Seminar :

Parallelisierung und verteilte Ausführung funktionaler Programme

Thema :

Semantik von pH : Ein paralleler Haskelldialekt

von

Sabine Schneider

nach einer Arbeit von

S. Aditya
Arvind
J.-W. Maessen
{shail, arvind, earwig}@lcs.mit.edu
L. Augustsson
augustss@cs.chalmers.se
R. S. Nikhil
nikhil@crl.dec.com

SEMANTIK VON PH : EIN PARALLELER HASKELLDIALEKT 1

1. WAS IST PH ? 1

1.1. WARUM PH ? 1
1.2. STRUKTUR VON PH 1
1.3. AUFBAU DER ARBEIT 2

2. PARALLELE AUSWERTUNGSREIHENFOLGE 2

2.1. DIE KERNSPRACHE VON PH 2
2.2. EINE PARALLELE ABSTRAKTE MASCHINE 3
2.3. INSTRUKTIONSMENGE 5
2.4. AUSWERTUNG VON AUSDRÜCKEN 8

3. SEITENEFFEKT-OPERATIONEN 10

3.1. KERNSPRACHENERWEITERUNGEN 10
3.2. AUSWERTUNGSREGELN FÜR SEITENEFFEKT-OPERATIONEN 11

4. SEMANTIK VON M-BARRIERS 12

5. ABSCHLIEßENDE BEMERKUNGEN 13

LITERATURVERZEICHNIS 15

Semantik von pH : Ein paralleler Haskelldialekt

1. Was ist pH ?

pH [11] ist eine parallele Haskell-Programmiersprachenvariante [8] mit Erweiterungen für Schleifen, synchronisierte Seiteneffekt-Operationen und explizite Sequentialisierung. Diese Arbeit beschreibt deren operationale Semantik.

Es gibt mehrere Ansätze funktionale Sprachen parallel zu implementieren. Z.B. ist es möglich, die implizite Parallelität von Haskellprogrammen auszunutzen, indem man gleichzeitig die Argumente von strikten Operatoren auswertet. Wenn es keine Seiteneffekte gibt, verändert diese Auswertung nicht das Ergebnis. Striktheitsanalysetechniken erweitern die Möglichkeiten dieser Idee, da es grundsätzlich möglich ist, Argumente einer strikten Funktion parallel auszuwerten. Außerdem sind Anmerkungen des Programmierers möglich, welche Teilausdrücke parallel ausgewertet werden sollen**,** auch wenn der Compiler nicht zeigen kann, daß der Wert all dieser Teilausdrücke gebraucht wird. Anders als ähnliche Angaben in imperativen Sprachen können diese nur die Terminierung, nicht aber das Ergebnis beeinflussen. All diese Ansätze gehen von einer *demand-driven*-Auswertung aus : Falls keine entsprechenden Angaben gemacht werden, werden Ausdrücke nur berechnet, wenn der Wert benötigt wird.

Ein alternativer Ansatz benutzt eine *parallele* Auswertungsreihenfolge, die alle Redexe parallel reduziert, also nicht nur diese, deren Wert benötigt wird. Diese Strategie implementiert auch eine nichtstrikte Semantik, wobei die Auswertung von Redexen nicht unbestimmt verzögert wird. Einer parallelen Auswertungsstrategie folgt die Id-Programmiersprache [10], bei der jeder Redex, der nicht im Rumpf einer Bedingung oder eines Lambda-Ausdrucks steht, reduziert wird. pH folgt derselben Strategie und wird im folgenden genauer beschrieben.

pH = Haskell-Syntax und Typsystem + Id-Auswertungsstrategie + Id-Seiteneffekt-Operationen.

In der Praxis können Implementierungen von pH nicht garantieren, daß jeder Redex reduziert wird. Das bedeutet, daß manche ph-Programme kein Ergebnis liefern, obwohl traditionelle Haskellprogramme dies könnten. Aber falls pH ein ordentliches Ergebnis für ein bestimmtes Programm liefert, würde dasselbe auch vom traditionellen Haskell berechnet
.

1.1. Warum pH ?

pH ist ein Versuch, die nichtstrikte (lazy) funktionale Gemeinde (z.B. Haskell) und die Dataflow-Gemeinde (z.B. Id, Sisal [7]) zusammenzubringen. Man hofft durch eine gemeinsame Basissprache Ideen und Implementierungen leichter zu teilen und Programme frei auszutauschen. Dadurch müssen Programmierer nur eine Syntax und ein Typsystem meistern. Es gibt trotzdem einige wichtige Unterschiede zwischen pH und Haskell, die mit Haskell als Grundlage einfach zu isolieren und zu verstehen sind.

1.2. Struktur von pH

pH ist aus mehreren Teilen zusammengesetzt. pH(F) ist die rein funktionale Teilmenge von pH. Die Haupterweiterungen im Vergleich zu Haskell sind for- und while-Schleifen. Z.B. könnte man die Summe von Integerzahlen zwischen 1 und n wie folgt berechnen :

```
let sum = 0
 in for i <- [1 .. n] do
       next sum = sum + i
    finally sum
```

Schleifen sind rein funktional und können leicht in endrekursive (tail-recursive) Funktionen übersetzt werden. Die strikte (eager) Semantik von pH ermöglicht es, daß Endrekursionen effizienter implementiert werden können als bei nichtstrikten Semantiken.

pH(I) fügt I-Strukturen [4] zu pH(F) hinzu. I-Strukturen sind *single-assignment* Datenstrukturen, die eine genaue Produzent-Konsument-Synchronisation zwischen parallelen Tasks erlauben. pH(I) bewahrt Determiniertheit, d.h. *Konfluenz*, obwohl die Sprache nicht mehr referentiell transparent ist.

pH(M) fügt M-Strukturen [6] zu pH(I) hinzu. M-Strukturen sind *multiple-assignment* Datenstrukturen, die eine gegenseitige Ausschluß-Synchronisation erlauben. Sie führen Seiteneffekte und Nichtdeterminiertheit ein. pH(M) bietet außerdem die Möglichkeit, Aussagen für eine sequentielle Ausführung zu gruppieren, wobei Schranken (*barriers*) benutzt werden, um ungewollte Effekte zwischen Seiteneffekt-Operationen zu vermeiden.

Das gesamte pH ist dasselbe wie pH(M).

