grundlegend ist dabei die Unterscheidung paralleler Maschinen nach (Fly 72) in SIMD- und MIMD- Architekturen. Die Maschinen der ersten Gruppe unterstützen Algorithmen, welche eine parallele Verarbeitung gleichartiger Daten vorsehen. MIMD-Architekturen dagegen ermöglichen Zerlegungen von Algorithmen auf der funktionalen Ebene in kooperierende Teile. Für die Verwaltung und Kooperation von Prozessen ist dabei wichtig, ob die Architektur über einen gemeinsamen Speicher verfügt, oder ob es sich um ein verteiltes System mit ausschließlich lokalen Speichern handelt.

Der wesentliche Beitrag dieser Arbeit besteht in einem

Konzept zur Implementation paralleler logischer Sprachen auf verteilten Systemen. Der Vorschlag orientiert sich an den folgenden Zielen:

* Implementation impliziter Parallelität
* Verwendung bewährter Implementationstechniken aus dem sequentiellen Bereich
* Modularisierung logischer Programme
* Integration von "committed-choice" - Nichtdeterminismus.

Die oben angesprochenen Probleme beruhen im wesentlichen auf der Verwendung von Backtracking. Parallele logische Sprachen wie PARLOG (Gre 87), (Cla, Gre 86), Concurrent Prolog (Sha 86) und GHC (Ued 85) (guarded horn clause) verzichten darauf und ersetzen den don't know - Nichtdeterminismus von Prolog durch den sog. don't care - Nichtdeterminismus. Dies gelingt durch Integration von Sprachmitteln (guards, committed choice operator), welche den ODER - Prallelismus abschwächen und die Reihenfolge der Klauseln resp. die der Literale je Klausel für die Auswertung bedeutungslos machen.

Der weitere Text gliedert sich wie folgt:

Zunächst werden die grundlegenden Begriffe zur Parallelität in höheren Sprachen eingeführt, und anschließend wird diskutiert, inwieweit logische Sprachen eine Prozeß-orientierte Sicht mit der Möglichkeit zur Parallelarbeit zulassen.

Problematisch für eine Implementation in einem verteilten System sind Konsistenzfragen erzeugter Variablenbindungen, deren Verteilung auf die lokalen Speicher und Fragen nach geeigneten Protokollen zur Verwaltung von Prozessen und deren Kommunikation.

Anschließend wird ein Modulmechanismus vorgeschlagen, dessen Grundlagen im Rahmen dieser Arbeit nicht dargestellt werden können, der aber im Hinblick auf beschränkte

Ressourcen von großer Wichtigkeit ist.

Den Abschluß der Arbeit bildet dann ein Konzept zur Implementation paralleler logischer Sprachen, welches das erwähnte Modulkonzept berücksichtigt und wesentlichen Gebrauch von durch den Committed-Choice-Nichtdeterminismus gegebenen Eigenschaften paralleler Sprachen macht.

Darüber hinaus integriert das Konzept bewährte Implementationstechniken aus dem sequentiellen Bereich und nutzt wesentliche Merkmale von Transputersystemen (Bar 83), an die in erster Linie bei einer geplanten Implementierung gedacht ist.

2. Parallelität in Programmiersprachen.

Wie in der Einleitung erwähnt ist für die Verwaltung und Kooperation von Prozessen wichtig, ob die zugrunde liegende Architektur über einen gemeinsamen Speicher verfügt, oder ob es sich um ein verteiltes System handelt.

Im ersten Fall spielen Synchronisationsmechanismen zum gegenseitigen Ausschluß, zur Behandlung gleichzeitiger Anforderungen und zur Reihenfolgesteuerung eine besondere Rolle (Fre 87). Zur Programmierung sehen höhere Programmiersprachen Kritische Regionen, Monitore, Pfad - Ausdrücke u.ä. vor. Implementiert werden diese Konzepte unter Verwendung elementarer Hardwareoperationen wie Test-and-Set-, Exchange-Befehle oder Semaphore, die alle auf der Benutzung gemeinsamer Datenstrukturen beruhen.

Im Unterschied hierzu verwenden Mailbox-, Rendez-Vous-Konzepte oder Ports keine gemeinsame Speicherbereiche, sondern haben vielmehr Mechanismen zum Austausch von Botschaften (message passing) als Grundlage. Die Kommunikation wird mit Hilfe von Anweisungen

(i) send message to receiver

zum Senden einer Nachricht an den Empfänger oder an eine Gruppe von Empfängern und Anweisungen

(ii) receive message from sender

zum Empfangen einer Nachricht ermöglicht. Die Senderangabe in (ii) kann fehlen: man spricht dann von einer asymmetrischen Kommunikation. Von einer symmetrischen Prozeßkommunikation ist die Rede, wenn beide Sender wie Empfänger ihren Partner kennen.

Ein wichtiges Unterscheidungsmerkmal betrifft die Synchronisation: die Kommunikation kann synchron oder asynchron erfolgen. Im ersten Fall wartet der Sender bei Ausführung einer send-Anweisung auf das Quittierungssignal vom Empfänger. Die Nachricht ist dann übernommen und beide Prozesse können fortfahren. Umgekehrt wartet auch der Empfänger bei Ausführen einer receive-Anweisung auf die entsprechende Sendeoperation. Man spricht daher auch von einer synchronisierten Kommunikation (Occam (INMOS), CSP (Hoa78)).

Im asynchronen Fall können Nachrichten ohne Rücksicht auf deren Empfang mit beliebiger Häufigkeit gesendet werden. Die Nachrichten sind zu puffern und können z.Bsp. nach der FIFO-Strategie vom Empfänger verarbeitet werden.

Zur Angabe von Adressen in (i), (ii) lassen sich wie in CSP im einfachsten Fall Prozeßbezeichner heranziehen. Zur Implementation weitergehender Kommunikationsmöglichkeiten verwendet man Mailboxes, die sich insbesondere für Client-Server-Beziehungen eignen: die von Servern für Client-Prozesse angebotenen Leistungen werden in einer Mailbox hinterlegt und können von dort mit Hilfe von receive-Anweisungen angefordert werden.

Das Protokoll zur Realisierung muß sichern:

(i) Nach Senden einer Nachricht N an eine Mailbox M muß N allen Prozessen mit einer receive-Operation für M bekannt gemacht werden.

(ii) Nach Übernahme von N durch einen Prozeß steht N nicht mehr zur Verfügung, und alle potentiellen Empfänger müssen davon in Kenntnis gesetzt werden.

In der eingeschränkten Situation, bei der eine Mailbox in receive-Anweisungen nur eines Prozesses vorkommen dürfen, wird die Implementation von (ii) wesentlich erleichtert. Zur Unterscheidung heißen Mailboxen dann Ports nach (Bal 71).

Sprachen mit Monitorkonzept wie Concurrent Pascal (Bri - 75) oder solche auf der Grundlage von send-receive-Anweisungen wie Ada (Ada 83),Occam, CSP ermöglichen die explizite Formulierung paralleler Abläufe und deren Kooperation.

Im Unterschied zu diesen imperativen Sprachen beinhalten funktionale- oder logische Sprachen implizit weitgehende Möglichkeiten zur Nutzung von Parallelität. Programme dieser Sprachen verzichten auf eine detaillierte Festlegung von Berechnungsabläufen; sie spezifizieren vielmehr Zusammenhänge oder Eigenschaften, die von gesuchten Lösungen erfüllt sein müssen.

Im Fall funktionaler Sprachen lassen sich z.Bsp. Argumente von Funktionsaufrufen unabhängig voneinander parallel auswerten (restricted parallelism in Gre 87), oder es ist eine parallele Auswertung funktionaler Ausdrücke und einige ihrer Argumente vorgesehen mit dem Ziel, die über Argumente vermittelte Information inkrementell zu erzeugen und an die Aufrufstellen weiterzugeben (stream parallelism in Gre 87). Dies entspricht der Vorstellung paralleler, kommunizierender Prozesse wie das etwa in CSP vorgesehen ist.

In ähnlicher Form läßt sich die parallele Ausführung der Literale von Programmklauseln (UND - Parallelität) logischer Programme interpretieren. Die Kommunikation erfolgt über gemeinsame Variable und kann auf inkrementelle Weise geschehen, indem die Abarbeitung eines Teilziels unterbrochen wird, sofern Information von einem Nachbarziel fehlt.

Die wechselseitigen Abhängigkeiten erschweren Implementationen von UND - Parallelismus erheblich, weil unabhängig voneinander Ergebnisse gewonnen werden können, die konsistent sein müssen. Wir verzichten hier auf Einzelheiten und verweisen auf eine genauere Diskussion dieses Problems im folgenden Paragraphen.

Zusätzlich besteht in logischen Sprachen die Möglichkeit, alle Klauseln, die eine erfolgreiche Unifikation mit einem Ziel zulassen, parallel auszuwerten (ODER - Parallelität). Der ODER - Parallelismus verwendet eine "breadth - first" Suchregel, nach der alle Alternativen für einen Zielaufruf zugleich berücksichtigt werden. Im Unterschied zum UND - Parallelismus sind die Auswertungen unabhängig voneinander und können unmittelbar parallel ausgeführt werden. Das Verfahren ist "fair" und liefert alle möglichen Lösungen. Nachteilig wirkt sich aus, daß i.a. der Aufwand zur Kontrolle und zur Verwaltung der Ergebnisse übermäßig ansteigt.

