Seminar „Parallelisierung und verteilte Ausführung funktionaler Programme“: PABC-Machine

Matthias Mann

5. Dezember 1996

Inhaltsverzeichnis

1 Einführung

Die Compilierung funktionaler Sprachen führt meist über eine Zwischensprache, die den Befehlssatz einer abstrakten Maschine bildet. Wir wollen in diesem Vortrag die *Parallel ABC machine* betrachten, die bei der Compilierung von Concurrent Clean als Abstraktion für MIMD-Rechner mit verteiltem Speicher dient. Dabei ist besonders das Konzept des *parallel graph rewriting* von Interesse, welches die theoretische Grundlage für die Funktionsweise der PABC darstellt. Das entsprechende Maschinenmodell für sequentielles Clean ist die *ABC machine*, welche wir als grundlegendes Modell einführend behandeln werden. Bei der Behandlung der PABC machine können dann die Erweiterungen zur ABC machine im Vordergrund stehen:

- *lazy copying*
- Prozeßerzeugung
- Kommunikation

2 ABC machine

Die ABC machine bildet, wie schon erwähnt, eine Zwischenstufe bei der Compilierung von Clean in Maschinencode. Wozu aber bedarf es einer solchen zusätzlichen Abstraktionsebene? Im großen und ganzen gibt es dafür fünf gute Gründe:

- Es wird ein strukturierte Ansatz gewählt. Die Performance-Verluste halten sich hierbei in so engen Grenzen, daß ein monolithischer Ansatz nicht gerechtfertigt wäre.
- Es wird ein generelles Verständnis für die Compilierung funktionaler Sprachen und dabei auftretender Probleme gewonnen, welches gleichsam als Schema genutzt werden kann.
- Man kann sich bei der Implementierung auf die wesentlichen Dinge beschränken und muß sich nicht mit maschinenspezifischen Details und Unterschieden abgeben.
- Auf diese Art konstruierte Compiler sind höchst portabel: Nur der Übersetzer von ABC Code in den Code der realen Maschine ist zu ändern. Dabei bleiben alle Optimierungen bei der Compilierung von Clean nach ABC wirksam.

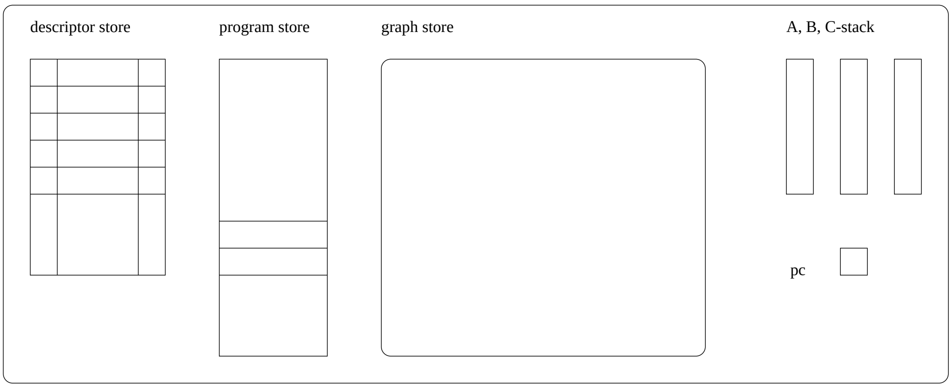

Abbildung 1: Prinzipieller Aufbau der ABC machine

- Bei der Modellbildung der ABC machine erhält man zugleich eine Defintion für einen ABC machine-Simulator/Interpreter.

Aus dem vorhergehenden Vortrag bzw. aus [PvE93] wissen wir, daß Clean eine Sprache ist, die auf *Functional Graph Rewriting Systems* (FGRS) basiert. Es wird also in diesem Abschnitt darum gehen, wie ein Graphersetzungssystem implementiert wird bzw. wie die Graphersetzung [1] auf der ABC machine nun konkret auszusehen hat.

2.1 Aufbau und Funktionsweise

Dazu beschäftigen wir uns als erstes mit dem Aufbau der ABC machine, der in Abbildung 1 schematisch zu sehen ist.

Hieraus kann man die grundlegenden Bestandteile der ABC machine erkennen; sie seien im folgenden aufgezählt:

- Im *graph store* wird der Graph bearbeitet.
- Der *descriptor store* ist eine Art Symboltabelle.
- Im *program store* steht das ABC machine-Programm. Dazu gehört natürlich auch ein Programmzähler *pc*.
- Drei Stacks geben der ABC machine ihren Namen:
  - Der *A*-Stack ist für Argumente, d.h. er speichert Zeiger auf Knoten im graph store.

[1] Bleiben wir in Zukunft doch lieber beim englischen Ausdruck „graph rewriting“.

  - Der *B*-Stack ist für Basic Values.
  - Der *C*-Stack speichert Control Informationen.

- Außerdem gibt es noch I/O-Kanäle, die aber in der Abbildung nicht zu sehen sind.

Die Architektur der ABC machine kann man im Prinzip in zwei Teile gliedern: Einen, der einer konventionellen Stackmaschine — repräsentiert durch den B-Stack — entspricht, und einem anderen bestehend aus graph store, A-Stack und descriptor store, der für das graph rewriting verantwortlich ist. Die übrigen Komponenten werden dabei von beiden Teilen genutzt.

Damit ähnelt die ABC machine stark der *G machine* [2], unterscheidet sich dann aber doch in Details. So gibt es die G machine zwar auch in Varianten mit dem optimierten basic value Stack, die Knoten des Graphen hingegen werden ganz anders dargestellt: In der G machine sind es immer zweiwertige *apply*-Knoten, wohingegen die Stelligkeit der Knoten in der ABC machine unterschiedlich sein kann.

Der Zustand — man könnte auch „die Konfiguration" sagen — der ABC machine ist eindeutig bestimmt durch die einzelnen Zustände all der oben aufgezählten Komponenten. Instruktionen der ABC machine überführen dann die Maschine von einem in einen anderen Zustand. Dies könnten wir im Prinzip folgermaßen spezifizieren [3]:

```
> state       == (astack, bstack, cstack, graphstore,
>                 descrstore, pc, programstore, io)
> instruction == state -> state
```

2.2 Der Befehlssatz der ABC machine

Dabei werden die einzelnen Komponenten nun als abstrakte Datentypen modelliert, auf denen jeweils eine Reihe von Mikro-Instruktionen als elementarste Operationen definiert werden.