1.3. Aufbau der Arbeit

Der Rest dieser Arbeit konzentriert sich auf die Teile der Semantik, in denen sich pH von Haskell unterscheidet. Kapitel 2 beschreibt die parallele Auswertungsreihenfolge von pH im Kontext einer Kernsprache und ihres multithreaded Interpreters. In Kapitel 3 wird die Kernsprache um I- und M-Struktur-Operationen erweitert. Kapitel 4 definiert die Semantik von barriers, indem sie in die Kernsprache übersetzt werden, wobei explizite Terminierungssignale von Seiteneffekt-Operationen benutzt werden. Am Ende schließt Kapitel 5 mit einigen Bemerkungen zu aktuellen Implementierungen ab.

2. Parallele Auswertungsreihenfolge

pH folgt einer *strikten* Auswertungsreihenfolge : Alle Tasks laufen parallel, nur durch die Datenabhängigkeiten untereinander begrenzt. Diese Strategie erzielt automatisch große Parallelität sowohl innerhalb als auch zwischen Prozeduren. Das ist anders als bei *nichtstrikten* Auswertungsstrategien wie z.B. bei Haskell : nur diese Tasks werden ausgewertet, die benötigt werden, um das Ergebnis zu berechnen. Diese Strategie zwängt der umfassenden Berechnung einen sequentiellen Constraint auf, obwohl die exakte Anordnung von Tasks dynamisch entschieden wird.

In diesem Kapitel wird das strikte Auswertungsmodell von pH beschrieben, wobei ein paralleler Interpreter für die Kernsprache von pH erklärt wird. Dieser Interpreter basiert auf einer parallelen abstrakten Maschine, die an die G-Maschine und ihre spätere Variante erinnert [9, 5, 12]. Zuerst wird die Kernsprache beschrieben (2.1.), dann die parallele abstrakte Maschine (2.2.) und die Menge ihrer Anweisungen (2.3.). Anschließend folgt eine Beschreibung des Interpreters (2.4.).

2.1. Die Kernsprache von pH

Die abstrakte Syntax der Kernsprache wird in Abb. 1 gezeigt. Jedes Zwischenergebnis eines komplexen Ausdrucks wird explizit durch einen Bezeichner benannt. Dies ist bequem, um gemeinsame Teilausdrücke darzustellen und zu bewahren [3]. Um die Beschreibung der Auswertungsregeln hier zu vereinfachen, werden keine Konstanten als Argumente von primitiven Funktionen erlaubt.

Außerhalb der normalen arithmetischen Funktionalität bietet die Kernsprache Funktionen für die Konstruktion und Selektion von Elementen einer Liste und *n*-dimensionalen Tupeln. Die Sprache erlaubt außerdem *n*-dimensionale case-Ausdrücke; verschachtelte Lambda-Ausdrücke, die freie Bezeichner enthalten können; einen generellen Funktionsanwendungsoperator ap und eine parallele Block-Konstruktion, die lexikographisches scoping kontrolliert und präzises gemeinsames Benutzen von Teilausdruckswerten ermöglicht.

c	$\in$	Constant
$f, t, x, y, z, ...$	$\in$	Identifier
$SE, X, Y, Z, ...$	$\in$	Simple Expression
E	$\in$	Expression
PF^n	$\in$	Primitive Function with n arguments
CF^n	$\in$	Data Constructor with n arguments
S	$\in$	Statement
Constant	::=	*Integer* | *Float* | *Boolean* | Nil
SE	::=	Identifier | Constant
PF^1	::=	hd | tl | select_k
PF^2	::=	+ | - | ... | < | > |...
CF^2	::=	Cons
CF^n	::=	make_n_tuple
E	::=	SE | $PF^n(x_1, ..., x_n)$ | $CF^n(x_1, ..., x_n)$ | $\lambda x.E$
		| case($x, E_1, ..., E_n$) | ap(f, z) | sap(f, z) | Block
Block	::=	$\{S$ in $x\}$
S	::=	ε | $x = E$ | $S_1 ; ... ; S_n$
Program	::=	Block

Abb. 1 : The kernel pH language

Die Bindungsreihenfolge in einem Block ist unwichtig. Der Bezeichner, der dem Keyword „in“ in einem Blockausdruck folgt, bezeichnet das Resultat des Blocks.

2.2. Eine parallele abstrakte Maschine

Die parallele abstrakte Maschine besteht aus einer Anzahl von sequentiellen Prozessoren, die an einen globalen shared memory angeschlossen sind. Die wichtigen Eigenschaften dieser Maschine werden in Abb. 2 beschrieben. Jeder Speicherplatz kann einen Wert oder eine Liste von verzögerten Threads enthalten und ist mit ihrem Zustand ausgezeichnet. Eine volle (full) Zelle enthält einen Wert. Eine neu allokierte Zelle wird mit defer bezeichnet und als leer initialisiert. Ein *Wert* ist entweder eine Konstante, eine allokierte Speicherzelle oder eine ausgewertete Funktion. Der globale Speicher enthält drei Typen von Inhalten : Heapspeicher, Aktivierungsframe und eine *Arbeitsqueue*, die eine Liste von *Threads* ist (d.h. (*Code, Umgebung*)-Paare).

Der Zustand des Prozessors besteht aus Code, Umgebung und einem *Akkumulator*, der einen Wert enthalten kann. Jeder untätige Prozessor kann Arbeit bekommen, indem er einen Thread von der Arbeitsqueue nimmt und den Code und die Umgebung des Threads lädt. Anfangs ist die Maschine leer. Ein Programm E wird gestartet, indem der folgende Code auf einem Prozessor in der leeren Umgebung ausgeführt wird :

[eval(E), print, schedule]

Die eval-Instruktion interpretiert einen Ausdruck, indem dieser in einer gegebenen Umgebung strukturell zerlegt wird. Umgebungen bilden Programmbezeichner auf Frameplätze ab und sind stackähnlich strukturiert. eval eines Blocks allokiert einen neuen *activation frame* im globalen Speicher, um die Werte der gebundenen Bezeichner des Blocks zu speichern. Es kreiert außerdem einen neuen *environment frame* für diese Bezeichner und pusht ihn auf die Prozessorumgebung. Die gebundenen Bezeichner des Blocks zeigen auf Plätze im neu aktivierten activation frame. Neue environment frames (aber nicht activation frames) werden außerdem anstelle von Lambda-Ausdrücken und Funktionsanwendungen kreiert. Die Aufteilung der Berechnung wird durch die activation frame-Plätze erreicht : alle Referenzen führen zu demselben activation frame-Platz. Die Trennung von environment frames und activation frames erlaubt es, Umgebungen frei zu kopieren und deshalb ausgewertete Funktionen genauso wie einfache Werte zu behandeln.