Im Hinblick auf beschränkte Ressourcen konzentriert sich diese Arbeit auf Vorschläge zur Implementation von UND - Parallelismus paralleler logischer Sprachen, zumal die bekannten Repräsentanten wie PARLOG, Concurrent PROLOG und GHC ohnehin auf den ODER - Parallelismus verzichten. Diese Sprachen sehen über die Horn-Klausel-Logik hinausgehende Sprachmittel vor, welche den ODER - Nichtdeterminismus einzuschränken gestatten. Einzelheiten hierzu werden in Paragraph 4 diskutiert.

3. Prozeßorientierte Sicht logischer Programme

Logische Programme gestatten zusätzlich zur deklarativen und prozeduralen Interpretation noch eine Prozeßorientierte Sicht. Es bietet sich an,

* Aufrufe oder Zielklauseln als Prozesse mit Programmklauseln als Programm- und Aufrufterme als Datenkomponente,

* Zielklauseln als Netzwerk kooperierender Prozesse und

* gemeinsame Variable als Kommunikationskanäle

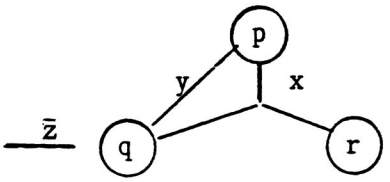
anzusehen.

Sei $p(T_1,\ldots, T_n)$ ein Aufruf.

Die zugehörige Programmkomponente ist die für p definierte Prozedur, d.h. die Programmklauseln mit p als Kopfprädikat. Zu den Prozeßdaten zählen die Terme $T_1,\ldots,T_n$ und weitere Daten zu den Kommunikationskanälen, zum Zustand u.ä.

Zielklauseln und die im Verlauf eines Programmlaufs anfallenden Resolventen sind Und-Verknüpfungen atomarer Formeln, die i.a. durch gemeinsame Variable voneinander abhängig sind. Dementsprechend lassen sich Resolventen als Netzwerk kooperierender Prozesse beschreiben, die über gemeinsame Variable kommunizieren können: Variablenbindungen, die innerhalb eines Prozesses gewonnen sind, können damit Nachbarprozessen zur Verfügung gestellt werden.

Die Zielklausel:- p (x,y,x), q(x,y,z),r(x)
z.Bsp. wird beschrieben durch:



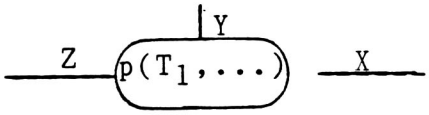
Z ist ein nach außen führender Kanal, dessen offenes Ende das Netzwerk mit der Umgebung (als Prozeß interpretiert) verbindet

Bemerkenswert ist, daß Mehrfachvorkommen einer Variablen in einem Ziel die Anlage nur eines Kanals notwendig macht. Dies ist zulässig, weil die durch Unifikationsschritte gewonnenen Substitutionen für alle Vorkommen einer Variablen dieselben Bindungen vorsehen.

Die Prozesse im Netzwerk führen folgende Schritte durch:

(1) Unifikation
(2) Reduktion
(3) Kommunikation

Zur Erläuterung betrachten wir einen Prozeß $B=p(T_1,\ldots,T_n)$ und gehen von folgenden Kanalverbindungen aus:



Von den Termen $T_1,\ldots, T_n$ wird im Folgenden nur T_1 berücksichtigt. Wir nehmen an:
$T_1=f(Z,g(X,Y), h(X,Z))$

Die Korrespondenz der Kanäle und der Variablen von T_1 ergibt sich durch deren Bezeichnung.

Zu (1):
Wie üblich erfolgen Unifikationsversuche von B mit Klauselköpfen der Prozedur zu p. Gelingt eine Unifikation mit der Programmregel

(i) $p(T_1,\ldots) :-A_1,\ldots, A_m$

und ist μ der dabei gewonnene mgu, dann kann einer der

Schritte (2) oder (3) durchgeführt werden. Im Fall eines Mißerfolges liegt eine Situation vor, deren Behandlung von der zugrunde liegenden Sprache und insbesondere von deren Semantik abhängt: Prolog implementiert i.a. Backtracking-Verfahren, welche in systematischer Weise die möglichen Alternativen für Unifikationszwecke heranziehen.

Parallele logische Sprachen gestatten dagegen backtrackfreie Implementationen. Dies gelingt durch Hinzunahme von Sprachmitteln, welche insbesondere eine Verfeinerung der Unifikation derart ermöglichen, daß zwei Situationen unterschieden werden können: die eine führt zur Suspendierung des Prozesses und die andere zur Aufgabe des Programmlaufs. Einzelheiten hierzu werden in Paragraph 4 diskutiert.

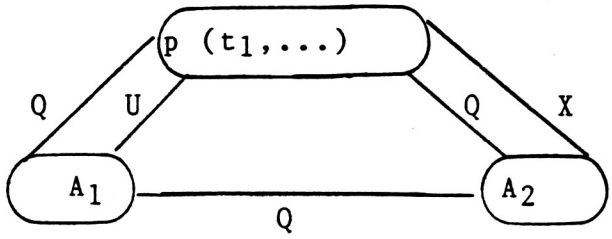
Zu beachten ist die symmetrische Auswirkung von μ: die Unifikation bewirkt i.a. Bindungen für Variable in T_1 und solche in t_1. Zur Demonstration sei m = 2 in (i), t_1 = f(1(U,V), Q,h(X,Y)), A_1 = q(m(Q,U)) und A_2=r(Q,X). Als mgu μ ergibt sich μ = { Z / 1(U,V) , Q / g (X,Y) }.

Zu (2)
Ein Reduktionsschritt besteht in der Verschmelzung des Aufrufs p (T_1,...) mit dem Kopf p (t_1,...) der Klausel zu p (T_1,...) μ = p (t_1,...)μ = p (f(1(U,V),g(X,Y),h(X,Z)), der Generierung von Prozessen zu $A_1\mu$, $A_2\mu$ und der Anlage von Kanälen.

Die Verschmelzung hat eine Aktualisierung von Prozeßdaten zur Folge. Im Beispiel ist T_1 durch $T_1\mu$ = f(1(u,v),g(x,y), h(X,Z)) zu ersetzen.

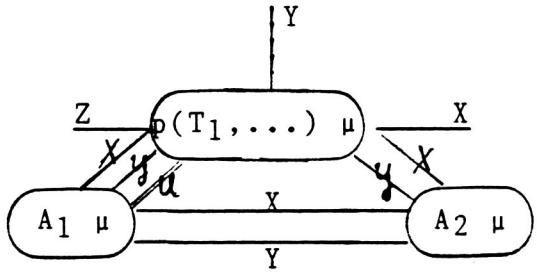
Die beiden Prozesse haben die Prozedur zu q resp. zu r als Programmkomponente und m (Q,U)μ = m (g(X,Y),U) bzw. Qμ = g(X,Y),X als Datenkomponente.

Grundlage für die Generierung von Kanälen ist die Klausel (i) und der mgu μ :



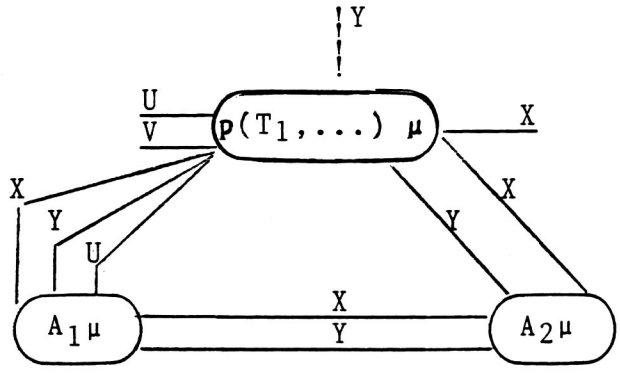
$p(t_1,\ldots)$:- A_1 , A_2

Die Bindung Z/1(U,V)hat offenbar keine Auswirkung.
Q / g(X,Y) dagegen bewirkt:



Zu (3):

Kommunikationsschritte haben zum Ziel, gewonnene Bindungen über ursprünglich für den Aufruf definierte Kanäle an die Umgebung weiterzuleiten. Für das Beispiel heißt dies, daß Z / 1(U,V) zu propagieren ist, was dann zur Anlage eines U- und eines V-Kanals als Ersatz für den "alten" Z - Kanal führt:



Bemerkenswert ist, daß ein Propagierungsschritt weitere Kommunikationen nach sich ziehen kann und im ungünstigen Fall weite Bereiche des Prozeßnetzes zu Kommunikationsschritten veranlaßt.

Bei den obigen Betrachtungen ist r > 1 angenommen. Für r = 1 hat man $p(t_1,...)$:- A_1 , und es kann auf die Generierung neuer Prozesse verzichtet werden: der für $p(t_1,...)$ bestehende Prozeß übernimmt die Rolle des für A_1 zu generierenden Prozesses.

Im Fall r = o liegt ein Fakt $p(t_1,...)$:- vor, der Prozeß zu $p(t_1,...)$ kann nach Propagieren der erzielten Variablenbindungen aufgegeben werden.

Der dynamische Ablauf ist damit durch ein pulsierendes Netzwerk von Prozessen gekennzeichnet, dessen Startsituation durch eine vorgelegte Zielklausel definiert ist. Charakteristisch ist dabei, daß sich für jedes Zielatom A eine baumartige Struktur von Prozessen mit A als Wurzel aufbaut: Kanalverbindungen bestehen zwischen Vater- und Sohnknoten untereinander. Unifikationsschritte werden von "Randprozessen" der Baumenden durchgeführt, und die gewonnenen Variablenbindungen ergänzen die Resultatsubstitution und bewirken i.a. eine Modifikation der Kanalstruktur.