2.2.1 Mikro-Instruktionen

Für jede Komponente der ABC machine seien im folgenden jeweils die Operationen kurz beschrieben.

[2] Diese wird bei der Compilierung von Haskell eingesetzt.

[3] und damit auch schon in Miranda codieren → Prototyp eines Interpreters

Graph store: Jeder Knoten, der im graph store gespeichert ist, verfügt über eine eindeutige id. *gs_get* liefert einen Knoten aus dem graph store, *gs_init* initialisiert den Speicher, *gs_newnode* erzeugt einen neuen Knoten und *gs_update* ändert einen bestehenden ab.

```
> gs_get     :: nodeid -> graphstore -> node
> gs_init    :: graphstore
> gs_newnode :: graphstore -> (graphstore, nodeid)
> gs_update  :: nodeid -> (node -> node) ->
>                          graphstore -> graphstore
```

Die Knoten im graph store enthalten nicht direkt Symbole sondern descriptor id's also Verweise in den

Descriptor store, der eine Art Symboltabelle darstellt. Es gibt wieder eine Initialisierung, diesmal *ds_init* genannt, sowie eine Mikro-Instruktion *ds_get*, die einen Symboltabelleneintrag liefert.

```
> ds_get  :: descrid -> descrstore -> descr
> ds_init :: [descr] -> descrstore
```

Die Einträge beinhalten in der Regel den symbolischen Namen, die Stelligkeit des Symbols und Information über einen speziellen Einsprungpunkt für partielle Funktionsanwendungen [4].

Program store: In den program store wird während des Bootvorgangs der Maschine eine Sequenz von ABC Befehlen geladen. Dazu dient *ps_init*, während *ps_get* einen instruction fetch darstellt.

```
> ps_get  :: instrid -> programstore -> instruction
> ps_init :: [instruction] -> programstore
```

Dazu muß Buch über die aktuelle Instruktion im program store geführt werden: Der

[4] *curried* ist das Schlagwort; aber man kann ja schlecht „gecurried“ schreiben.

Program Counter unterstützt die Operation *pc_get*, die einen Verweis auf die als nächstes abzuarbeitende Instruktion im program store liefert. Darüberhinaus kann man mit *pc_end* abfragen, ob die letzte Instruktion des Programms erreicht ist, oder mittels *pc_halt* das Programm anhalten. Mit *pc_next* wird der program counter inkrementiert und mit *pc_update* kann er explizit gesetzt werden (Sprünge). *pc_init* stellt eine Initialisierung dar.

```
> pc_init   :: pc
> pc_next   :: pc -> pc
> pc_halt   :: pc -> pc
> pc_end    :: pc -> bool
> pc_get    :: pc -> instrid
> pc_update :: instrid -> pc -> pc
```

A-Stack Der A-Stack soll Verweise auf Knoten im graph store, die Argumente oder das Ergebnis einer Reduktion darstellen, aufnehmen. Dazu gibt es wieder eine Initialisierung *as_init* und verschiedene Instruktionen, die den Stack manipulieren: *as_get* liefert ein beliebiges Element und *as_topn* die n obersten Elemente des Stacks. *as_popn* wirft n Elemente vom und *as_pushn* auf den Stack. *as_push* bringt ein Element auf den Stack und *as_update* verändert das angegebene.

```
> as_init   :: astack
> as_get    :: a_src -> astack -> nodeid
> as_topn   :: nr_args -> astack -> nodeid_seq
> as_popn   :: nr_args -> astack -> astack
> as_push   :: nodeid -> astack -> astack
> as_pushn  :: nodeid_seq -> astack -> astack
> as_update :: a_dst -> nodeid -> astack -> astack
```

B-Stack Der B-Stack dient dazu, Berechnungen auf einfachen (skalaren) Datentypen — z.B. Boolean, Integer, Real — nicht auf dem Umweg über den graph store machen zu müssen sondern direkt ausführen zu können. Dazu unterstützt der B-Stack verschiedene *get*-Operationen, wie *bs_getB*, *bs_getI*, usf. *bs_init*, *bs_popn* und *bs_update* haben eine zum A-Stack analoge Bedeutung.

Darüberhinaus gibt es noch *bs_pushB*, *bs_pushI* und *bs_copy* mit offensichtlicher Bedeutung. Außerdem gibt es Operationen zum Rechnen mit einfachen Typen, z.B. *bs_addI* zum addieren zweier Integer, die sich auf dem B-Stack befinden.

```
> bs_getB  :: b_src -> bstack -> bool
> bs_pushB :: bool -> bstack -> bstack
> bs_copy  :: b_src -> bstack -> bstack
> bs_addI  :: bstack -> bstack
```

C-Stack Auf dem C-Stack werden Kontrollinformationen gespeichert, d.h. Rücksprungadressen im Falle verschachtelter Reduktionen. Dazu werden nur wenige Operationen benötigt: *cs_init*, *cs_get*, *cs_popn* und *cs_push*, deren Bedeutung entsprechend der anderen beiden Stacks erklärt ist.

I/O-Kanäle Die Ein-/Ausgabekanäle waren in Abbildung 1 nicht aufgeführt. Sie sollen auch hier nur im Sinne eines primitiven Outputs erklärt werden: *io_init* initialisiert, *io_print* gib eine Zeichenkette aus und *io_print_symbol* den Eintrag aus dem descriptor store, der zu einem gegebenen Knoten aus dem graph store korrespondiert. Diese Operationen definieren Seiteneffekte, die aufgrund des *unique typing* in Clean direkt von dort aus aufgerufen werden können.

```
> io_init         :: io
> io_print        :: string -> io -> io
> io_print_symbol :: node -> descr -> io -> io
```