Bei jedem Schritt führt der Prozessor die Instruktion am Kopf der aktuellen Codesequenz aus, wobei der Pro-

i	$\in$	Instruction		
l	$\in$	Location		
v	$\in$	Value	::=	Constant | Location | $\langle \lambda x.E, p \rangle$
p	$\in$	Environment Frame	=	Identifier $\rightarrow$ Location
σ	$\in$	Store	=	Location $\rightarrow$ {$\langle$ full v $\rangle$ | $\langle$ defer δs $\rangle$}
w	$\in$	Thread	=	Code x Environment
δ	$\in$	Suspension	=	Thread
δs	$\in$	Suspensions	=	*List* (Thread)
ws	$\in$	Work Queue	=	*List* (Thread)
		Accumulator	=	Value
is	$\in$	Code	=	*List* (Instruction)
ps	$\in$	Environment	=	*List* (Environment Frame)
		Processor	=	Accumulator x Code x Environment
		Machine	=	*List* (Processor) x Work Queue x Store

State	Processor			Global Memory	
	Accumulator	Code	Environment	Work Queue	Store
Initial (starting proc.)	-	is_0	[ ]	[ ]	{ }
Initital (other proc.)	-	[ ]	[ ]	[ ]	{ }
Final (all proc.)	-	[ ]	[ ]	[ ]	σ
Error (any Proc.)	storerr	is	ps	ws	σ

Where is_0 = [eval(Program), print, schedule]

Abb. 2 : Dynamic entities used by the abstract machine

zessorzustand verändert wird. Die Instruktionsmenge des Prozessors enthält Instruktionen zum Laden und Speichern des Akkumulators, arithmetische und logische Operationen auf Speicherplätzen und Verzögern der aktuellen Threads da Werte fehlen. Das ist im Gegensatz zu nicht-strikten funktionalen Sprachen, wo unausgewertete Plätze *thunks* beinhalten, die den Wert bei Anfrage berechnen. Typischerweise wird eine Instruktion von der Codesequenz gepopt, wenn sie ausgeführt wurde, aber manche Instruktionen können neue Instruktionen zur Codesequenz hinzufügen. Dies ist der primäre Weg den Kontrollfluß zu ändern. Außerdem gibt es Instruktionen, um environment frames zu pushen / popen und um Threads von der Arbeitsqueue zu enqueuen / dequeuen.

Die Arbeitsqueue ist eine FIFO-Queue, um faires scheduling zu garantieren. Diese Queue muß atomisch (atomically) manipuliert werden. Es ist einfach, die Maschine so zu ändern, daß jeder Prozessor seine eigene Arbeitsqueue hat, die als FIFO realisiert ist. Falls ein Prozessor untätig wird, kann er einen Thread vom Ende der Arbeitsqueue eines anderen Prozessors nehmen. So lange kein Thread auf irgendeinem Prozessor willkürlich verzögert wird, wird diese Strategie faires scheduling implementieren, während Zusammenstöße aufgrund der atomischen Manipulation beträchtlich reduziert werden.

Der Gesamtzustand der parallelen Maschine wird durch die Zustände aller Prozessoren, des globalen Speichers und der Arbeitsqueue beschrieben. Abb. 2 zeigt außerdem den Start-, End- und die Fehlerzustände der Prozessoren der Maschine. Die Maschine beginnt, indem sie eine Startsequenz von Instruktionen auf einem Prozessor, der das Programm auswertet, schedulet, druckt das Resultat, das im Akkumulator ist und schedulet dann den Rest in der Arbeitsqueue. Dies betont die nicht-hyperstrikte Natur dieses Auswertungsmodells : Das Programm kann ein Ergebnis drucken bevor es endgültig terminiert. Die Maschine hält, wenn die Codesequenz und die Arbeitsqueue leer sind. Die Maschine ist *deadlocked*, wenn es noch Berechnungen im Speicher gibt aber keine Arbeit mehr. Wenn es keine Berechnungen mehr gibt, hat die Maschine *normal terminiert*. Ein Fehler kann während der Ausführung entstehen, weil es außergewöhnliche Bedingungen gab, z.B. type mismatch oder einen arithmetischen Overflow. Wenn ein Versuch unternommen wird, in einen Speicherplatz zu schreiben, der schon full ist, erreicht die gesamte Maschine einen Fehlerzustand, wie in Abb. 2 zu sehen ist.

eval(*E*)	::	Evaluate *E* in the current environment and leave the result in the accumulator
touch(*l*)	::	Check if the location *l* is full, if not suspend the current thread
rtouch(*l*)	::	Check if the location *l* is full, if not suspend the current thread including the rtouch instruction. (Thus, rtouch can be retried when the thread is activated again).
load(*l*)	::	Load the accumulator with the value present in the location *l* (the location must be full)
take(*l*)	::	Load the accumulator with the value present in the location *l* and mark the location empty (the location must be full)
loadi_*k*(*l*)	::	Load the accumulator with the value in the location at an offset *k* from the location pointed to by *l*, or suspend the thread if the location is empty
store(*l*)	::	Store the value in the accumulator in the location *l* and reactivate any suspensions
storerr(*l*)	::	Generate an error if the location *l* is full
$PF^n(l_1, ..., l_n)$	::	Apply strict primitive operator PF^n (such as +) to the values at locations l_1, ..., l_n (the locations must be full)
switch(l_x, is_1, ..., is_n)	::	Switch to the branch indexed by the value at location l_x
pushenv(*p*)	::	Push the environment frame *p* onto the environment stack
popenv	::	Pop the top-level environment frame from the environment stack
schedule	::	Schedule the next thread from the work queue
print	::	Print the accumulator contents

Abb. 3 : The instruction set of the parallel abstract machine

2.3. Instruktionsmenge

Abb. 3 zeigt die Instruktionen, die von dem Interpreter benutzt werden. Die Semantik dieser Instruktionen wird in Form von Zustandsübergängen gezeigt, wobei ein 5-Tupel (*v, is, ps, ws*, σ) benutzt wird. Die ersten 3 Elemente dieses Tupels beziehen sich auf den Akkumulator : Wert *v*, Codesequenz *is* und die Umgebung *ps* auf dem Prozessor, wo die Instruktion ausgeführt wird. Die letzten beiden Elemente des Tupels sind die Arbeitsqueue *ws* und der dynamische Speicher σ, die von allen Prozessoren geteilt werden.

Alle Instruktionen, die den globalen Speicher verändern, müssen atomisch ausgeführt werden. Das bedeutet : während ein Prozessor eine Speicherzelle leert oder verändert, darf kein anderer Prozessor auf diese Zelle zugreifen. Transaktionen an verschiedenen Zellen können parallel stattfinden. Falls eine Instruktionssequenz *is* atomisch ausgeführt werden muß, wird dies durch *atomic*(*is*) angezeigt.