Aus den bisherigen Betrachtungen ergeben sich zwei Problembereiche, welche

(1) die Verwaltung der Variablenbindungen und
(2) die Abwicklung der Prozeßkommunikation

betreffen.

3.1 Verwaltung der Variablenbindungen

Die bisherige Sicht setzt voraus, daß die den Prozessen zukommenden Daten jeweils vollständig und lokal verfügbar sind. Hiervon kann aber nicht ausgegangen werden, da keine Annahmen über den zeitlichen Aufwand zur Erzeugung und zum Transfer von Variablenbindungen gemacht

werden sollen. Für die projektierte verteilte Implementation ist vielmehr ein Verhalten der Prozesse realistisch, bei dem in global unkoordinierter Weise Kommunikations- und Unifikationsschritte durchgeführt werden. Kennzeichnend ist damit, daß die Verteilung der insgesamt im System verfügbaren Information über die Kanäle i.a. nicht abgeschlossen ist, zumal durch Unifikation immer wieder zusätzliche Variablenbindungen auftreten werden.

Es eröffnet sich die Option, die Ergebnisse nur dort zu speichern, wo sie gewonnen sind und Bindungen für andere Prozesse dort durch geeignete Adressen zugänglich zu machen. Kopierschritte fallen i.a. nur dann an, wenn Prozesse aufzugeben sind.

Die Folge einer solchen Organisation ist natürlich, daß die für Unifikationen benötigten Informationen i.a. lokal nicht verfügbar sind, sodaß für ihre Durchführung zahlreiche Kommunikationsvorgänge anfallen.

Für die Verwaltung von Variablenbindungen und von Kanälen ist wichtig, daß Konflikte auftreten können. Zielklauseln sind nämlich i.a. nicht unabhängig: sie können gemeinsame Variable aufweisen, oder es können sich im Verlauf des inkrementellen Aufbaus von Bindungen gemeinsame Variable ergeben. Damit besteht die Möglichkeit, daß Variablen in parallel ablaufenden Prozessen unabhängig voneinander Bindungen erfahren, die aufgrund von Kommunikationsschritten miteinander zu vergleichen sind. Hierbei können durchaus verschiedene Bindungen auftreten, ohne daß eine Fehlersituation vorliegt.

Es muß dann allerdings gewährleistet sein, daß die Bindungen unifizierbar sind. Dies berücksichtigt die Tatsache, daß ein endgültiges Substitutionsergebnis auf unterschiedliche Weise inkrementell zusammengesetzt werden kann.

Zur Implementation eines Testverfahrens ist der zeitliche Ablauf der Erzeugung von Kanälen und deren Abhängigkeit von zuvor erzeugten Kanälen zu registrieren.

Das Problem vereinfacht sich wesentlich, wenn mit gerichteten Kanälen gearbeitet werden kann. Konfliktsituationen ergeben sich dann höchstens für Fälle, wo Kanäle zusammenstoßen. Der sich hieraus ergebende Vorteil besteht darin, daß sich die oben angedeuteten Konsistenztests allein auf diese Stellen beschränken können.

Parallele logische Sprachen sehen eine Modifikation der Unifikationsprozedur vor, die im wesentlichen auf der Auszeichnung von Richtungen für den Informationstransfer vom Aufruf zum Klauselkopf (Input) und vom Klauselkopf zum Aufruf (Output) beruht, so daß die oben angedeutete vereinfachte Situation vorliegt.

Einzelheiten hierzu werden im folgenden Paragraphen diskutiert.

3.2. Abwicklung der Prozeßkommunikation

Eine Organisation, welche garantiert, daß jederzeit die im Netz vorhandenen Bindungen einer Variablen allen über Kanalverbindungen erreichbaren Prozessen verfügbar sind, ist aufwendig und widerspricht der grundsätzlichen Philosophie verteilter Systeme.

Damit ist von einem asynchronen Verhalten des Gesamtsystems auszugehen, bei dem Prozesse weitgehend unabhängig voneinander arbeiten. Es werden Variablenbindungen in unkoordinierter Weise erzeugt und über das Netz verteilt, so daß die Gefahr besteht, daß Bindungen nicht in der Reihenfolge ihrer Erzeugung ausgewertet werden.

Die Konventionen zur Unifikation sichern aber, daß dies nicht möglich ist: der Unifikationsalgorithmus setzt nämlich für jeden Schritt voraus, daß die verwendeten Klauseln neue Variablen enthalten, so daß Bezeichner generierter Kanäle neu gegenüber den bisher verwendeten sind.

Dies sichert, daß Bindungen für Variable in der richtigen Reihenfolge ausgewertet werden: enthält ein Substitutionsterm

Variable und sind für einige davon bereits Bindungen gewonnen, dann ist durch die Wahl der Variablenbezeichner und ihre Verwendung als Kanalbezeichner gesichert, daß die zeitlich zuerst erzeugte Bindung ausgewertet sein muß, bevor die "Sekundärbindungen" berücksichtigt werden.

Es bietet sich an, auch für die Kommunikation der Prozesse untereinander ein asynchrones Protokoll vorzusehen. Bei einer synchronen Organisation fällt nur nachteilig ins Gewicht, daß ggf. "busy-waiting" Zeiträume auftreten, die für nützliche Arbeit verloren gehen. Zugleich eröffnen sich durch die Verwendung von Puffern für die Ein- Ausgabe der Prozesse Möglichkeiten, die Abfolge der Unifikationsschritte und die der Kommunikationen aufeinander abzustimmen, so daß eine optimale Nutzung der Prozesse möglich ist. Wir kommen im Praragraph 6 auf diese Möglichkeit zurück.

Der bekannte Nachteil asynchroner Protokolle - Prozesse verarbeiten evtl. veraltete Daten - fällt hier nicht ins Gewicht: alle von Nachbarprozessen übernommenen Variablenbindungen müssen verarbeitet werden.

Grundsätzlich ist festzulegen, wie die Steuerung innerhalb der einzelnen Prozesse ablaufen soll. Hier bietet sich an, eine Mast-Slave-Organisation vorzusehen: ein Hauptprozeß übernimmt die Verwaltung der Daten (Variablenbindungen, Prozeßqueues etc.) und veranlaßt Unifikations-, Kommunikations-, Verarbeitungsschritte etc.

Weitere Aufgaben betreffen die Behandlung von Fehlersituationen, die Generierung und Terminierung von Prozessen und die damit verbundene Verwaltung von Kanälen. Einzelheiten hierzu ergeben sich im folgenden Paragraphen, der insbesondere zeigt, daß für parallele logische Sprachen backtrack-freie Implementationen möglich sind, so daß eine PROLOG entsprechende Prozeßverwaltung wegfällt.

Nach der bisherigen Diskussion ergeben sich einige Anforderungen an die Hardware:

3.3 Hardwareanforderungen

Der Zielsetzung entsprechend wird von einem verteilten System von Prozessoren ohne gemeinsamen Speicher ausgegangen. Ferner wird angenommen, daß die Prozessoren synchronisierte Kommunikationen und nichtdeterministische Auswahloperationen (committed choice nondeterminism) unterstützen. Konkret ist dabei an ein Netz von Transputern (Bar 83) gedacht, welche das Modell kommunizierender Prozesse in besonderem Maße unterstützen.

Transputer sind Mikrocomputer mit lokalem (4 K - Byte) Speicher und vier Links für Punkt-zu-Punkt-Verbindungen mit Nachbartransputern. Die Standard-CPU kann zur Implementation der üblichen Datentransformationen und Kontrollstrukturen imperativer Sprachen herangezogen werden, der Speicher dient zur kurzfristigen Ablage von Zwischenergebnissen, und die vier Links sind in Form unabhängiger DMA-Kanäle implementiert, die physikalische bidirektionale Kanäle zu weiteren Transputern im Netz ermöglichen.

Bemerkenswert sind die Instruktionen zur Verwaltung paralleler Prozesse und die zur synchronisierten Kommunikation und nichtdeterministischen Auswahl. Sie unterstützen damit unmittelbar die auf Dijkstra zurückgehenden "guarded-commands" und "cobegin-coend-"Konstrukte, die sich ihrerseits in CSP und der auf CSP basierenden Implementationssprache Occam des Transputers wiederfinden.

Die Diskussion paralleler logischer Sprachen wird zeigen, daß dort analoge Konzepte integriert sind: die nichtdeterministische Auswahl (Alt-Konstrukt in Occam) entspricht dort der Ausführung des sog. Commitment-Operators, der unter i.a. mehreren Kandidatenklauseln genau eine Klausel zur Durchführung des nächsten Resolutionsschritts auswählt.

3.4 Zusammenfassung

Die obigen Betrachtungen zeigen den Aufwand, der zur Implementation von Kommunikationsprotokollen und Verwaltungsmechanismen der Variablenbindungen zu treiben ist. Eine Ursache hierfür liegt in der feinen Granularität. Das Modell arbeitet mit Prozessen, zwischen denen häufig Kommunikationen stattfinden. Abhilfe kann erreicht werden durch

(1) Bindung von Unifikationen an Voraussetzungen

(2) Erweiterung des Aufgabenbereichs der einzelnen Prozesse.

Maßnahmen zu (1) haben das Ziel, die Auswirkungen von Unifikationen einzuschränken. Wir betrachten zunächst Möglichkeiten zu (1) im folgenden Paragraphen und dann zu (2) in Paragraph 5.

4. Parallele Logische Sprachen

Es gibt eine Reihe von Vorschlägen in der Literatur, welche durch zusätzliche Sprachmittel oder durch eine geeignete Semantik den in der Einleitung erwähnten Schwachpunkten logischer Sprachen begegnen.