2.2.2 Maschinen-Instruktionen

Wie wir schon gesehen hatten, ist der Zustand der ABC machine eindeutig durch die Zustände der einzelnen Komponenten bestimmt und Maschinen-Instruktionen überführenn die Maschine von einem Zustand in einen anderen. Die einzelnen Komponenten haben wir über die Mikro-Instruktionen spezifiziert. Wir können nun also die Maschinen-Instruktionen mit Hilfe der Mikro-Instruktionen erklären. Das Schema bleibt dabei im Prinzip immer das gleiche: Maschinen-Instruktion angewandt auf Zustand ergibt Folgezustand, wobei die entsprechenden Komponenten durch die zugehörigen Mikro-Instruktionen abgeändert wurden. Diejenigen Befehle, die auf den Stacks operieren bzw. Ein-/Ausgaben tätigen, sind sicherlich weniger interessant — da sie eher einer konventionellen Stackmaschine ähneln — als die, die den Graphen manipulieren oder Informationen aus den Knoten heraussuchen. Daher seien nur für die letztere Gruppe einige Beispiele gegeben.

```
> create :: instruction
> create (as,bs,cs,gs,ds,pc,ps,io) =
>    (as´,bs,cs,gs´,ds,pc,ps,io)
>    where
>       as´           = as_push nodeid as
>       (gs´, nodeid) = gs_newnode gs
```

create erzeugt einen neuen Knoten im graph store und hinterläßt die id des Knoten auf dem A-Stack. Mit *fill* kann der Inhalt eines schon existierenden Knoten verändert werden. Und zwar müssen sich die Argumente für den betreffenden Knoten auf dem A-Stack befinden [5].

```
> fill :: descrid -> nr_args -> instrid -> a_dst -> instruction
> fill descr nr_args entry a_dst (as,bs,cs,gs,ds,pc,ps,io) =
>    (as´,bs,cs,gs´,ds,pc,ps,io)
>    where
>       as´    = as_popn nr_args as
>       gs´    = gs_update nodeid (n_fill descr entry args) gs
>       nodeid = as_get a_dst as
>       args   = as_topn nr_args as
```

Wir können nun schon eine Sequenz von ABC Befehlen angeben, die den Graphen (`Cons 1 Nil`) erzeugt:

```
create
create
```

[5] *n_fill* ist dabei eine von weiteren Mikro-Instruktionen, die bisher unerwähnt blieben: Es handelt sich um eine für nodes; *(n_fill descr entry args)* ist dabei vom Typ *node* → *node*.

```
fill descrid_nil 0 entry_nil 0
create
fill descrid_int_const 1 entry_int_const 0
fill descrid_cons 2 entry_cons 2
```

Dabei müßten für *descr_id...* und *entry_...* konkrete Zahlen eingesetzt werden, die den Deskriptor und den Einsprungpunkt für die Reduktion des Knotens zur *root normal form* (rnf) bestimmen. Um dies zu vermeiden und die Lesbarkeit zu erhöhen, kann man auch eine ABC Assemblersprache einführen, die einen algebraischen Datentyp darstellt, dessen Konstruktoren gerade dieselben Namen haben wie die jeweiligen Befehle.

```
Create
Create
Fill "Nil" 0 "_rnf" 0
Create
Fill 1 0
Fill "Cons" 2 "_rnf" 2
```

Dabei steht der Label „_rnf" für einen Einsprungpunkt von Knoten, die bereits in rnf sind: An jener Stelle wird gewöhnlich nur eine Returnanweisung stehen, da nichts zu tun ist. Die Namen „Nil" und „Cons" bezeichnen dann die jeweiligen Deskriptoren.

Die Veränderungen von A-Stack und graph store durch die Ausführung dieser Codesequenz ist in Abbildung 2 illustriert. Zwei Beispiele für Instruktionen, die Informationen aus den Knoten des graph store auf den A-Stack bringen oder auswerten, seien noch beschrieben. Dies ist zum einen *push_args*, welche — wie der Name schon sagt — die Argumente eines Knotens [6], der sich an Position `a_src` auf dem A-Stack befindet, auf den A-Stack lädt.

```
> push_args :: a_src -> arity -> nr_args -> instruction
> push_args a_src arity nr_args (as,bs,cs,gs,ds,pc,ps,io) =
>    (as´,bs,cs,gs,ds,pc,ps,io)
>    where
>       as´    = as_pushn args as
>       args   = n_nargs (gs_get nodeid gs) nr_args arity
>       nodeid = as_get a_src as
```

[6] *n_nargs* ist wiederum eine Mikro-Instruktion für Knoten; sie ist vom Typ $node \rightarrow nr\_args \rightarrow arity \rightarrow nodeid\_seq$.

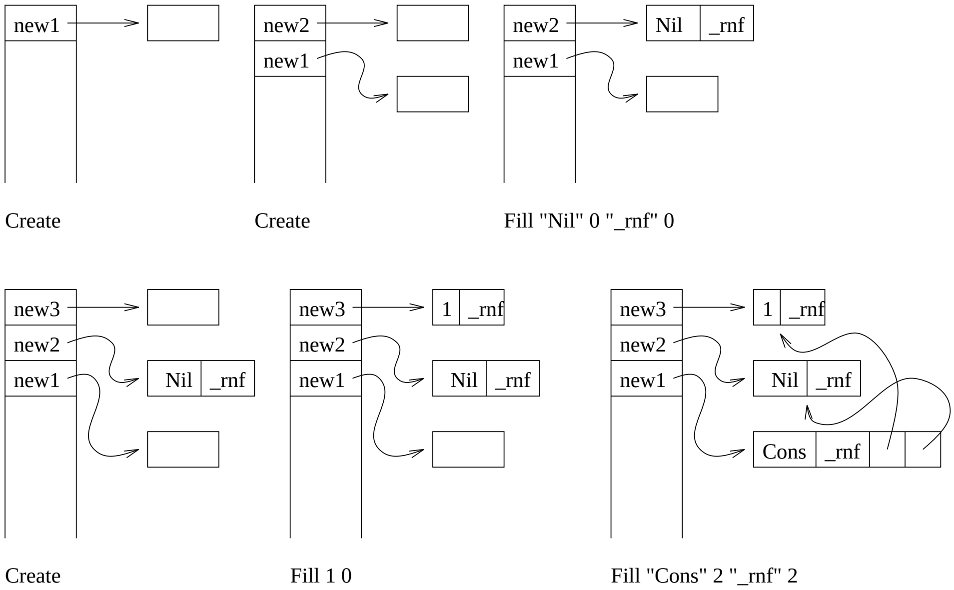

Abbildung 2: Code Beispiel

Die andere ist *eq_descr_arity*, welche prüft, ob ein Knoten, dessen Adresse an Position `a_src` auf dem A-Stack befindet, einen bestimmten Deskriptor zugeordnet und eine gegebene Stelligkeit hat. Das Ergebnis des Tests wird auf dem B-Stack hinterlassen [7].

```
> eq_descr_arity :: descrid -> arity -> a_src -> instruction
> eq_descr_arity descrid arity a_src (as,bs,cs,gs,ds,pc,ps,io)
>  = (as,bs´,cs,gs,ds,pc,ps,io)
>    where
>       bs´    = bs_pushB equal bs
>       equal  = n_eq_descrid node descrid &
>                n_eq_arity node arity
>       node   = gs_get nodeid gs
>       nodeid = as_get a_src as
```

[7] Die Bedeutung der Mikro-Instruktionen *n_eq_descrid* und *n_eq_arity* ist wohl selbsterklärend.