Touch-Instruktionen

Die touch-Instruktion testet den Zustand der Zelle. Wenn der Zustand full ist, führt sie nichts aus. Ansonsten (Zustand ist defer) wird die Codesequenz, die der aktuellen Instruktion folgt, ab dieser Zelle mit der aktuellen Umgebung angehängt. Die rtouch-Instruktion ist ähnlich, nur daß die touch-Operation wiederholt wird.

touch :

$$\frac{\langle _ \quad touch(l) : is \quad ps \quad ws \quad \sigma[l \mapsto \langle full\ v \rangle] \rangle}{\langle _ \quad is \quad ps \quad ws \quad \sigma \rangle}$$

$$\frac{\langle _ \quad touch(l) : is \quad ps \quad ws \quad \sigma[l \mapsto \langle defer\ \delta\ s \rangle] \rangle}{\langle _ \quad [schedule] \quad ps \quad ws \quad \sigma[l \mapsto \langle defer(is, ps) : \delta\ s \rangle] \rangle}$$

rtouch :

$$\frac{\langle _ \quad rtouch(l):is \quad ps \quad ws \quad \sigma[l \mapsto \langle full\ v \rangle] \rangle}{\langle _ \quad is \quad ps \quad ws \quad \sigma \rangle}$$

$$\frac{\langle _ \quad rtouch(l):is \quad ps \quad ws \quad \sigma[l \mapsto \langle defer\ \delta\ s \rangle] \rangle}{\langle _ \quad [schedule] \quad ps \quad ws \quad \sigma[l \mapsto \langle defer(rtouch(l):is, ps):\delta\ s \rangle] \rangle}$$

Diese Instruktionen starten nicht die Auswertung der Ausdrücke in den Speicherzellen, die getoucht werden. In nicht-strikter Auswertung ist dieser Ausdruck immer bekannt und ein thunk kann dafür in dieser Zelle gespeichert werden. In diesem Fall können die oben genannten Instruktionen so verallgemeinert werden, daß sie den thunk in die Arbeitsqueue legen.

Lade- und Speicherinstruktionen

Die load-Instruktion lädt den Inhalt bestimmter Speicherzellen in den Akkumulator. Die Zelle muß full sein. Die take-Instruktion ist ähnlich zu load, aber sie leert die Speicherzelle.

load :

$$\frac{\langle _ \quad load(l):is \quad ps \quad ws \quad \sigma[l \mapsto \langle full\ v \rangle] \rangle}{\langle v \quad is \quad ps \quad ws \quad \sigma \rangle}$$

take :

$$\frac{\langle _ \quad take(l):is \quad ps \quad ws \quad \sigma[l \mapsto \langle full\ v \rangle] \rangle}{\langle v \quad is \quad ps \quad ws \quad \sigma[l \mapsto \langle defer[\ ] \rangle] \rangle}$$

loadi_*k* ist eine indizierte, indirekte load-Instruktion. Sie toucht zuerst eine indizierte Speicherzelle und lädt dann deren Wert in den Akkumulator.

loadi_*k* :

$$\frac{\langle _ \quad loadi_k(l_x):is \quad ps \quad ws \quad \sigma[l_x \mapsto \langle full\ l \rangle] \rangle}{\langle _ \quad touch(l+k):load(l+k):is \quad ps \quad ws \quad \sigma \rangle}$$

Die store-Instruktion speichert den Inhalt des Akkumulators in der angegebenen Speicherzelle. Außerdem reaktiviert sie jeden Aufschub in der Zelle, indem sie diesen in die Arbeitsqueue legt.

store :

$$\frac{\langle v \quad store(l):is \quad ps \quad ws \quad \sigma[l \mapsto \langle defer\ \delta\ s \rangle] \rangle}{\langle v \quad is \quad ps \quad ws{+}{+}\delta\ s \quad \sigma[l \mapsto \langle full\ v \rangle] \rangle}$$

Speichern in eine bereits volle Speicherzelle wird als ein Fehler betrachtet. Die storerr-Instruktion testet den Zustand einer Zelle und produziert einen Fehler, wenn sie schon full ist. Diese Instruktion kann einer store-Instruktion vorausgehen um mehrfaches Speichern an derselben Stelle zu vermeiden.

storerr :

$$\frac{\langle _ \quad storerr(l):is \quad ps \quad ws \quad \sigma[l \mapsto \langle full\ v \rangle] \rangle}{\langle storerr \quad is \quad ps \quad ws \quad \sigma \rangle}$$

$$\frac{\langle _ \quad storerr(l):is \quad ps \quad ws \quad \sigma[l \mapsto \langle defer\ \delta\ s\rangle]\rangle}{\langle _ \quad is \quad ps \quad ws \quad \sigma\rangle}$$

Arithmetische und logische Instruktionen

Die Standardarithmetik und -logik sind klar. Jede Instruktion legt das Ergebnis in den Akkumulator.

+ :

$$\frac{\langle _ \quad +(l_1,l_2):is \quad ps \quad ws \quad \sigma[l_1 \mapsto \langle full\ \underline{m}\rangle, l_2 \mapsto \langle full\ \underline{n}\rangle]\rangle}{\langle \underline{m+n} \quad is \quad ps \quad ws \quad \sigma\rangle}$$

Switch-Instruktion

Die switch-Instruktion wählt einen ihrer Äste basierend auf dem Wert in der Zelle, die durch den Dispatch-bezeichner bestimmt wird.

switch ($1 \le m \le n$) :

$$\frac{\langle _ \quad switch(l_x, is_1, \ldots, is_n):is \quad ps \quad ws \quad \sigma[l_x \mapsto \langle full\ \underline{m}\rangle]\rangle}{\langle _ \quad is_m ++ is \quad ps \quad ws \quad \sigma\rangle}$$

Instruktionen zur Manipulation der Umgebung

Die pushenv-Instruktion pusht den gegebenen Umgebungsframe auf den lokalen Umgebungsstack eines Prozessors, während die popenv-Instruktion einen Frame vom Top nimmt. Der Inhalt des Akkumulators wird davon nicht berührt.

pushenv :

$$\frac{\langle _ \quad pushenv(p):is \quad ps \quad ws \quad \sigma\rangle}{\langle _ \quad is \quad p:ps \quad ws \quad \sigma\rangle}$$

popenv :

$$\frac{\langle v \quad popenv:is \quad p:ps \quad ws \quad \sigma\rangle}{\langle v \quad is \quad ps \quad ws \quad \sigma\rangle}$$

Thread scheduling Instruktionen

Die Ausführung einer schedule-Instruktion auf einem Prozessor bedeutet entweder die Terminierung oder die Verzögerung eines Threads. Der Prozessor schedulet einen neuen Thread vom Kopf der Arbeitsqueue oder wird *untätig* falls die Arbeitsqueue leer ist. Die Maschine hält, wenn alle Prozessoren untätig sind.

schedule :

$$\frac{\langle _ \quad [schedule] \quad ps \quad (is', ps'):ws \quad \sigma\rangle}{\langle _ \quad is' \quad ps' \quad ws \quad \sigma\rangle}$$

$$\frac{\langle _ \quad [schedule] \quad ps \quad [\,] \quad \sigma\rangle}{\langle _ \quad [\,] \quad ps \quad [\,] \quad \sigma\rangle}$$

2.4. Auswertung von Ausdrücken

In diesem Kapitel wird die eval-Instruktion, die benutzt wird, um Ausdrücke zu zerlegen und auszuwerten, erklärt. Der Wert des Ausdrucks wird in den Akkumulator gelegt.