Parallelismus kann dabei in unterschiedlicher Form genutzt werden als ODER-, UND-, Strom- und Suchparallelismus.

Zu den Sprachen mit Strom-Parallelismus zählen u.a. GHC, Concurrent Prolog, PARLOG. Alle drei verwenden nach Vorschlägen von (Dij 75) Guards und sehen Programmregeln als "guarded clauses" in der Form

$$H :- G_1, \ldots, G_n \mid B_1, \ldots, B_m$$

vor. H, die G_i und die B_i sind atomare prädikaten logische Ausdrücke der Form $p(t_1, \ldots, t_n)$, wo p ein k-stelliges Prädikatzeichen ist und die t_r Terme sind.

Der Kopf H und die Guards G_i stehen vor dem "Commitment" - Operator "|" (in PARLOG mit ":" bezeichnet). Sie bilden den Guard - Teil der Regel, während die B_i den Body - Teil ausmachen. Bei deklarativer Lesart stehen "," und "|" für "und" und :- für eine Implikation "←" .

Guards dienen

(1) zur Spezifikation von Synchronisationsregeln und
(2) zur Implementation von "committed-choice" - Nichtdeterminismus.

4.1 Guards als Synchronisationsmittel

Wie üblich sieht die operationale Semantik auch paralleler logischer Sprachen vor, Zielklauseln mit Hilfe von Klauseln eines Programms auf die leere Klausel zu reduzieren. Dies geschieht durch Unifikationsversuche des Aufrufs mit den Kopfformeln der zugehörigen Prozedur im Programm. Hierbei werden Guards ebenso behandelt wie die Body-Formeln auch. Es macht demnach Sinn davon zu reden, daß ein Guard zutrifft, (success) bzw. daß die Guards einer Klausel zutreffen.

Entsprechend können einzelne Guards und damit ganze Guard - Teile fehlschlagen (failure).

In parallelen Sprachen gibt es einen weiteren Zustand : Guards oder Unifikationen können suspendiert sein. Solange aufgrund fehlender Information nicht entschieden werden kann, ob der Guard-Teil einer Programmklausel zutrifft, wird eine weitere Auswertung abgebrochen. Damit besteht die Möglichkeit, die Auswahl einer nächsten Klausel solange hinzuzögern, bis mit Hilfe zusätzlicher Information von "außen" eine begründete Auswahl möglich ist.

Die drei genannten Sprachen unterscheiden sich im Hinblick auf die zusätzlichen Anforderungen an die Unifikationsprozedur. In jedem Fall wird aber erreicht, daß die drei Zustände "Erfolg", "Fehlschlag", "Suspendierung" unterschieden werden können.

Zugleich ergeben sich Richtungen für die Kanäle, die beim Transfer von Variablenbindungen zwischen Prozessen einzuhalten sind. Besonders deutlich wird das für PARLOG. Daher soll sich die folgende Diskussion besonders auf diese Sprache beziehen.

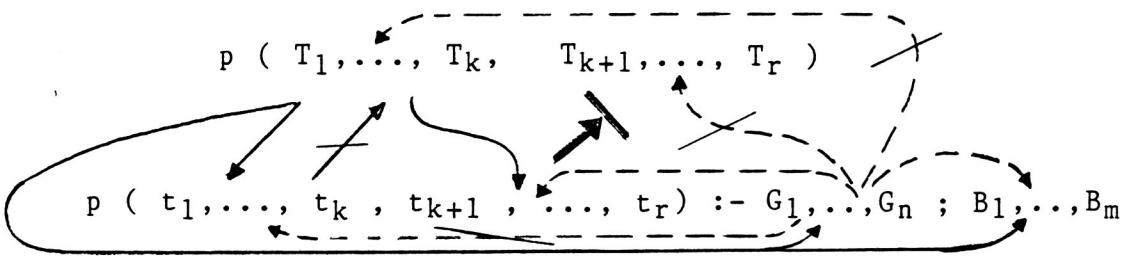
PARLOG verwendet Eingabe- und Ausgabepositionen für die Argumente von Prädikatzeichen und macht dies mit Hilfe von Mode - Deklarationen der Form
mode p(...?, ...?, ...!, ...?, ...)
kenntlich : die mit ? versehenen Stellen sind Input - Mode - Positionen, und die anderen sind Output - Mode - Positionen. Für jedes in einem PARLOG - Programm vorkommende Prädikat wird eine Mode-Deklaration erwartet.

Zur Beschreibung der Auswirkungen von Annotationen "?","!" auf die Unifikationsprozedur sei $p(T_1,\ldots,T_r)$ Aufruf und $p(t_1,\ldots,t_r)$:- $G_1,\ldots, G_n$: $B_1,\ldots, B_m$ Programmregel. Die T_i $(1 \leq i \leq k)$ stehen auf Input-Mode Positionen und die T_j $(k < j \leq r)$ auf Output - Mode - Positionen.

Die PARLOG - Unifikation erfolgt in zwei Phasen:

(i) einer Input - Unifikation und Auswertung von Guards
(ii) einer Output - Unifikation.

Das folgende Bild zeigt (i):



(i) In der ersten Phase wird parallel eine Input - Unifikation und eine Auswertung der Guards vorgenommen.

a) Die Input - Unifikation ist eine Unifikation der T_i und t_i $(1 \leq i \leq k)$ auf Input-Mode-Position. Sie ist erfolgreich, falls der Informationstransfer von den T_i zu den t_i verläuft, d.h. falls keine Variable in den T_i durch einen nichtvariablen Term gebunden wird. Die Input-Unifikation

kann suspendieren oder mit einem Fehlschlag terminieren. Letzteres ist der Fall, wenn eine Unifikation nicht möglich ist. Ist sie nur so möglich, daß Variable in einem T_i ($1 \leq i \leq k$) durch nichtvariable Terme zu binden sind, dann suspendiert die Input-Unifikation.

b) Die Auswertung der Guards erfolgt wie die der Aufrufe.

Eine erfolgreiche Auswertung setzt allerdings voraus, daß Variable in den T_i ($1 \leq i \leq r$) und den t_j ($1 \leq j \leq k$) nicht gebunden werden. Erlaubt dagegen ist die Erzeugung von Bindungen für Variable in den t_j ($k < j \leq r$), den Atomen B_s im Body-Teil und den G_t ($1 \leq t \leq n$) selbst.

Ergibt sich im Verlauf der Auswertung eines Guards eine suspendierende Input-Unifikation, dann suspendiert die Guard-Auswertung selbst. Entsprechend kann die Guard-Auswertung fehlschlagen oder erfolgreich sein.

Wichtig für die erste Phase ist, daß keine Bindungen für aktuale Output-Parameter T_j ($k < j \leq r$) wirksam gemacht werden (im Bild durch $\overline{\uparrow}$ kenntlich). Dies geschieht in der zweiten Phase nach Ausführen des Commitment Operators.

(ii) In (Cla, Gre 86) wird verlangt, daß Output-Mode-Positionen in Aufrufen lediglich Variable aufweisen dürfen, so daß eine Output-Unifikation auf die Zuweisung der t_i ($k < i \leq r$) zu den entsprechenden Ausgabevariablen hinausläuft.

Diese einschränkende Bedingung kann verallgemeinert werden: in (Gre 87) wird auf die obige Forderung verzichtet, die t_i ($k < i \leq r$) dürfen beliebige Terme sein.

Gelingt dann die Unifikation der t_i, T_i ($i < k \leq r$) nicht, dann schlägt die Output-Unifikation fehl, und es liegt ein Fehler vor, der zum Abbruch des Programmlaufs führt. Andernfalls ist von einer erfolgreichen Output-Unifikation zu fordern, daß Variablen in den t_i ($k < i \leq r$) nicht durch nichtvariable Terme gebunden werden. Ist diese Bedingung nicht erfüllt, dann suspendiert die Output-Unifikation.

4.2 Committed-Choice-Nichtdeterminismus

Im Unterschied zu PROLOG ist für einen PARLOG - Aufruf eine ODER - parallele Suche nach Kandidatenklauseln vorgesehen. Als Kandidaten kommen die Klauseln der zum Aufruf gehörenden Prozedur in Betracht, die eine erfolgreiche Input-Unifikation zulassen und deren Guards sich als zutreffend erweisen (erste Phase der Unifikation in PARLOG).

Stehen mehrere Kandidatenklauseln zur Verfügung, dann erfolgt eine Auswahl in nichtdeterministischer Weise.

Aufrufe können erfolgreich sein, fehlschlagen oder suspendieren. Sie suspendieren, falls es suspendierende Kandidatenklauseln gibt und keine Kandidatenklausel erfolgreich ist. Eine Klausel ist dann erfolgreich, wenn die erste Phase der PARLOG-Unifikation erfolgreich ist, d.h. wenn die Klausel eine erfolgreiche Input-Unifikation zuläßt und die Guards zutreffen. Sie ist suspendiert, falls die Input-Unifikation suspendiert ist oder die Auswertung eines Guards suspendiert ist und alle anderen Guards nicht fehlschlagen.

Ein Aufruf schlägt fehl, wenn alle Kandidatenklauseln fehlschlagen, d.h. wenn für jede Kandidatenklausel die Input-Unifikation oder die Auswertung eines Guards fehlschlägt.

Der Aufruf istschließlich erfolgreich, wenn es eine erfolgreiche Kandidatenklausel gibt.