2.3 Compilierung von Clean

Das Ziel der ABC machine ist es ja, für jede Regel in Clean eine entsprechende Sequenz von ABC Befehlen angeben zu können, die bei Anwendung auf ein Argument den Graphen genau so bearbeitet, wie es die Regel vorschreibt. Diese Transformation ist recht interessant aber auch komplex, weil es darum geht, eine funktionale Sprache in eine imperative Maschinensprache zu übersetzen.

Dabei können einige allgemeine Optimierungen für den ABC Code durchgeführt werden. Dazu zählen z.B. daß der graph store während des rewriting nicht ständig aktualisiert wird, da solche Zugriffe teuer sind. Außerdem kann extensiver Gebrauch vom B-Stack gemacht werden, d.h. immer wenn Argumente einer Funktion strikt und einfachen Typs sind.

Wir werden uns im folgenden aber nicht mit Optimierungen sondern nur mit dem grundlegenden Modell des graph rewriting auf der ABC machine befassen.

2.3.1 Graph Rewriting mit der ABC machine

Soll nun ein Clean Programm in ein äquivalentes ABC Programm übersetzt werden, so wird für jede Alternative einer jeden Regel des Clean Programms eine Sequenz von ABC Befehlen erzeugt. Im entsprechenden Code wird dann aus Effizienzgründen dann die Suche nach dem nächsten Redex mit dem eigentlichen Rewrite vermischt. Außerdem gibt es für jede Clean Regel Code für eine Reihe von Aufgaben:

- Erzeugung eines *stack frames*, so daß die Argumente der Funktion auf dem Stack zu finden sind, strikte schon in rnf.

- Einen Einsprungpunkt für jede Alternative einer Regel mit dem Code für das pattern matching.

- Einen Einsprungpunkt, falls kein pattern gematscht werden konnte.

Das beste ist es wohl, sich dies an einem Beispiel vor Augen zu führen. Die folgende Clean Regel soll dabei die Länge einer Liste berechnen

```
:: Length !INT  ![x]     ->  INT;
   Length n     [a | b]  ->  Length (+ n 1) b;
   Length n     []       ->  n;
```

und führt auf folgenden ABC Code

```
              Descriptor
"Length"      "a_Length" 2 "Length",
              Label
"a_Length",   Repl_args 1 1,
              Jmp "e_Length",
              Label
"n_Length",   Set_entry "_cycle" 0,
              Push_args 0 2 2,
              Label
"e_Length",   Jsr_eval,
              Push_a 1,
              Jsr_eval,
              Pop_a 1,
              Label
"Length1",    Eq_descr_arity "Cons" 2 1,
              Jmp_false "Length2",
              Push_args 1 2 2,
              Push_a 1,
              Jsr_eval,
              Create,
              Create,
              Fill 1 0,
              Push_a 5,
              Jsr "+1",
              Update_a 1 5,
              Update_a 0 4,
              Pop_a 4,
              Jmp "Length1",
              Label
"Length2",    Eq_descr_arity "Nil" 0 1,
              Jmp_false "Length3",
              Fill_a 0 2,
              Pop_a 2,
              Rtn,
              Label
"Length3",    Jmp "_mismatch"
```

Gehen wir nun den ABC Code Schritt für Schritt durch. Die erste Anweisung erzeugt den Deskriptor, womit wir uns nicht näher beschäftigen wollen. Danach sind eine Reihe von Einsprungpunkten zu finden, die da wären

Apply entry: Partielle Anwendungen von Funktionen werden mit Hilfe einer internen *_AP*-Regel übersetzt. Gelangt nun eine solche Funktionsanwendung an ihr letztes Argument, so wird von der internen *_AP*-Funktion der apply entry angesprungen.

Zu diesem Zeitpunkt befinden sich die node-id des Redexes und des Knotens, der die restlichen Argumente enthält sowie das letzte Argument auf dem Stack. Durch das `Repl_args` wird die Referenz auf die restlichen Argumente durch dieselben ersetzt. Mit `Jmp "e_Length"` wird dann damit fortgefahren, die Argumente auszuwerten.

Node entry: Dieser wird immer dann angesprungen, wenn `Length` angewendet wird auf ausreichend viele Argumente und die rnf des Ausdrucks benötigt wird.

Die node-id des Redexes befindet sich dabei auf dem Stack an Position 0. Mit `Set_entry "_cycle" 0` wird `"_cycle"` in das Codefeld dieses Knotens eingetragen, um einen *cycle-in-spine-error* entdecken zu können [8]. Über `Push_args 0 2 2` werden dann die Argumente auf den Stack gebracht.

Eval args entry: Hierhin wird sowohl vom apply wie auch vom node entry gesprungen. Die node-id's der Argumente befinden sich auf dem Stack, erstes Argument zuoberst. Da beide Argumente der Funktion `Length` strikt sind, werden sie ausgewertet. `Jsr_eval` für das erste Argument und `Push_a 1, Jsr_eval, Pop_a 1` für das zweite [9].

Rule alternative entry: Für jede Alternative der Regel gibt es einen speziellen Einsprungpunkt:

- Die erste Alternative ist `"Length1"`. Hier wird mit der Anweisung `Eq_descr_arity "Cons" 2 1` getestet, ob das zweite Argument (auf Position 1 auf dem A-Stack) mit einem `Cons` (der Stelligkeit 2) gebildet worden ist. Ist dies nicht der Fall so wird mittels `Jmp_false "Length2"` zur nächsten Alternative übergegangen.

  Ist es ein `Cons`, so sieht man bei dem rekursiven Aufruf sehr schön, wie alle Informationen während des rewrite implizit auf dem A-

[8] Ein *cycle-in-spine-error* ist ein zirkulärer Bezug. Wenn nun bei der Auswertung des Redexes dessen Ergebnis selbst wieder benötigt würde, so würde nicht zu dem node entry *"n_Length"* sondern zu *"_cycle"* verzweigt, wo dann eine Fehlerbehandlung stattfände

[9] *push_a* lädt das Element von der angegeben Position noch einmal auf die Spitze des A-Stacks, *pop_a* löscht die Anzahl der angegebenen Elemente von der Spitze des A-Stacks.