Auswertung einer Konstanten lädt diese Konstante einfach in den Akkumulator :

eval Konstante :

$$\frac{\langle _ \quad eval(c):is \quad ps \quad ws \quad \sigma \rangle}{\langle c \quad is \quad ps \quad ws \quad \sigma \rangle}$$

Die Auswertung eines Bezeichners toucht diesen zuerst und lädt dann dessen Wert in den Akkumulator :

eval Bezeichner :

$$\frac{\langle _ \quad eval(x):is \quad ps \quad ws \quad \sigma \rangle}{\langle _ \quad touch(ps(x)):load(ps(x)):is \quad ps \quad ws \quad \sigma \rangle}$$

Im Falle einer strikten primitiven Funktion werden alle Argumente getoucht, bevor die primitive Anwendung ausgeführt wird.

eval strikte PF :

$$\frac{\langle _ \quad eval(PF^n(x_1,\ldots,x_n)):is \quad ps \quad ws \quad \sigma \rangle}{\langle _ \quad touch(ps(x_1)):\ldots:touch(ps(x_n)):PF^n(ps(x_1),\ldots,ps(x_n)):is \quad ps \quad ws \quad \sigma \rangle}$$

Alle datenselektierenden Funktionen wie hd, tl und select_*k* werden direkt durch die indizierte indirekte load-Instruktion loadi_*k* implementiert.

Der nicht strikte Konstruktor Cons allokiert ein Paar von freien Speicherzellen im Speicher und gibt den Zeiger auf die erste Zelle als Wert zurück. Außerdem pusht er Arbeit in die Arbeitsqueue, um die Argumente von Cons auszuwerten und die Speicherzellen strikt zu füllen. Der *n*-dimensionale Tupel-Konstruktor make_*n*_tuple operiert genauso. Bei nicht-strikter Auswertung würden thunks in diesen Speicherzellen gespeichert, die bei Bedarf ausgewertet würden.

eval cons :

$$\frac{\langle _ \quad eval(cons(x_1,x_2)):is \quad ps \quad ws \quad \sigma \rangle}{\langle l \quad is \quad ps \quad ws ++ [w_1,w_2] \quad \sigma' \rangle}$$

where

$$\sigma' = \sigma + \{ l \mapsto \langle defer[\ ] \rangle, (l+1) \mapsto \langle defer[\ ] \rangle \}$$

$$w_1 = ([eval(x_1), store(l), schedule], ps)$$

$$w_2 = ([eval(x_2), store(l+1), schedule], ps)$$

λ-Ausdrücke werden zu einem Schlußwert ausgewertet. Dieser Schluß zeichnet die Plätze für die freien Bezeichner des λ-Rumpfes in einem neuen environment frame auf.

eval λ-Ausdrücke :

$$\frac{\langle _ \quad eval(\lambda\, x.E):is \quad ps \quad ws \quad \sigma \rangle}{\langle \langle \lambda\, x.E, p \rangle \quad is \quad ps \quad ws \quad \sigma \rangle}$$

where $y_1, \ldots, y_m = FV(\lambda\, x.E)$
$p = \{y_1 \mapsto ps(y_1), \ldots, y_m \mapsto ps(y_m)\}$

Eine Anwendung einer Funktion wird ausgewertet, indem zuerst die Funktion getoucht wird und dann der resultierende Schluß auf das Argument angewendet wird. Das Funktionsargument wird vor der Anwendung getoucht, falls die Funktionsanwendung strikt ist (sap).

eval ap :

$$\frac{\langle_ \quad eval(ap(f,z)) : is \quad ps \quad ws \quad \sigma\rangle}{\langle_ \quad touch(ps(f)) : ap(ps(f), ps(z)) : is \quad ps \quad ws \quad \sigma\rangle}$$

eval sap :

$$\frac{\langle_ \quad eval(sap(f,z)) : is \quad ps \quad ws \quad \sigma\rangle}{\langle_ \quad touch(ps(f)) : touch(ps(z)) : ap(ps(f), ps(z)) : is \quad ps \quad ws \quad \sigma\rangle}$$

Falls ein Schluß gegeben ist, erweitert die ap-Instruktion dessen environment frame mit der gegebenen Argumentspeicherzelle und startet die Auswertung des Funktionsrumpfes nachdem der neue environment frame auf den aktuellen gepusht ist. Die alte Umgebung wird, nachdem die Auswertung des Rumpfes ein Ergebnis im Akkumulator erzeugt hat, wieder gespeichert.

ap :

$$\frac{\langle_ \quad ap(l_f, l_z) : is \quad ps \quad ws \quad \sigma[l_f \mapsto \langle full\langle \lambda\, x.E, p\rangle\rangle]\rangle}{\langle_ \quad pushenv(p') : eval(E) : popenv : is \quad ps \quad ws \quad \sigma\rangle}$$

where $p' = p + \{x \mapsto l_z\}$

Bei der Auswertung des case-Ausdrucks wird zuerst der Dispatchausdruck getoucht. Dies muß eine Integerzahl innerhalb des Dispatchradius, der benutzt wird, um den entsprechenden Ast auszuwählen, ergeben.

eval case :

$$\frac{\langle_ \quad eval(case(x, E_1, \ldots, E_n)) : is \quad ps \quad ws \quad \sigma\rangle}{\langle_ \quad touch(ps(x)) : switch(ps(x), [eval(E_1)], \ldots, [eval(E_n)]) : is \quad ps \quad ws \quad \sigma\rangle}$$

Die Auswertung eines Blockausdrucks allokiert einen neuen activation frame im Speicher mit einer neuen Speicherzelle für jeden gebundenen Bezeichner des Blocks. Sie allokiert außerdem einen neuen environment frame, der auf die Speicherzelle im neuen activation frame zeigt. Dieser environment frame wird auf die aktuelle Umgebung gepusht. Nach strikter Auswertung werden Hilfsthreads zu der Arbeitsqueue hinzugefügt, um alle rechtsseitigen Ausdrücke in der erweiterten Umgebung auszuwerten. Der Ergebnisbezeichner wird ebenfalls in dieser erweiterten Umgebung ausgewertet.