Die oben skizzierte Auswahlregel einer Klausel für einen Aufruf manifestiert sich in der Ausführung des Commit-Operators für den Fall eines erfolgreichen Aufrufs. Er bewirkt, daß

* unter den erfolgreichen Klauseln eine Klausel ausgewählt wird
* die einmal getroffene Wahl nicht rückgängig gemacht wird

* Bindungen für die Ausgabe wirksam werden (zweite Phase der PARLOG-Unifikation)
* Die Atome im Body-Teil der ausgewählten Klausel in UND-parallele Aufrufe überführt werden.

Damit kann bei der Implementation paralleler logischer Sprachen auf eine feste Strategie zur Auswahl von Klauseln mit der Möglichkeit zur Rücknahme verzichtet werden.

Die Semantik ermöglicht vielmehr eine backtrack-freie Implementation mit paralleler Suche nach Programmklauseln: der ODER-Parallelismus ist ersetzt durch eine ODER-parallele Suche mit anschließender nichtdeterministischer Auswahl unter den erfolgreichen Klauseln. Nach Ausführung des Commitment-Operators ist die Auswahl abgeschlossen und die Auswertung von "Konkurrenz"-Klauseln kann aufgegeben werden.

4.3 Zusammenfassung

Die obige Diskussion zeigt zweierlei:

(i) Der Informationsfluß ist gerichtet
(ii) Backtrack-freie Implementationen sind möglich

Zu (i):
Der Informationsfluß verläuft bei erfolgreicher Input-Unifikation (und dies schließt Fälle ein, bei denen einer erfolgreichen Input-Unifikation eine Reihe von suspendierten Input-Unifikationen vorangehen) über die Input-Mode-Positionen vom Aufruf zu den Input-Mode-Argumenten im Kopf einer Kandidatenklausel. Die parallel zur Input-Unifikation ablaufende Guard-Auswertung erzeugt im Erfolgsfall keine Bindungen für Variable im Aufruf bzw. in Input-Mode-Argumenten des Regelkopfes.

Nach Ausführung des Commitment-Operators werden die gefundenen Bindungen für die Call-Argumente auf Output-Mode-Positionen wirksam. Die Output-Unifikation bewirkt im Erfolgsfall einen Informationstransfer vom Regelkopf zum Aufruf.

Die in Paragraph 3 betrachteten Kanäle sind demnach gerichtet.

Zu (ii):
Die Eigenschaften (i), (ii) erleichtern eine Implementation erheblich. (i) hat wesentlichen Einfluß auf die Verwaltung von Variablenbindungen und die Durchführung von Konsistenztests, und (ii) hat zur Folge, daß Regelauswahlen nicht zurückgenommen werden müssen. Dies ist besonders bedeutsam für eine verteilte Implementation.

5. Spezifikation von Moduln

In 3.4 hat sich als zweite Möglichkeit zur Aufwandsminderung ergeben, für Prozesse einen umfangreicheren Aufgabenbereich vorzusehen. Dies kann z. Bsp. dadurch geschehen daß eine Gruppe von Prozessen im Sinne von Paragraph 3 zu einem Prozeß zusammengefaßt wird.

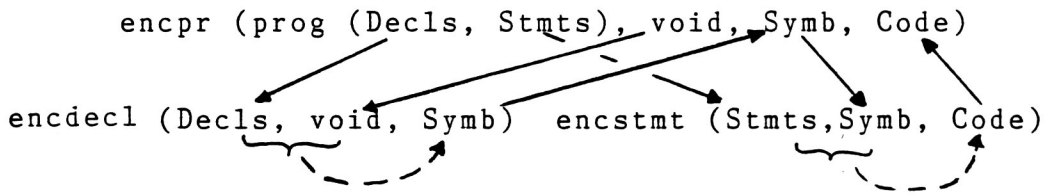
Es ist dabei klar, daß eine solche Maßnahme den impliziten Parallelismus logischer Programme nicht voll ausschöpft. Andererseits besteht erst hierdurch die Möglichkeit, eine Implementation für verteilte Systeme mit einer bescheidenen Anzahl von Prozessen durchzuführen.

Diese Idee führt dazu, Programme in Teilprogramme zu zerlegen, die dann Grundlage für umfassendere Prozesse sind. Zugleich ergibt sich die Möglichkeit, bewährte Techniken aus dem sequentiellen Bereich zu übernehmen.

Aufteilungen ergeben sich i.a. auf natürliche Weise aus der Kommunikationsstruktur. Wir betrachten dazu ein Beispiel aus dem Compilerbau (War 80):

Beispiel

Für die semantische Analyse und für Übersetzungszwecke werden im Compiler Symboltabellen verwendet, die durch Analyse des Deklarationsteils von Programmen aufgebaut werden und deren Eintragungen z.Bsp. zur Übersetzung von Ausdrücken wieder benötigt werden. Eine PARLOG - Implementation führt zu folgenden Abhängigkeiten:



Eine Modularisierung kann vorsehen, daß der Aufbau der Symboltabelle durch einen Dekl-Modul wahrgenommen wird, die Behandlung der Statements durch einen Statement-Modul und die Übersetzung von Ausdrücken durch einen Ausdrucks-Modul bewirkt werden.

Die mit einer Modularisierung verbundene Vorstellung ist die, daß im Zuge der Abarbeitung eines Programms Prozesse für Moduln generiert werden (und entsprechend terminieren und verschwinden) und Strom-Parallelismus zwischen parallel ablaufenden Modul-Prozessen möglich ist.

Im Beispiel läuft das darauf hinaus, daß zugleich der Aufbau der Symboltabelle und die Übersetzung von Ausdrücken stattfinden kann. Die jeweils gewonnene (partielle) Symboltabelle steht den Ausdrucks-Prozessen zur Verfügung. Diese werden suspendiert, falls die Information in der Symboltabelle nicht ausreicht, d.h. falls im Ausdruck ein Bezeichner referiert wird, der noch nicht analysiert oder transferiert ist.

Wir gehen im Folgenden von modularisierten logischen Programmen aus, wobei verlangt wird, daß für jeden Aufruf und für jede dazu ausgewählte Programmklausel (nach Ausführung des Commitment-Operators) entschieden werden kann, ob diese Klauselanwendung entweder Startklausel eines zu

generierenden Prozesses ist oder zum weiteren Ablauf des auslösenden Prozesses dient. Auf Einzelheiten hierzu kann in dieser Arbeit nicht eingegangen werden.

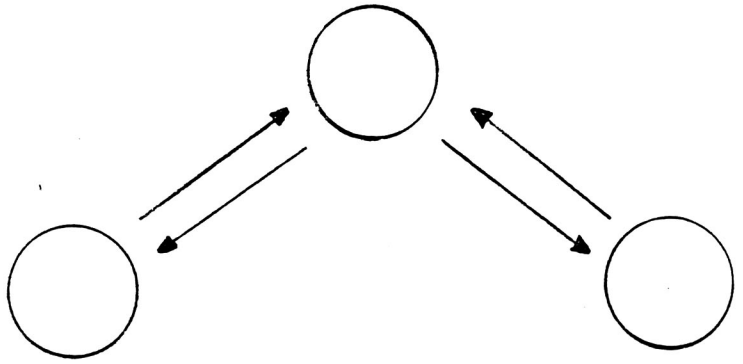
6. Parallele Warren Maschine

Der folgende Vorschlag zur Implementation paralleler logischer Sprachen sieht die Verwendung paralleler abstrakter Maschinen vor. Das Modell ist eine parallele Erweiterung der üblicherweise bei PROLOG-Implementationen verwendeten "Warren Abstract Machine" (War 83), (Gab et al. 85) und wird daher mit PWAM abgekürzt.

Die Darstellung beschränkt sich auf das Konzept und die zugrunde liegenden Ideen.

Die grundsätzliche Arbeitsweise der PWAM orientiert sich an den in Paragraph 3 entwickelten Vorstellungen mit dem Unterschied, daß die dynamisch angesprochenen Module auf PWAM's ablaufen.

Das oben entwickelte Bild eines Netzes von Prozessen findet sich hier vergröbert wieder: Teile des früheren Prozeßbaums werden nun innerhalb der PWAM's als Kontrollbaum verwaltet.



Die Bearbeitung der Module erfolgt sequentiell und bewirkt den Aufbau von Beweisbaumteilen innerhalb der PWAM's.

Bei Auswahl einer Regel, welche nicht für den eigenen Beweisbaum genutzt werden kann, ist eine neue PWAM zu generieren, die über Kanäle mit der "Vater-PWAM" verbunden ist.

Im weiteren Ablauf besteht für die "alte" PWAM die Möglichkeit, bisher nicht bearbeitete Äste ihres Beweisbaums zu ergänzen. Ist dies nicht weiter möglich, dann werden nur noch Vermittlerdienste für den Transfer von Variablenbindungen wahrgenommen.

6.1 Warren Abstract Machine

Effiziente PROLOG - Implementationen arbeiten mit Compilern für abstrakte Zielmaschinen. Die bekannteste unter diesen Maschinen ist die WAM, welche sich zur Abarbeitung iterativ formulierter Programme im WAM - Assembler eignet.

Die Techniken zur Transformation rekursiver Programme in iterative sind von den imperativen Sprachen her bekannt und laufen im wesentlichen auf die Verwendung eines Laufzeitstacks hinaus.

Bemerkenswert am WAM - Assembler sind die Instruktionen zum Aufbau und zurUnifikation von Termen. Mit Hilfe zweier Modi (read-, write - Modus) bewirken sie Vergleiche korrespondierender Terme und veranlassen im Bedarfsfall Unifikationen und Bindungen für Variable. Hierzu verfügt die WAM über einen Heap, der die gewonnenen Resultate aufnimmt.