Stack gespeichert werden, ohne daß der graph store ständig aktualisiert wird.

- In der zweiten Alternative `"Length2"`, wird über die Anweisung `Eq_descr_arity "Nil" 0 1` geprüft, ob das zweite Argument ein `Nil` (ohne Argumente; Stelligkeit 0) ist. Ist dies der Fall, so wird die sich auf Position 2 auf dem A-Stack befindliche Wurzel des Redexes mit dem ersten Argument — dem `n` — von Position 0 des A-Stacks überschrieben, da aus der Striktheit folgt, daß `n` schon in rnf ist. Dies geschieht über `Fill_a 0 2`. Danach werden die beiden Argumente mit `Pop_a 2` vom A-Stack entfernt und `Rtn` springt zurück. War der pattern match nicht erfolgreich so wird zu einem

Additional rule alternative entry gesprungen. Hier wird dann ein runtime error generiert.

3 PABC machine

Wie in der Einführung schon erwähnt, ist die PABC machine eine Erweiterung der ABC machine im Hinblick auf eine MIMD-Architetur. Dies führt auch zu einer höheren Komplexität der PABC machine im Gegensatz zur ABC machine, wie wir im folgenden sehen werden.

3.1 Aufbau und Funktionsweise

Die schematische Darstellung für einen Prozessor der PABC machine ist in Abbildung 3 zu sehen.

Insgesamt bestehe die PABC machine aus m Prozessoren auf denen jeweils eine Anzahl N_m von *reducer*-Prozessen laufen können. Wie wir sehen, läuft auf jedem Prozessor außerdem ein Betriebssystem (os) und ein Kommunikationsprozess (comu process). Dafür sind die I/O-Kanäle weggefallen. Ein-/Ausgaben werden jetzt über das Betriebssystem in Verbindung mit dem Kommunikationsprozess getätigt. Die ABC machine bestand bisher nur aus passiven Komponenten. Mit os und comu process sind nun zwei *aktive* Komponenten in die PABC machine eingezogen. Dies ist aus zwei Gründen nötig geworden:

- Reducer-Prozesse sollen dynamisch erzeugt werden, müssen aktiviert und blockiert werden können. Außerdem wird ein scheduler benötigt.

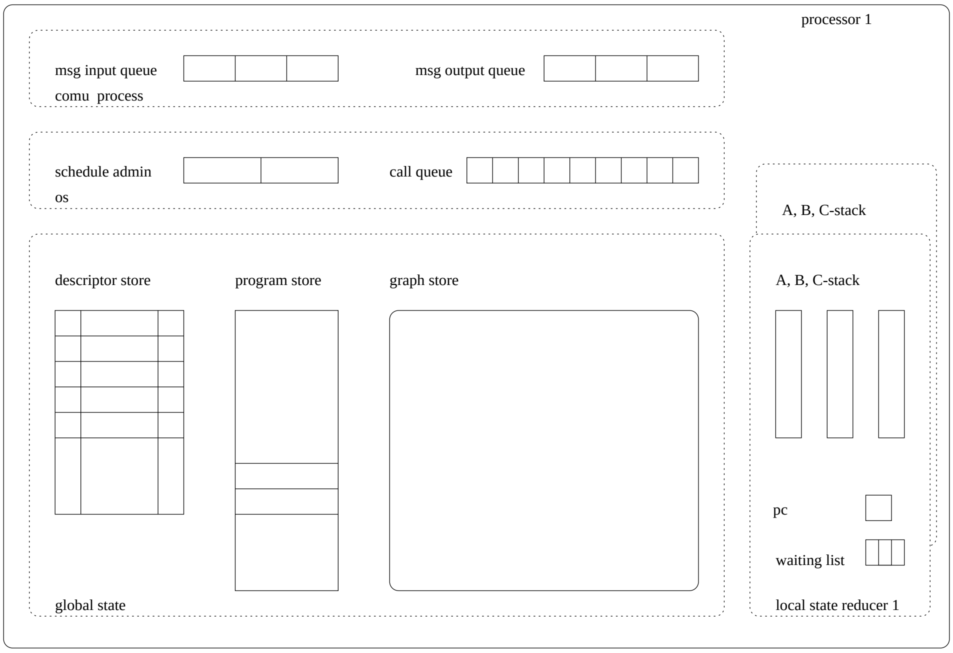

Abbildung 3: Prinzipieller Aufbau eines Prozessors der PABC machine

- Alle Prozesse sollen miteinander kommunizieren können, auch wenn sie auf verschiedenen Prozessoren laufen.

Bei der Erzeugung von reducer-Prozessen muß das Betriebbsystem einen jedem Prozess private Kopien der Stacks und des Befehlszählers bereitstellen. Darüberhinaus wird jedem reducer ein *waiting list* Register und die sogenannte *N-queue* [10] zugeordnet.

In der waiting list eines reducers werden alle anderen reducer gespeichert, die blockiert wurden und wieder aktiviert werden müssen, wenn der betreffende reducer die rnf seines Redexes hervorgebracht hat.

Die N-queue dient dazu, die PABC machine flexibel im Bezug auf die verwendbaren Reduktionsstrategien zu halten; in ihr sind alle Wurzeln von Teilgraphen zu finden, die ein reducer zu rnf auswerten muß:

- Soll ein reducer zur nf auswerten befinden sich u.U. mehrere Knoten in der N-queue.

- Ist nur einer darin, so soll der reducer eben nur diesen einen Ausdruck in rnf bringen.

[10] Diese ist nicht in der Abbildung aufgeführt. Das „N“ steht für Node.

- Ist die N-queue leer, terminiert der reducer.