eval block :

$$\frac{\langle_ \quad eval(\{x_1 = E_1; \ldots; x_n = E_n \; in \; x\}) : is \quad ps \quad ws \quad \sigma\rangle}{\langle_ \quad pushenv(p) : eval(x) : popenv : is \quad ps \quad ws ++ ws' \quad \sigma'\rangle}$$

$$p = \{x_1 \mapsto l_1, \ldots, x_n \mapsto l_n\}$$

where $$\sigma' = \sigma + \{l_1 \mapsto \langle defer[\ ] \rangle, \ldots, l_n \mapsto \langle defer[\ ] \rangle\}$$

$$ws' = \begin{bmatrix} ([eval(E_1), store(l_1), schedule], p : ps), \\ \ldots \\ ([eval(E_n), store(l_n), schedule], p : ps) \end{bmatrix}$$

Es ist auch möglich, die neu allokierten Blockspeicherplätze mit den thunks für ihre betreffenden rechtsseitigen Ausdrücke zu initialisieren, und nicht Threads in die Arbeitsqueue zu legen. Diese thunks werden in die Queue gelegt, wenn die Speicherzellen getoucht werden.

3. Seiteneffekt-Operationen

Funktionale Programmiersprachen erlauben keine Zuweisung oder mehrfache Speicherung. Allerdings werden solche Dinge häufig durch verschiedene Implementierungstricks angeboten. Außerdem können Seiteneffekte durch den traditionellen Zuweisungsoperator eingeführt werden, nachdem eine präzise sequentielle Reihenfolge aller Operatoren spezifiziert wurde (z.B. Scheme und ML). Id, durch die Frage nach Parallelität motiviert, bietet eine andere Alternative für verbundene Seiteneffekte. Bei Id ist es möglich, nur eine partielle Reihenfolge bei Seiteneffekt-Operationen zu spezifizieren und trotzdem ein insgesamt konsistentes Bild der Berechnung zu behalten.

pH unterstützt imperative Operationen auf I-Struktur- [4] und M-Strukturobjekten [6]. I-Strukturen erlauben die Erzeugung einer Datenstruktur, die eine Trennung von der Definition ihrer Komponenten ermöglicht : Versuche, den Wert einer Komponenten zu benutzen, werden automatisch verzögert, bis diese Komponente definiert ist; Versuche, Komponenten erneut zu definieren, führen zu einem Fehlerzustand. M-Strukturen andererseits sind voll veränderbare Datenstrukturen, deren Komponenten wiederholt neu definiert werden können : eine mFetch-Operation liest und leert eine volle Komponente; eine mStore-Operation definiert sie (erneut); zwei aufeinanderfolgende mStore-Operationen bei derselben Komponente führen zu einem Fehlerzustand, wenn sie nicht durch eine mFetch-Operation getrennt sind.

Für einige Programme in pH wird ein Weg benötigt, M-Strukturoperationen zu sequentialisieren, um ungewollte Effekte durch die parallele Auswertungsreihenfolge zu vermeiden. Dies wird durch die Benutzung von *control regions* und *barriers* [2] erreicht. Eine Kontrollregion ist als eine Menge von zusammenwirkenden Threads, die unter derselben Kontrollabhängigkeit sind und deshalb immer zusammen arbeiten, definiert. Threads innerhalb derselben Kontrollregion können in irgendeiner Reihenfolge oder auf eine verbundene Weise arbeiten, so lange die Datenabhängigkeiten unter ihnen respektiert werden. Barriers bieten einen Mechanismus, um die Terminierung einer Menge von parallelen Aktivitäten innerhalb einer Kontrollregion zu entdekken. Eine barrier (---) kreiert zwei Subregionen innerhalb einer gegebenen Kontrollregion - eine über der barrier genannt *pre-region* und die andere darunter genannt *post-region*. Keine Berechnung innerhalb der post-region darf fortfahren, bevor alle Seiteneffektberechnungen in der pre-region beendet sind. Diese Semantik ist anders als in [2], wo eine barrier wartet, bis alle Berechnungen beendet sind anstatt nur diese, die Seiteneffekte verursachen.

Abschließend in diesem Kapitel werden die pH-Erweiterungen beschrieben, mit denen man I- und M-Strukturoperationen behandelt. Die Semantik von Barriers wird in Kapitel 4 beschrieben.

3.1. Kernsprachenerweiterungen

Die Kernsprache wird erweitert, um Seiteneffekt-Operationen und barriers wie in Abb. 4 zu verbinden. Es werden primitives zum Allokieren, Lesen und Schreiben von I- und M-Strukturen hinzugefügt. Die Fetch- und

Constant	::=	... | •
PF^1	::=	... | iAlloc | mAlloc | iFetch | mFetch | W
PF^2	::=	... | iStore | mStore | &
S	::=	... | (S_1 --- S_2)

Abb. 4 : Kernel language extensions for side-effect operations

Storeoperationen adressieren direkt Speicherzellen: indiziertes Adressieren wird in der Kernsprache separat behandelt.

Die Syntax wird außerdem für Blockbindungen erweitert, um barriers zu ermöglichen - zwei Mengen von parallelen Blockbindungen können nun sequentialisiert werden, indem eine barrier (---) zwischen ihnen benutzt wird. Zusätzliche Konstanten (•), primitive Operatoren (W und &) und strikte Funktionsanwendung (sap) werden benutzt, um barriers in eine gewöhnliche Menge von Bindungen zu übersetzen.

3.2. Auswertungsregeln für Seiteneffekt-Operationen

In diesem Kapitel werden weitere Auswertungsregeln für interne primitive Operatoren beschrieben, dabei auch I- und M-Strukturen und barriers. Alle diese Operatoren sind strikt bei allen ihren Eingaben.