Zusätzlich gibt es Instruktionen zur Programmkontrolle, Auswahl von alternativen Klauseln etc.

Zu den Datenstrukturen der WAM zählen der Heap, zwei weitere Stacks

* Kontrollstack
* Trace - Stack

und Argumentregister.

Der Kontrollstack entspricht dem Laufzeitstack imperativer Sprachen: er besteht (im Regelfall) aus einer Abfolge sog.

Choice-Points und Environments, die den Aufrufen von Zielklauseln entsprechen.

Die Choice-Points beinhalten die übliche Kontrollinformation für Rücksprünge u.ä. und darüberhinaus Informationen zur Realisierung des Nichtdeterminismus von PROLOG - Programmen: Hierzu zählen insbesondere Verweise auf die aktualen Parameter und Angaben für die nächste verfügbare Alternative. Diese Information wird im Fall eines Fehlschlags genutzt, um die nächste Alternative anzuwenden und für diese Parameter und benötigte Variablenbindungen bereitzustellen. Dies letztere geschieht mit Hilfe von Environments, welche entsprechende Verweise in den Heap aufnehmen.

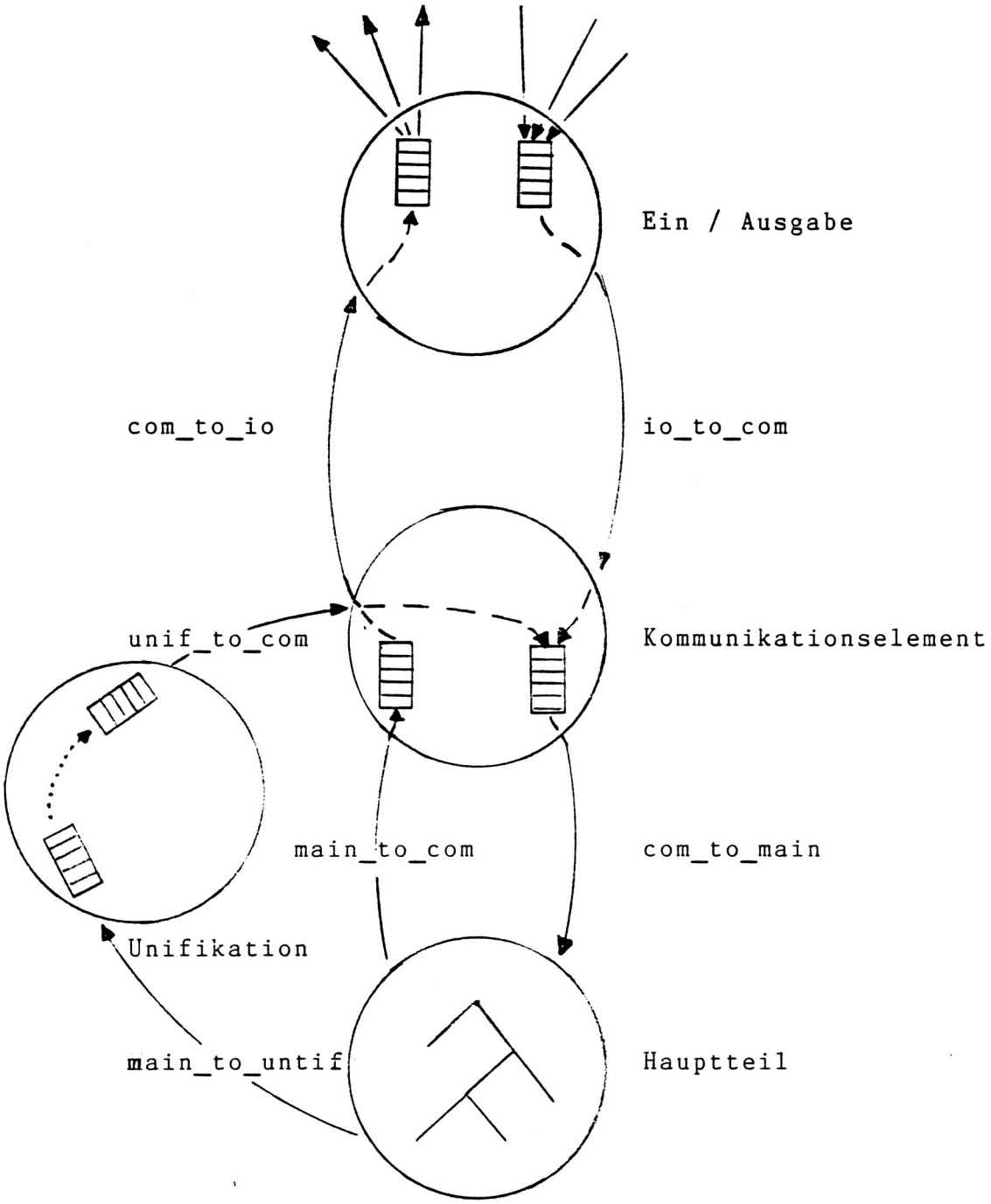
Der Trace - Stack dient zur Implementation von Backtracking: über ihn sind nämlich diejenigen Variablen zugänglich, deren Bindungen im Fall eines Backtrack-Schritts zurückgenommen werden müssen.

Die Argumentregister schließlich werden hauptsächlich dazu verwendet, die aktualen Parameter für Aufrufe zur Verfügung zu stellen, um folgende Unifikationsschritte zu ermöglichen.

6.2 Struktur der Parallelen WAM (PWAM)

Die Aufgabe der PWAM besteht in der Bearbeitung ihres Moduls, in der Erzeugung von Variablenbindungen, deren Speicherung und Weitergabe an Nachbar - PWAM's und der Abwicklung des Kommunikationsprotokolls. Weiter sind Konsistenztests für Mehrfachbindungen durchzuführen, und es sind Maßnahmen im Fehlerfall zu ergreifen, die auf einen Abbruch der Bearbeitung und eine Fehlerausgabe hinauslaufen. Hierzu verfügt jede PWAM über vier Prozesse:

* Ein / Ausgabe
* Hauptteil
* Unifikation
* Kommunikationselement


Ein / Ausgabe
com_to_io
io_to_com
unif_to_com
Kommunikationselement
main_to_com
com_to_main
Unifikation
main_to_untif
Hauptteil

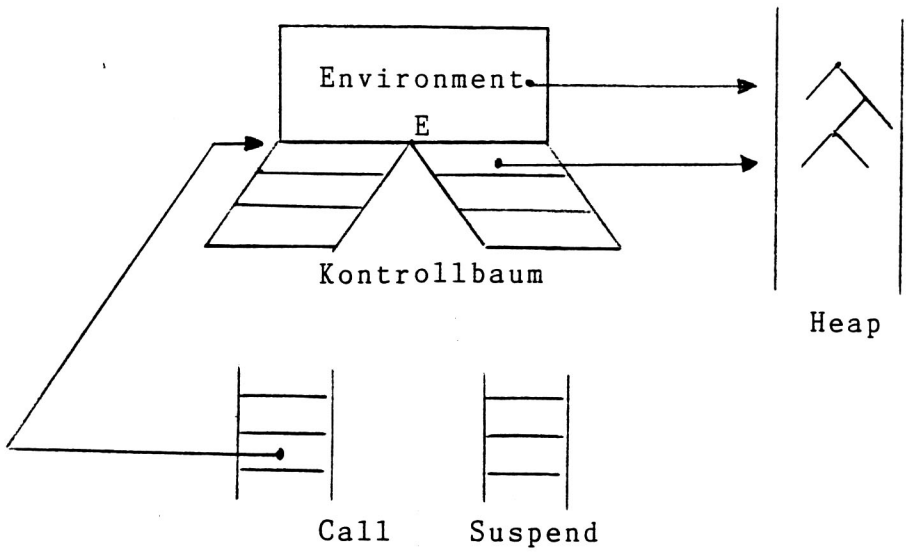
Der Ein / Ausgabe-Prozeß nimmt die Kommunikation mit der Außenwelt wahr. Er vermittelt Variablenbindungen, Teilziele und Signale für das Kommunikationsprotokoll. Der Hauptprozeß verwaltet den Programmablauf und die erzielten Resultate. Zugleich übernimmt er die Master - Rolle und kontrolliert und koordiniert die anderen Prozesse der PWAM. Der Unifikationsprozeß wird mit einem Aufruf versorgt und hat die Aufgabe, die nächste anzuwendende Programmregel im Sinne des committed-choice-Nichtdeterminismus auszuwählen. Das Kommunikationselement schließlich vermittelt intern erzeugte und von außen angelieferte Unifikationsergebnisse.

Die folgende Darstellung der Datenstrukturen und der Arbeitsweise von PWAM's beschränkt sich im wesentlichen auf die der Hauptprozesse.

6.3 Datenbereiche

Stacks

Seiner Aufgabe entsprechend verwaltet der Hauptprozeß einen Heap, Kontrollbaum und zwei Queues. Der Heap dient zur Aufnahme von Substitutionsergebnissen, der Kontrollbaum ist eine Abfolge von Environments, wobei auf Choice-Points verzichtet werden kann, und die Queues nehmen Beschreibungen suspendierter resp. zum Aufruf bereiter Literale auf. Verzweigungen im Kontrollbaum entsprechen den Body-Literalen einer Regel, die als Subgoals im Anschluß an einen ausgeführten Commitment-Operator entstehen:



Ausgangspunkt für die Anlage einer Verzweigung ist ein Aufruf $p(T_1,...,T_n)$ aus der Call- oder Suspend-Queue, dessen Terme T_i über das Environment E (vgl. Bild) zugänglich sind. Diese Informationen werden dem Unifikationsprozeß übergeben, der im Erfolgsfall Variablenbindungen für die Body-Literale $B_1,...,B_r$ der zu $p(T_1,...,T_n)$ ausgewählten Regel

(*) $p(t_1,...,t_n)$:- $G_1,...,G_m$: $B_1,...,B_r$

liefert. Diese ergeben sich aus der Input-Unifikation von $p(T_1,...,T_n)$ und $p(t_1,...,t_n)$, der Auswertung der Guards und der Output-Unifikation.