Die übrigen Komponenten graph store, descriptor store und program store existieren auf jedem Proessor jeweils einmal global und werden von allen reducern gemeinsam benutzt. Dazu kommt noch ein administration store (admin in der Abb.), in dem u.a. vermerkt ist, welcher reducer-Prozess gerade bedient wird.

```
> global_state == (admin, graphstore, descrstorem programstore)
> local_state  == (astack, bstack, cstack, nqueue, pc, wlist)
```

3.1.1 Betriebssystem

Da das Betriebssystem die vollständige Kontrolle über alle Prozesse eines Prozessors hat, ist der Zustand desjenigen gleichbedeutend mit dem Zustand des Betriebssystems.

```
> prozessor         == opsys_state
> opsys_instruction == opsys_state -> opsys_state
> opsys_state       == (schedule_adm, in_msgs, out_msgs,
                         global_state, opsys_callqueue)
> schedule_adm      == ([reducer], [reducer])
> reducer           == (redid, local_state)
```

Ein Prozess kann einen Betriebssystemaufruf ausführen, indem es die entsprechende Anforderung in die `opsys_callqueue` einreiht. Sobald der Betriebssystemprozess wieder die Kontrolle erlangt, nimmt er die Anforderungen aus der queue und führt die entsprechenden Betriebssystembefehle aus. Instruktionen überführen dann das Betriebssystem von einem Zustand in einen anderen, wobei der Zustand des Betriebssystem gekennzeichnet ist durch

- Schedulding-Information: `schedule_adm` ist ein Paar aus zwei Listen von reducern; den aktiven und den blockierten,

- die ein- und ausgehenden Nachrichten, um mit dem comu process Verbindung aufnehmen zu können.

- den Zustand der globalen, gemeinsam benutzten Speicher und
- den Zustand der opsys_callqueue.

Im folgenden sind die Betriebssystembefehle kurz aufgeführt:

```
> os_suspend_reducer :: opsys_instruction
> os_schedule        :: opsys_instruction
> os_reducerprio     :: opsys_instruction

> os_send_newreducer :: instrid -> nodeid -> opsys_instruction
> os_new_reducer     :: instrid -> nodeid -> opsys_instruction
> os_stop_reducer    :: opsys_instruction
> os_halt            :: opsys_instruction

> os_send_newchannel :: procid -> nodeid -> opsys_instruction
> os_send_graph      :: nodeid -> nodeid -> opsys_instruction
> os_send_request    :: nodeid -> opsys_instruction
> os_setwait         :: nodeid -> opsys_instruction
> os_release         :: wlist -> opsys_instruction

> os_print           :: string -> opsys_instruction
```

3.1.2 Kommunikationsprozess

Der Zustand des Kommunikationsprozesses hängt nicht nur von den ein- und ausgehenden Nachrichten ab, sondern auch von der opsys_callqueue, um mit dem Betriebssystem zusammenarbeiten zu können, und vom Zustand der globalen Speicher. Hierbei ist natürlich besonders das Lesen und Schreiben des graph store von Bedeutung, wenn es um lazy copying von Graphen geht.

```
> comproc_state       == (in_msgs, out_msgs,
                          global_state, opsys_callqueue)
> comproc_instruction == comproc_state -> comproc_state
> in_msgs             == [msg_contents]
> out_msgs            == [message]
> message             == (proc_src, proc_dst, msg_contents)
```

```
> cp_accept_msg       :: msg_contents -> comproc_instruction
```

Der comu process besitzt eine einzige Instruktion `cp_accept_msg`. Diese wird wiederholt ausgeführt und sorgt dafür, daß eine Nachricht auch auf dem gewünschten Prozessor landet. Außerdem kann das Betriebssystem den Auftrag geben, Nachrichten an andere Prozessoren zu schicken, indem es diese in die `out_msgs`-Liste einfügt.

3.1.3 Reducer-Prozesse

Neue reducer-Prozesse können mit den Instruktionen `new_int_reducer` und `new_ext_reducer` gestartet werden. Der Zustand eines reducers ist bestimmt durch seinen `local_state`, den `global_state` und die `opsys_callqueue`.

```
> instruction == state -> state
> state       == (local_state, global_state, opsys_callqueue)
```

Aufgrund des veränderten Typs von `instruction` und `state` wären auch alle Maschinen-Instruktionen anzupassen, aber damit wollen wir uns nicht abgeben. Es seien im folgenden nur die neu hinzugekommenen Maschinen-Instruktionen aufgelistet:

```
> new_ext_reducer :: instrid -> a_src -> instruction
> new_int_reducer :: instrid -> a_src -> instruction
> stop_reducer    :: instruction

> newP            :: instruction
> randomP         :: instruction
> currentP        :: instruction
> neighbourP      :: instruction
> channelP        :: a_src -> instruction

> create_channel  :: instruction
> send_graph      :: a_src -> a_src -> instruction
> send_request    :: a_src -> instruction

> is_empty_n      :: instruction
```

```
> get_node        :: instruction
> set_continue    :: a_src -> instruction

> set_wait        :: a_src -> instruction
> suspend         :: instruction
> getWL           :: a_src -> instruction
> release         :: instruction
```

Außer den schon erwähnten Instruktionen für die reducer teilen sich die übrigen in vier Gruppen:

Prozessor: *newP*, *randomP*, *currentP*, *neighbourP* und *channelP* liefern allesamt Prozessor-id's.

Kommunikation: *send_graph* kopiert einen Graph lazy und *send_request* verlangt nach einem solchen.

N-queue: *is_empty_n* fragt die N-queue auf Leerheit ab, *get_node* holt einen Knoten aus der N-queue und *set_continue* speichert einen hinein.

Waiting List: *suspend* blockiert einen reducer, *getWL* holt alle blockierten reducer in die waiting list und *release* aktiviert wieder alle reducer, die sich noch in der waiting list befanden.

3.2 Interleaved Reduction auf einem Prozessor

Wir werden uns in diesem Abschnitt erst einmal mit reducer-Prozessen beschäftigen, die mittels der {I}-Annotation auf dem gleichen Prozessor erzeugt wurden. Parallele reducer werden in Abschnitt 3.3 behandelt werden. Die Instruktion `new_int_reducer` erzeugt einen neuen reducer, indem sie den Betriebssystembefehl `os_new_reducer` ausführt.

```
> new_int_reducer :: instrid -> a_src -> instruction
> new_int_reducer code a_depth ((as,bs,cs,nq,pc,wl),glob,cq) =
>    ((as,bs,cs,nq,pc,wl),glob,cq´)
>    where
>       nid = as_get a_depth as
>       cq´ = os_call cq [ os_new_reducer code nid ]
```

Dabei kann über die Angabe eines Einsprungpunktes gesteuert werden, ob der entstehende reducer zu rnf oder nf oder sonst irgendetwas reduzieren soll. Im Falle eines nf-reducers würde dann von der N-queue intensiver Gebrauch gemacht.