I-Struktur-Instruktionen

Die iAlloc-Instruktion allokiert ein leeres I-Strukturfeld. Die iFetch- und iStore-Instruktionen adressieren eine I-Strukturspeicherzelle direkt; jede Adreßarithmetik muß separat vorgenommen werden. Die iFetch-Instruktion ist mit loadi_*k* vergleichbar. Die iStore-Instruktion aktualisiert eine leere Speicherzelle mit einem Wert und reaktiviert Verzögerungen. Speichern in eine Zelle mit dem Zustand full wird als Fehler betrachtet.

iAlloc :

$$\frac{\langle _ \quad iAlloc(l_x) : is \quad ps \quad ws \quad \sigma\,[l_x \mapsto \langle full\ \underline{n} \rangle] \rangle}{\langle l \quad is \quad ps \quad ws \quad \sigma\,' \rangle}$$

where $\sigma\,' = \sigma + \{l \mapsto \langle defer[\ ] \rangle, \ldots, (l+n-1) \mapsto \langle defer[\ ] \rangle\}$

iFetch :

$$\frac{\langle _ \quad iFetch(l_x) : is \quad ps \quad ws \quad \sigma\,[l_x \mapsto \langle full\ l \rangle] \rangle}{\langle _ \quad touch(l) : load(l) : is \quad ps \quad ws \quad \sigma \rangle}$$

iStore :

$$\frac{\langle _ \quad iStore(l_x, l_z) : is \quad ps \quad ws \quad \sigma\,[l_x \mapsto \langle full\ l \rangle] \rangle}{\langle _ \quad atomic([storerr(l), load(l_z), store(l)]) : is \quad ps \quad ws \quad \sigma \rangle}$$

M-Struktur-Instruktionen :

Die mAlloc- und mStore-Instruktionen sind zu den entsprechenden Instruktionen bei I-Strukturen identisch. Die mFetch-Instruktion ist mit iFetch vergleichbar, leert aber die Zelle, auf die zugegriffen wird, und muß wiederholt versucht werden, falls die Zelle schon leer ist.

mAlloc :

$$\frac{\langle_ \quad mAlloc(l_x):is \quad ps \quad ws \quad \sigma\,[l_x \mapsto \langle full\ \underline{n}\rangle]\rangle}{\langle l \quad is \quad ps \quad ws \quad \sigma\ '\rangle}$$

where $\sigma\ ' = \sigma + \{l \mapsto \langle defer[\ ]\rangle, \ldots, (l+n-1) \mapsto \langle defer[\ ]\rangle\}$

mFetch :

$$\frac{\langle_ \quad mFetch(l_x):is \quad ps \quad ws \quad \sigma\,[l_x \mapsto \langle full\ l\rangle]\rangle}{\langle_ \quad atomic([rtouch(l), take(l)]):is \quad ps \quad ws \quad \sigma\rangle}$$

mStore :

$$\frac{\langle_ \quad mStore(l_x, l_z):is \quad ps \quad ws \quad \sigma\,[l_x \mapsto \langle full\ l\rangle]\rangle}{\langle_ \quad atomic([storerr(l), load(l_z), store(l)]):is \quad ps \quad ws \quad \sigma\rangle}$$

4. Semantik von M-Barriers

In diesem Kapitel wird die Semantik von barriers beschrieben, indem sie in gewöhnliche Blockbindungen übersetzt werden. Die Idee ist, systematisch ein zusammengesetztes Terminierungssignal von allen Seiteneffekt-Operationen, die innerhalb der pre-region einer barrier stattfinden (einschließlich der Kinderregionen) zu konstruieren, das dann benutzt wird, um die Operationen in der post-region der barrier zu triggern. Die Bedeutung einer Signalgenerierung und Zusammenstellung wird verständlicher, wenn man die Übersetzung eines einfachen parallelen Blocks betrachtet :

$$\begin{array}{lcl} TE[\{x_1 = E_1; & & \{x_1, s_1 = TE[E_1]; \\ \quad \ldots & & \quad \ldots \\ \quad x_n = E_n; & = & \quad x_n, s_n = TE[E_n]; \\ \quad in\ x\}] & & \quad in\ x, s_1\ \&\ \ldots\ \&\ s_n\} \end{array}$$

Ein Ausdruck E_i wird als **TE**[[E_i]] übersetzt, was dynamisch einen Wert (gebunden zu x_i) zurückgibt zusammen mit einem expliziten Terminierungssignal s_i. Der Operator W wird benutzt, um die Terminierung aller Seiteneffekt-Operationen (mFetch und mStore) innerhalb von E_i zu bemerken. Der Wert von s_i wird zu •, sobald alle Seiteneffekt-Operationen in E_i beendet sind. Der Operator & wird dann benutzt, um all diese Signale in ein einzelnes Signal für den ganzen Block zu bündeln.

Der primitive W-Operator produziert ein Signal •, wenn ein gegebener Bezeichner einen Wert erhält. Die generelle Auswertungsregel für strikte primitive Funktionen würde dafür sorgen, daß der Bezeichner, der betrachtet wird, getoucht wird, bevor der W-Operator angewendet wird. Der primitive &-Operator wird benutzt, um Signale von zwei Subausdrücken in ein Signal umzuwandeln, nachdem sie getoucht wurden.

W :

$$\frac{\langle_ \quad W(l_x):is \quad ps \quad ws \quad \sigma\,[l_x \mapsto \langle full\ v\rangle]\rangle}{\langle\bullet \quad is \quad ps \quad ws \quad \sigma\rangle}$$

& :

$$\frac{\langle_ \quad \&(l_1, l_2):is \quad ps \quad ws \quad \sigma\,[l_1 \mapsto \langle full\ \bullet\rangle, l_2 \mapsto \langle full\ \bullet\rangle]\rangle}{\langle\bullet \quad is \quad ps \quad ws \quad \sigma\rangle}$$

EXPRESSIONS

TE	::	Expression -> Expression
TE[[c]]	=	c, •
TE[[x]]	=	x, •
TE[[mFetch(x)]]	=	{ y = mFetch(x); in y, W(y) }
TE[[mStore(x, z)]]	=	{ y = mStore(x, z); in y, W(y) }
TE[[$PF^n(x_1, ..., x_n)$]]	=	$PF^n(x_1, ..., x_n)$, •
TE[[$\lambda x.E$]]	=	$\lambda x.$**TE**[[E]], •
TE[[case(x, E_1, ... , E_n)]]	=	case (x, **TE**[[E_1]], ..., **TE**[[E_n]])
TE[[ap(f,x)]]	=	ap(f, x)
TE[[{S in x}]]	=	{ S' in x, s }

where S', s = **TS**[[S]]

STATEMENTS

TS[[ ]]	::	Statement -> *List*(Statement) x Identifier
TS[[ε]]	=	(s = •), s
TS[[$x = E$]]	=	(x, s = **TE**[[E]]), s
TS [[S_1; ... ; S_n]]	=	(S_1'; ... ; S_n'; $s = s_1$ & ... & s_n), s

where S_i', s_i = **TS**[[S_i]] $1 \le i \le n$

TS[[S_1 --- S_2]]	=	(S_1'; $f = \lambda s.$\{ S_2' in y_1, ..., y_m , s_2 \}; y_1 , ..., y_m , s_2' = sap(f, s_1)), s_2'

where S_i', s_i = **TS**[[S]] $1 \le i \le 2$

y_1, ..., y_m = $BV(S_2)$

Abb. 5 : Semantics of M-barriers

Die komplette Barrier-Übersetzung ist in Abb. 5 dargestellt.
Die einzigen nichttrivialen Basisfälle für Signalgenerierung sind jene für die mFetch und mStore-Operatoren. Die mFetch-Operation wird als terminiert angesehen, wenn der Wert der Zelle, die gefetcht wurde, zurückgegeben wird. Die mStore-Operation wird entsprechend als beendet betrachtet, wenn sie den Wert zurückgibt, der gespeichert wurde.