Die durch die Output-Unifikation gewonnenen Ergebnisse dienen zugleich dazu, die für $p(T_1,...,T_n)$ und für die Aufrufumgebungen davon vorliegenden Bindungen zu ergänzen. Dies hat Konsequenzen für E und bewirkt i.a., daß Bindungen an Nachbar-PWAM's zu übergeben sind. Das Bild auf S. 29 sieht Puffer im Kommunikationselement vor, welche diese Information für den Ein/Ausgabe-Prozeß wie für den Hauptprozeß bereitstellen. Von hier aus gelangen die Body-Goals und ihre Aufrufparameter zum Hauptprozeß und können dort wie im Bild auf S. 30 gespeichert werden. Zugleich ist eine Beschreibung der B_v in die Call-Queue aufzunehmen und es ist zu registrieren, welche der früher suspendierten Aufrufe aufgrund der gewonnen Information für einen weiteren Unifikationsversuch herangezogen werden sollten. Dies äußert sich dadurch, daß geeignete Kandidaten aus der Suspend-Queue in die Call-Queue übernommen werden. Einzelheiten hierzu werden unten weiter diskutiert.

Der oben skizzierte Ablauf setzt voraus, daß die innerhalb des Unifikationsprozesses ausgewählte Regel(*) zum Aufbau des eigenen Moduls heranzuziehen ist. Als zweite Möglichkeit ist zu berücksichtigen, daß (*) Startregel eines neuen Moduls ist, so daß eine neue PWAM zu generieren ist. Wir gehen unten weiter hierauf ein.

Interne Kanäle

Die Zusammenfassung von Einzelprozessen zu Modulprozessen hat zur Folge, daß ehemals externe Kanäle innerhalb des Kontrollbaums als interne Kanäle auftreten. Für ihre Anlage und insbesondere für Zwecke der Konsistenzprüfung ist die oben erwähnte Eigenschaft von Aufrufen wichtig:

Ein Aufruf $p(T_1,\ldots,T_n)$ bewirkt für die durch Unifikation ausgewählte Regel den Transfer von Input-Information von den Input-Mode-Positionen von p zu den Body-Literalen der Regel und umgekehrt den Transfer von Information in die Aufrufumgebung über die Output-Mode-Positionen des Aufrufs.

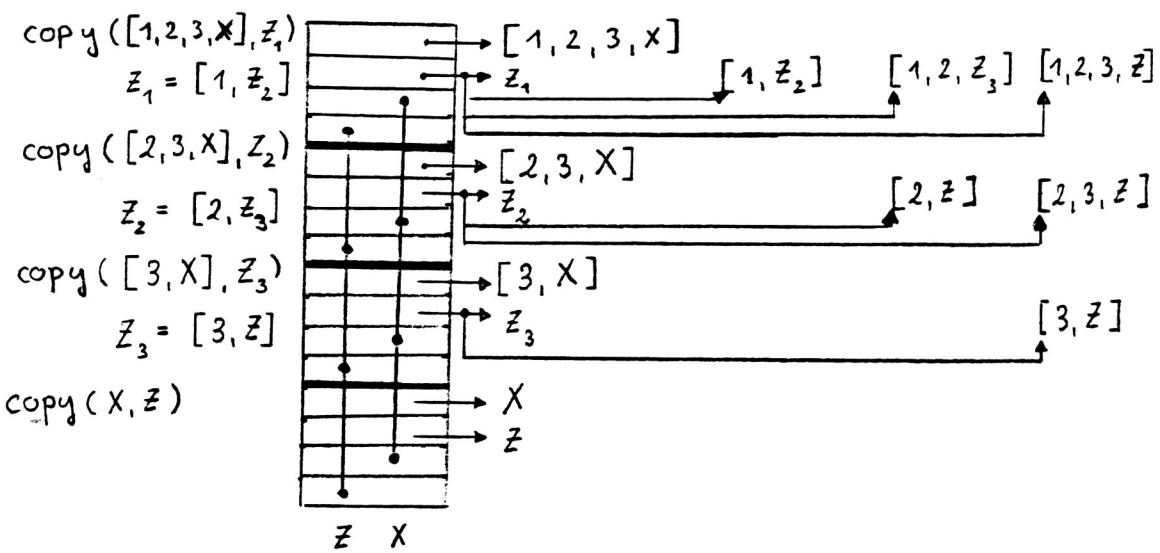
Die sich danach ergebenden Kanäle sind bidirektional und die Richtung ihrer Benutzung hängt davon ab, ob sie in Input-Mode-Positionen oder Output-Mode-Positionen weiterer Aufrufe enden.

Führt die Bearbeitung eines Aufrufs zur Auswahl der Startregel eines Moduls, dann ist eine neue PWAM zu generieren. Zur Beschreibung des Zusammenhangs der Kanäle zwischen der alten - und der neuen PWAM sind Schnittstellenbeschreibungen anzulegen, welche die jeweilige Kanalsituation charakterisieren und welche Grundlage für Konsistenztests im Anschluß an Transferoperationen sind. Wir verzichten hier auf Details und beschreiben den Aufbau interner Kanäle vielmehr anhand eines Beispiels.

copy ([u|x], [u,z]) :- copy (x,z)

mode copy (in ? , out !)

Wir betrachten copy ([1,2,3,x], z_1)



6.4 Ablauf der PWAM

Die PWAM führt Unifikationen für Elemente der Queues durch und veranlaßt den Austausch von Variablenbindungen mit Nachbar-PWAM's.

Unifikationspipeline

Die erste Teilaufgabe wird in Form einer Pipeline gelöst, welche den Hauptprozeß, das Kommunikationselement und den Unifikationsprozeß umfaßt. Im Hauptprozeß werden Aufrufe $p(T_1,\ldots,T_n)$ der Call-Queue ausgewählt und zusammen mit ihren Aufrufparametern T_i an den Unifikationsprozeß übergeben. Von diesem wird erwartet, daß er terminiert und als Ergebnis für das Kommunikationselement eine erfolgreiche Unifikation, eine Suspendierungsaussage oder eine Fehlermeldung liefert. Der Hauptprozeß reagiert entsprechend: im ersten Fall ergeben sich neue Bindungen und Subgoals, im zweiten ist eine Suspendierung zu registrieren und im dritten Fall ist der Programmlauf abzubrechen.

Ablaufkontrolle

Die Steuerung betrifft die Aufeinanderfolge intern ausgelöster Unifikationen und Schritte zum Informationsaustausch zwischen PWAM's. Ein optimaler Ablauf wird erreicht, wenn PWAM's frühzeitig generiert und derart mit Informationen beliefert werden können, daß möglichst jeder Unifikationsversuch erfolgreich zu einem Ableitungsschritt führt, d.h. wenn es gelingt, die Suspend-Queue weitgehend ungenutzt zu lassen.

Dieses ideale Verhalten wird i.a. nicht erreichbar sein: einerseits kann eine Abfolge von Unifikationsschritten gelingen und andererseits kann sein, daß erst durch die Übernahme von externer Information weitere Unifikationsschritte erfolgreich sind.

Steuerung der Pipeline

Es bietet sich an, diesbezügliche Informationen zu ermitteln und diese zusammen mit den Aufrufen auf den Queues abzulegen. Im allgemeinen steht diese Information a priori nicht zur Verfügung und kann erst durch Unifikationsversuche ermittelt werden. Allerdings sind eine Reihe von Zwischenlösungen denkbar, die mehr oder weniger

umfassenden Erfolg garantieren.

Diese zusätzliche Information zusammen mit einer Beschreibung der jeweiligen Kanalverbindungen nach außen ist Statusinformation, die für Steuerzwecke verwendbar ist, z. Bsp. so, daß mit Priorität Unifikationen durchgeführt werden, solange Aussicht auf deren erfolgreiche Durchführbarkeit besteht. Ist dies nicht mehr gegeben, dann wird bevorzugt versucht, durch Kommunikation mit den umgebenden PWAM's extern generierte Variablenbindungen zu nutzen.

Kommunikationsprotokoll

Das Verhalten der PWAM's wird weiter bestimmt durch die Auszeichnung des Hauptprozesses als Master und die Verwendung einer asynchronen Kommunikation.

Der erste Punkt besagt, daß Aufgaben wie

* Ausführen einer Unifikation
* Eingabe oder Ausgabe von Informationen

vom Master ausgelöst und quittiert werden müssen. Für eine solche Lösung spricht die Tatsache, daß die Steuerinformation im Hauptprozeß zur Verfügung steht.

Für die Implementation der Pipeline bietet sich allerdings an, dem Unifikationsprozeß eine aktive Möglichkeit zum Transfer seiner Ergebnisse über den unif-to-com-Kanal zu ermöglichen.

In der folgenden Beschreibung treten Instruktionen

* send(Info, Kanal)
* receive (Info, Kanal)

auf, welche eine asynchrone Kommunikation ermöglichen und welche die Ablage resp. Übernahme der Information in einem Puffer resp. aus einem Puffer bewirken. Der Puffer ist als Ziel resp. als Quelle durch den jeweiligen Kanal eindeutig bestimmt.