3.2.1 Blockieren von reducern

Um zu verhindern, daß ein reducer einen Knoten bearbeiten will, der schon längst von einem anderen bearbeitet wird, muß dieser Knoten als allererstes reserviert werden. Dafür wird der Code, der für den node entry des Knotens erzeugt wird, wie nachfolgend abgeändert:

```
            Label
"n_entry",  Set_entry "_reserve" ...,
            Push_args 0 ... ...,
            ...
```

Anstatt "_cycle" wird nun eben "_reserve" in das Codefeld der Wurzel des Redexes eingetragen. Dadurch springt ein anderer reducer, der auch diesen Knoten auswerten möchte nicht in dessen node entry, sondern zu dem zu definierende "_reserve" Einsprungpunkt.

```
            Label
"_reserve", Set_wait 0
            Suspend
            Rtn
```

Durch den `Set_wait 0` Befehl wird der aktuelle reducer in die waiting list des Knotens eingetragen. Danach wird er mit `Suspend` schlafen geschickt. Wenn er wieder aktiviert wird, hat er außer einem `Rtn` nichts zu tun, da sich der gewünschte Knoten ja jetzt schon in rnf befindet. Diese Arbeit hat ein anderer reducer getan.

3.2.2 Das defer-Attribut

Die {I}-Annotation wird ja bekanntlich folgendermaßen in die Notation eines FGRS übersetzt:

$$n:\ \{I\}\,Sym\ a_1 \ldots a_m \quad\longrightarrow\quad n:\ \{p_!\}\,Sym_d\ a_1 \ldots a_m$$

Das Sym_d bedeutet, daß der Knoten Sym mit dem defer-Attribut versehen wird. Dies wird gemacht, um zu verhindern, daß der Subgraph mit Wurzel

Sym eventuell lazy kopiert wird, obwohl er noch nicht in rnf ist. Dies könnte z.B. passieren, wenn der aktuelle reducer aus einem irgendeinem Grund die Kontrolle abgibt.

Das defer-Attribut wird in der PABC machine nun durch einen speziellen Indirektionsknoten repräsentiert. Das hat den Vorteil, daß spezieller Code erzeugt werden kann und das defer-Attribut beim Überschreiben des Redexes von selbst verschwindet. Der Code zur Auswertung eines Knotens mit dem defer-Attribut sieht folgendermaßen aus:

```
                Label
"_defer_code", Set_entry "_reserve" 1,
                Push_args 0 1 1,
                Jsr_eval,
                GetWL 1,
                Fill_a 0 1,
                Release,
                Pop_a 1,
                Rtn,
```

Ein reducer, der einen solchen Knoten auswerten möchte, obwohl er reserviert ist, muß natürlich genauso blockiert werden wie in Abschnitt 3.2.1 beschrieben.

Der spezielle Indirektionsknoten hat ja im Prinzip ein Argument — nämlich den ursprünglichen — daher wirkt `Set_entry "_reserve" 1` gerade auf den sich auf Position 1 auf dem A-Stack befindenden Indirektionsknoten. `Push_args 0 1 1` lädt dann genau den „wirklichen“ Knoten auf den A-Stack, `Jsr_eval` wertet ihn aus und `GetWL 1` holt alle wegen des Indirektionsknotens blockierten reducer in die waiting list. `Fill_a 0 1` überschreibt dann den Indirektionsknoten mit dem Ergebnis der Reduktion und `Release` gibt sogleich alle blockierten reducer wieder frei. Anschließend wird nur noch mit `Pop_a 1` das ehemalige „Argument“ des Indirektionsknotens vom Stack geräumt und mit `Rtn` zum Aufrufer zurückgekehrt.

3.3 Parallel Graph Rewriting mit der PABC machine

Mit der Instruktion `new_ext_reducer` kann ein neuer reducer erzeugt werden, der auf einem anderen Prozessor läuft. Die {P}-Annotation wird wie folgt in die Notation eines FGRS übersetzt:

$$\begin{array}{lcl} n: \{P\}\, Sym\ a_1 \ldots a_m & \longrightarrow & n: LCopy\ x, \\ & & x: \{p_!\}\, I_d\ (\{*\}\, LCopy\ y), \\ & & y: Sym\ a_1 \ldots a_m \end{array}$$

Dabei ist I die Identitätsfunktion, die hier als parallel annotierte Funktion mit dem defer-Attribut versehen wird. Durch die doppelte Anwendung von *LCopy* wird sowohl der Redex lazy auf einen anderen Prozessor kopiert als auch das Ergebnis von dort wieder lazy zurückkopiert. Dazu muß x das defer-Attribut besitzen, damit das *LCopy* x solange verschoben wird, bis x in rnf eintrifft. Auf diese Weise ist der Subgraph x *loosely connected* mit dem Rest des Graphen und es entsteht eine Art Kanal zwischen den beiden Prozessoren.

In diesem Abschnitt wird es nun darum gehen, wie solche Kanäle aufgebaut werden und *LCopy* implementiert wird. Dabei ist das Erzeugen eines Kanals zwischen zwei Prozessoren eigentlich erst der dritte Schritt infolge einer {P}-Annotation.

3.3.1 Konstruktion des Graphen

Zuerst muß sichergestellt sein, daß der graph store konsistent ist, d.h. wirklich den Graphen enthält, der bearbeitet werden soll. Denn wir hatten ja in Abschnitt 2.3.1 gesehen, daß sich während des rewrite insbesondere bei rekursiven Aufrufen einige Information implizit auf dem A-Stack befinden kann. Es muß hier also Code erzeugt werden, der diese Information wieder explizit in den graph store zurück überträgt. Dafür sind aber weder neue Instruktionen noch Konzepte erforderlich.

3.3.2 Wahl des Prozessors

Als Zweites muß ein Prozessor bestimmt werden, auf dem der parallele reducer gestartet werden soll. Dafür sind die in Abschnitt 3.1.3 angegebenen Instruktionen *newP*, *randomP*, *currentP*, *neighbourP* und *channelP* gedacht, die eine Prozessor-id auf dem B-Stack hinterlassen.