In der Übersetzung von S_1 --- S_2 wird S_2 von einem λ-Ausdruck geschützt, so daß es nicht ausgewertet wird, bevor der λ–Ausdruck auf etwas (das Terminierungssignal von S_1) angewendet wird. sap wird benutzt, damit die Anwendung nicht beginnt, bevor das Terminierungssignal verfügbar ist. Schließlich, wenn die gebundenen Bezeichner von S_2 in einem inneren Bereich sind (im λ–Ausdruck), werden sie alle zurückgegeben und im äußeren Bereich gebunden.

Die Barriersemantik, die hier gezeigt wurde, ist anders als die Semantik in [2] insofern, daß hier nur die Terminierung der Seiteneffekt-Operationen innerhalb der pre-region einer barrier betrachtet wird, während die Semantik in [2] auf die Terminierung der ganzen Berechnung innerhalb der pre-region wartet.

5. Abschließende Bemerkungen

In dieser Arbeit wurden verschiedene Semantikthemen präsentiert, in denen sich pH von Haskell unterscheidet oder dieses erweitert. Der Hauptunterschied ist, daß pH eine parallele strikte Auswertungsstrategie benutzt im Gegensatz zur nicht-strikten Auswertungsstrategie bei Haskell. Zuerst wurde die Kernsprache von pH und ein normalisierender Interpreter, der die parallele Ausführungsreihenfolge implementiert, beschrieben. Als nächstes wurden die Sprache und der Interpreter um synchronisierte Seiteneffekt-Operationen, I-Strukturen und M-Strukturen erweitert. Schließlich wurde die Sequentialisierung von Seiteneffekten mit Hilfe von barriers ge-

zeigt. Es wurde eine systematische Übersetzung eines Kernsprache-pH-Programms mit barriers mit Hilfe von primitiven Terminierungsentdeckungsoperatoren gezeigt.

Der Interpreter benutzte einen expliziten Instruktionststrom statt direkt Kernsprachenausdrücke auszuwerten. Diese Organisation erlaubt es, ein multithreaded Compilierungsschema für die Kernsprache mit demselben Framework zu definieren. Der Prozeß des Compilierens kann als einfaches statisches Generieren aller Threads definiert werden anstatt den Instruktionsstrom dynamisch zu manipulieren. Dadurch wird hauptsächlich die eval-Instruktion entfernt. Ein solches Compilierungsschema wird in [1] beschrieben.

Zur Zeit gibt es eine arbeitende Implementierung von pH, die das Haskell / pH HBCC front-end (in Haskell geschrieben) von der Chalmers University benutzt und das Id Compiler Monsoon back end (geschrieben in Lisp) vom MIT. Das HBCC front-end produziert eine Kernsprache von pH mittleren Formats, die konvertiert wird in Dataflow Graphen und in das Monsoon back-end gespeist wird. Bis jetzt wurde dieser Compiler mit reinen funktionalen Haskellprogrammen oder pH-Programmen getestet. Am MIT wird auf einem neuen pH back-end gearbeitet für kommerzielle Uniprozessor- und Multiprozessorarchitekturen, was der Idee des Interpretationsschemas in dieser Arbeit näher kommt.

Literaturverzeichnis :

[0] S. Aditya, Arvind, L. Augustsson, J.-W. Maessen, R. S. Nikhil. Semantics of pH : A parallel dialect of Haskell

[1] S. Aditya. Normalizing Strategies for Multithreaded Interpretation and Compilation of Non-Strict Languages. CSG Memo 374, MIT Laboratory for Computer Science, Cambridge, MA 02139, May 1995

[2] S. Aditya, Arvind, J. E. Stoy. Semantics of Barriers in a Non-Strict, Implicitly-Parallel Language. In *Proc. Functional Programming Languages and Computer Architecture, La Jolla*, CA, Cambridge, MA 02139, June 1995. Also available as CSG Memo 367-1, MIT Lab. for Computer Sc., Cambridge, MA 02139

[3] Z. M. Ariola, Arvind. Properties of a First-order Functional Language with Sharing. CSG Memo 347-1, Laboratory for Computer Science, MIT, Cambridge, MA 02139, June 1994. To appear in *Theoretical Computer Science*, September 1995

[4] Arvind, R. S. Nikhil, K. K. Pingali. I-Structures : Data Structures for Parallel Computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598-632, 1989

[5] L. Augustsson, T. Johnsson. Parallel Graph Reduction with the < *v, G* >-machine. *In Proc. Fourth Intl. Conf. on Functional Programming Languages and Computer Architecture, London*, pages 202-213. ACM Press, September 1989

[6] P. S. Barth, R. S. Nikhil, Arvind. M-Structures : Extending a Parallel, Non-Strict, Functional Language with State. In *Proc. Functional Programming Languages and Computer Architecture*, pages 538-568. Springer-Verlag, 1991. LNCS 523

[7] A. P. W. Böhm, D. C. Cann, J. T. Feo, R. R. Oldehoeft. SISAL 2.0 Reference Manual. Technical Report UCRL-MA-109098, Lawrence Livermore National Laboratory, December 1991

[8] P. Hudak, S. P. Jones, P. Wadler (editors). Report on the Programming Language Haskell : A Non-Strict Purely Functional Language, Version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992

[9] T. Johnsson. Efficient Compilation of Lazy Evaluation. *Proc. ACM SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices*, 19(6):58-69, June 1984

[10] R. S. Nikhil. Id Language Reference Manual Version 90.1. Technical Report CSG Memo 284-2, Laboratory for Computer Science, MIT, Cambridge, MA 02139, July 15 1991

[11] R. S. Nikhil, Arvind, J. Hicks, S. Aditya, L. Augustsson, J.-W. Maessen, Y. Zhou. pH Language Reference Manual, Version 1.0 - preliminary. CSG Memo 369, Laboratory for Computer Science, MIT, Cambridge, MA 02139, January 1995

[12] S. L. P. Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127-202, April 1992