Weiter gestatten wir, daß Ein/Ausgabeprozesse und Kommunikationselemente weitgehend unabhängig und parallel zum Ablauf der Pipeline Daten ein- und ausgeben können. Es bietet sich an, das Verhalten an das der Hauptprozesse

anzupassen: Im Verlauf interner Operationen fallen i.a. Variablenbindungen für umgebende PWAM's an, die frühzeitig weiterzugeben sind. Werden andererseits externe Informationen benötigt, dann wird erwartet, daß bevorzugt Leseoperationen versucht werden.

6.5 Ablauf des Hauptprozesses

Die grundsätzliche Arbeitsweise des Hauptprozesses kann so beschrieben werden:

```
1.   seq
2.        receive (Info, inp-channel)
3.        update (Info)
4.        while not-to-be-killed do
5.              if  int-ops-priority
6.                  then
7.                       send (top (call-queue), main-to-unif)
8.                       receive (Info, com-to-main)
9.                       update (Info)
10.                 else
11.                      receive (Info, com-to-main)
12.                      update (Info) od
```

Die ersten beiden Zeilen 2.,3. werden werden zur Initialisierung erzeugter PWAM's angesprochen. Wir kommen später hierauf zurück.
Im Normalfall wird die While-Schleife durchlaufen: die PWAM existiert solange wie "not-to-be-killed" zutrifft. Diese Variable beschreibt den Status der Queues und externe Verbindungen zur Umgebung. Sie hat "true" als Wert, solange eine der beiden Queues nicht leer ist oder externe Variablenbindungen zu erwarten sind.

Die Verzweigung in Zeile 5. unterscheidet die beiden oben erwähnten Fälle: für int-ops-priority = true sind die internen Operationen mit Priorität zu bearbeiten. Erwartet wird in dieser Situation, daß die Call-Queue nichtleer ist. Damit kann das oberste Element dieser Queue an den Unifikationsprozeß übergeben werden. Dieser hat eine neue Aufgabe, und der Hauptprozeß kann in Zeile 8. Information vom Kommunikationselement anfordern.

Die vom Hauptprozeß zu leistende Datenmanipulation ist in "update (Info)" versteckt. Diese Instruktion muß entscheiden, ob die in Zeile 8. übernommene Information direkt vom Ein/Ausgabe-Prozeß stammt oder vom Unifikationsprozeß geliefert ist. Die Unterscheidung ist möglich durch eine Kennung, die nach Empfang über den io-to-com- resp. unif-to-com- Kanal zur Information hinzugefügt wird.

a) Handelt es sich um ein eigenes Unifikationsergebnis, dann hängt die Wirkung von update davon ab, ob eine Fehlermeldung , eine Suspendierungs- oder eine Erfolgsaussage gewonnen ist.

   Im letzten Fall sind die auf S.30 beschriebenen Aktionen durchzuführen, die Statusinformationen und damit die Variablen not-to-be-killed, int-ops-priority sind zu aktualisieren, und es sind ggf. Konsistenztests gewonnener Bindungen mit bereits vorhandenen durchzuführen. Dies letztere hat dann entsprechende Auswirkungen auf Schnittstellenbeschreibungen.

   Die verbleibenden Fälle führen zur Aufnahme der Information in die Suspend-Queue resp. zum Abbruch des Programmlaufs.

b) Ist eine externe Variablenbindung zu behandeln, dann führt dies i.a. zu einer Aktualisierung des Heaps und Kontrollbaums und hat wie unter a) ggf. Konsistenztests zur Folge.

Zwei Bemerkungen sind wichtig:

1) In update manifestieren sich die Konzepte der sequentiellen WAM; Programme zu update lassen sich in weiten Teilen als WAM-Assembler-Programme angeben.

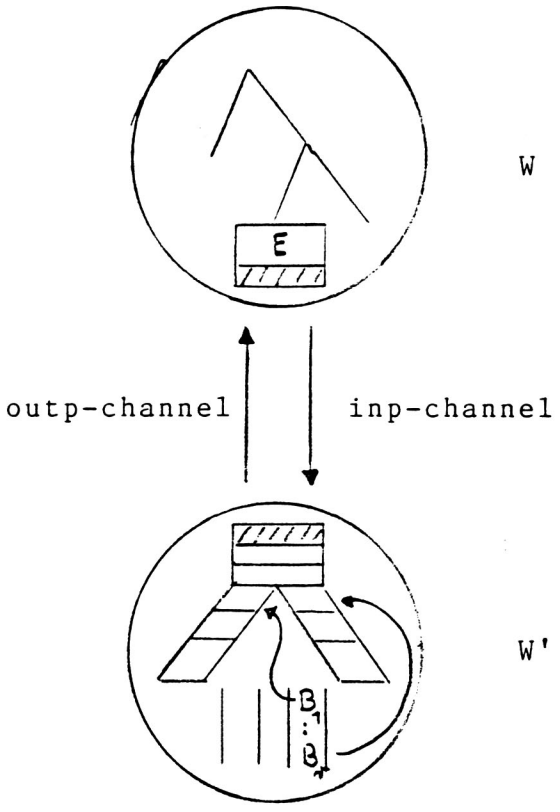
2) Der unter a) beschriebene Ablauf ist zu modifizieren für den Fall, wo die vom Unifikationsprozeß gefundene Regel Startregel eines neuen Moduls ist.

Generierung von PWAM's

Wir knüpfen an die Bemerkung 2 von oben an:
Die Entscheidung zur Erzeugung einer neuen PWAM W' fällt durch einen Aufruf von update in Zeile 9. auf S. 35. In dieser Situation sind innerhalb der generierenden PWAM W die anzuwendende Regel, die Variablenbindungen zum Aufbau von Environments und der Modul für W' bekannt.

Mit der Erzeugung von W' ist verbunden:

* Ergänzung von E (vgl. Bild auf S. 30)
* Anlage zweier Kanäle inp-channel, outp-channel
* Initialisierung der Datenstrukturen von W'



Die Ergänzung zu E betrifft die Schnittstellenbeschreibung (schraffiertes Feld im Bild), die in W anzulegen ist und nach W' zu übernehmen ist. Die Kanäle stellen Verbindungen zwischen dem Ende E des Kontrollbaums in W und seiner Wurzel in W' her. Die Initialisierung erfolgt durch den Start des Hauptprozesses zu W', der mit Hilfe von receive (Zeile 2, S.35) die in W generierten Environments und Body-Literale nach W' transferiert. Der folgende Aufruf von update bewirkt dann eine Initialisierung des Heaps, der Stacks und Queues.

7. Literatur

(Ada 83) Reference Manual for the Ada Programming Language
ANSI/MIL-STD 1815 A, DoD, Jan. 1983

(Bal 71) R.M. Balzer: PORTS - A Method for Dynamic Interprogram Communication and Job Control, Proc. AFIPS Spring Joint Comp. Conf., Atlantic City, 1971, 485-489

(Bri 75) P.Brinch Hansen: The Programming Language Concurrent PASCAL, IEEE Trans. Softw. Eng., Vol. SE-1, No.2, June 1975, 199-206

(Bar 83) I.M. Barron et al.: The Transputer, Electronics, 17. Nov.,1983

(Cla, Gre 86) K.Clark, S. Gregory: PARLOG: Parallel Programming in Logic, ACM Trans. Progr. Lang. Systems, Vol. 8, No. 1, Jan. 1986, 1-49

(Col 73) A. Colmerauer, H. Kanoui, P. Roussel, R. Pasero: Un Systeme de Communication Homme-Machine en Francais, Groupe de Recherche en Intelligence Artificelle, Université d'Aix-Marseille, 1973

(Dij 75) E.W.Dijkstra: Guarded commands, non-determinacy and a calculus for the derivation of programs, CACM 18 (8), 1975, 453-457

(Fly 72) M.J. Flynn: Some computer organizations and their effectiveness, IEEE Trans. Comp. 21 (9), 1972, 948-960

(Fre 87) B. Freisleben: Mechanismen zur Synchronisation paralleler Prozesse, Springer, Informatik Fachbericht 133, 1987

(Gab et al. 85) J. Gabriel, T. Lindholm, E.L. Lusk, R.A. Overbeek: A Tutorial on the Warren Abstract Machine for Computational Logic, Argonne National Laboratory, June 1985

(Gre 87) S. Gregory: Parallel Logic Programming in PARLOG, Add. Wesley, 1987

(Hoa 78) C.A.R. Hoare: Communicating sequential processes, CACM 21, 8, (Aug. 1978), 666-677

(INMOS) Occam Programming Manual, Prentice Hall, 1984

(Kow 74) R.A. Kowalski: Predicate Logic as a Programming Language, IFIP 74, 569-574

(Kow 79) R.A. Kowalski: Algorithm = Logic + Control, CACM, 22, 7 (July 1979), 424-436

(Rob 65) J.A. Robinson: A Machine-oriented Logic Based on the Resolution Principle, JACM, 12, 1 (Jan. 1965), 23-41

(Sha 86) E. Shapiro: Concurrent PROLOG: A Progress Report, in Fundamentals of Art. Intell. (eds.: W. Bibel, Ph. Jorrand), LNCS, Vol. 232, 1986

(Ued 85) K. Ueda: Guarded Horn Clauses, Technical Report TR-103, ICOT, Tokyo, Sept. 1985

(War 80) D.H.D. Warren: Logic Programming and Compiler Writing, Softw. Pract. and Exp., Vol. 10, 97-125 (1980)

(War 83) D.H.D. Warren: An abstract PROLOG Instruction Set, Technical Note 309, SRI Project 4776, 1983