3.3.3 Erzeugung eines Kanals

Nun geht es erst daran, einen Kanal zu dem anderen Prozessor — dessen id wir ja jetzt kennen — zu öffnen. Die Instruktion `create_channel` wird hierzu benutzt.

```
> create_channel :: instruction
> create_channel ((as,bs,cs,nq,pc,wl),(ad,gs,ds,ps),cq) =
>    ((as´,bs´,cs,nq,pc,wl),(ad,gs´,ds,ps),cq´)
>    where
>       bs´       = bs_popn 1 bs
>       pid       = bs_getP 0 bs
>       (gs´,nid) = gs_newchannel pid gs
>       as´       = as_push nid as
>       cq´       = os_call [ os_send_newchannel pid nid,
>                             os_setwait nid,
>                             os_suspend_reducer ]
```

Sie erwartet die Prozessor-id auf dem B-Stack und schickt diesem Prozessor die entsprechende Nachricht unter Zuhilfenahme von Betriebssystembefehlen. Der remote Prozessor erzeugt auf die Nachricht hin einen globalen Knoten, dessen node-id er dem initiierenden Prozessor wiederum in einer Nachricht übermittelt. Dieser erstellt nun einen speziellen Indirektionsknoten, eine *channel-node*, in dem die o.g. node-id und die Prozessor-id des remote Prozessors vermerkt werden.

3.3.4 Lazy Copying

Erst jetzt kann die Instruktion `send_graph` dazu verwendet werden, einen Subgraphen, der parallel ausgewertet werden soll, über den Kanal zu dem remote Prozessor zu schicken.

```
> send_graph :: a_src -> a_src -> instruction
> send_graph graph_depth chan_depth ((as,bs,cs,nq,wl),glob,cq)
>    = ((as,bs,cs,nq,wl),glob,cq´)
>    where
>       chanid  = as_get chan_depth as
>       graphid = as_get graph_depth as
>       cq´     = os_call [ os_send_graph graphid chanid ]
```

Dabei wird der Betriebssystembefehl `os_send_graph` benutzt, der wiederum dafür verantwortlich ist, den Graphen wirklich physisch zu kopieren. Dabei sind zwei Durchläufe über den betreffenden Graphen notwendig, um die Struktur zu bewahren, d.h. gemeinsam benutzte Knoten nicht zweimal zu kopieren.

Nachdem der Graph auf den remote Prozessor geschickt wurde, wird dort mit der Instruktion `new_ext_reducer` ein reducer gestartet.

```
> new_ext_reducer :: instrid -> a_src -> instruction
> new_ext_reducer code chan_depth ((as,bs,cs,nq,wl),glob,cq) =
>    ((as,bs,cs,nq,wl),glob,cq´)
>    where
>       chanid = as_get chan_depth as
>       cq´    = os_call [ os_send_newreducer code chanid ]
```

Wie bei `new_int_reducer` kann wieder über `code` gesteuert werden, ob der reducer zu rnf oder zu nf reduzieren soll.

Irgendwann wird der initiierende reducer das Ergebnis des parallel gestarteten Prozesses benötigen. Er versucht dann, die entsprechende channel-node auszuwerten, indem er deren Code ausführt. Dieser sieht im allgemeinen folgendermaßen aus:

```
                  Label
"_channel_code",  Set_entry "_reserve" 0,
                  Send_request 0
                  Set_wait 0
                  Suspend
                  Rtn
```

Als erstes wird die channel-node also reserviert, danach wird vom remote Prozessor mit `Send_request 0` das Ergebnis verlangt. Unmittelbar danach wird der reducer mit `Set_wait 0` und `Suspend` schlafen geschickt, wie wir es schon beim Code für `"_reserve"` gesehen haben. Ist dann das Ergebnis vom remote Prozessor eingetroffen, wird der reducer wieder aktiviert und kann mit `Rtn` beenden.

```
> send_request :: a_src -> instruction
> send_request chan_depth ((as,bs,cs,nq,wl),glob,cq) =
>    ((as,bs,cs,nq,wl),glob,cq´)
>    where
>       chanid = as_get chan_depth as
```

```
>       cq´    = os_call [ os_send_request chanid,
>                          os_reducerprio ]
```

4 Realisierung der PABC machine

Die PABC machine steht auf einer relativ hohen Abstraktionsebene. Daher ist noch einige Arbeit zu tun, wenn man PABC Code in den Maschinencode einer konkreten Maschine übersetzen möchte. Die folgenden Dinge müssen vor der Realisierung der PABC machine auf einer konkreten Hardwareplattform geklärt werden:

- Wie werden die Maschinenkomponenten im einzelnen dargestellt?

  1. Der program store und der program counter sind recht einfach zu realisieren, wenn man das Programm statisch auf alle Prozessoren lädt und deren Programmzähler benutzt.
  2. Bei den Stacks bekommen wir ein Problem: Da mehrere reducer auf einem Prozessor laufen, ist es a priori nicht klar, wieviele Stacks benötigt werden und wie groß diese werden. Stacks müssen also irgendwie als dynamische Objekte im Heap verwaltet werden. Dabei kommen wieder ein paar Fragen auf:
     - Soll die Anfangsgröße der Stacks möglichst klein gewählt werden, und bei Bedarf Stacks verkettet oder eine Reallokation durchgeführt werden?
     - Stacks von reducern, die terminiert haben, müssen vom garbage collector beseitigt werden.

     Auf jeden Fall wird man den B- und C-Stack zusammenlegen, um den erforderlichen Aufwand für die Stackverwaltung einigermaßen zu reduzieren.
  3. Die Implementierung der N-queue geht dann genauso wie bei den Stacks vonstatten.
  4. Die waiting list sieht wiederum schwierig aus: Bekommen die Knoten ein zusätzliches Feld, werden alle Knoten größer. Man kann auch ein vorhandenes Feld, welches zu diesem Zeitpunkt nicht benötigt wird, gebrauchen. Eine andere Möglichkeit ist, auf die waiting list ganz zu verzichten und die reducer pollend zu betreiben.

5. Den descriptor store wird man einfach implementieren können, analog zum program store.
6. Die Komponente mit dem höchsten Abstraktionsgrad ist wohl das Betriebssystem. Hier gibt es einige Aufgaben zu lösen, sei es das Scheduling, die Kommunikation oder das lazy copying. Auch eine möglichst gleichmäßige Auslastung aller Prozessoren ist wünschenswert.

- Auch die garbage collection über mehrere Prozessoren hinweg wird Schwierigkeiten mit sich bringen. Es gibt ja schon für den einfachen Fall unzählige Varianten. Auf einer verteilten Architektur wird man wohl generell auf jedem Prozessor eine Unterscheidung in globale und lokale Knoten und garbage collections machen.

In [PvE93] wird anschließend noch ein Einblick in die Umsetzung der PABC machine auf einem Transputernetzwerk gegeben, worauf ich aber nicht mehr näher eingehen möchte.

Literatur

[PvE93] Rinus Plasmeijer, Marko van Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addison-Wesley, Workingham, 1993.