Seminar zur theoretischen Informatik: Konzepte von Programmiersprachen

Thema der vorliegenden Arbeit:

Fortsetzungssemantik für parallele Dialekte von Haskell

ausgearbeitet von Torsten Graf
betreut von Prof. Dr. Rita Loogen

Fachbereich Mathematik und Informatik
der Philipps-Universität Marburg

15. Juni 2005

Basierend auf der Arbeit:
Continuation semantics for Parallel Haskell Dialects
von Mercedes Hidalgo-Herrero und Yolanda Ortega-Mallén

Kurzzusammenfassung

Diese Arbeit beschäftigt sich mit der Umsetzung von Parallelität in der funktionalen Programmierung. Es wird ein Modell für eine Fortsetzungssemantik definiert, die es ermöglicht die verschiedenen Ansätze miteinander zu vergleichen und die Parallelität und dabei entstehende Seiteneffekte näher zu beleuchten. Es werden drei funktionale Programmiersprachen mit unterschiedlichen Umsetzungen näher betrachtet.

Inhaltsverzeichnis

1. Einführung

Wenn man bedenkt, dass Parallelität natürlich und inhärent in den meisten uns bekannten Problemstellungen vorhanden ist, so kommt man zu dem Schluss, dass man diese in der funktionalen Programmierung vorteilhaft nutzen kann, die es uns ermöglicht, auf einem abstrakteren Niveau zu programmieren,. Es gibt grundsätzlich verschiedene Ansätze, um Parallelität in Programmiersprachen umzusetzen. Im Folgenden wird die Klassifizierung in drei Bereiche vorgenommen. Diese Einteilung ist aus [LOO99], wo auch ein detaillierterer Einblick in diese Klassifizierung gegeben wird.

Es gibt zum Einen die Möglichkeit die Parallelität **explizit** in die Sprache einzuführen. Die Sprache wird dabei um Konstrukte zur expliziten Verwaltung der Prozesserstellung, -synchronisierung und -kommunikation erweitert. Somit liegt ein geringerer Abstraktionsgrad vor, in dem Sinne, dass die eben genannten Aufgaben vom Programmierer durchzuführen sind.

Zudem gibt es einen Ansatz, den man allgemein als **semi-explizit** bezeichnet. Bei diesem Ansatz wird dem Programmierer die Möglichkeit gegeben, dem Compiler den Wunsch nach paralleler Auswertung im Programm mitzuteilen. Es handelt sich hierbei um einen hohen Abstraktionsgrad, der es ermöglicht mit Skeletten zu arbeiten oder die Auswertungsstrategie festzulegen. Es muss allerdings beachtet werden, dass bei diesen Gegebenheiten eine zentrale Instanz in Form eines Schedulers eingesetzt werden muss, um die Reihenfolge der Auswertungen korrekt ablaufen zu lassen. Als Nachteil ist allerdings anzuführen, dass man auf Transformationssysteme und deren Algorithmen beschränkt wird.

Den dritten und letzten Ansatz kann man am Besten als **implizit** benennen. Man spricht in solchen Fällen von der Umsetzung eines primitiven Parallelismus auf niedrigem Niveau. Die Parallelität wird überwiegend aus voneinander unabhängigen Auswertungsausdrücken gewonnen, die parallel oder in beliebiger Reihenfolge ausgewertet werden können, was mit der inhärent parallelen Reduktionssemantik in Zusammenhang steht. Auf diese Art wird eine automatische Parallelisierung eines Programms vom Compiler durchgeführt.

Wie ebenfalls aus [LOO99] hervorgeht, wird die denotationelle Semantik einer Sprache ohne explizite Prozessverwaltung nicht verändert. Dies ist unter der Voraussetzung einer gewissen standardisierten denotationellen Semantik zu sehen, worauf etwas später noch eingegangen wird. Es gibt allerdings verschiedene Meinungen dazu, in welchem Maße die Semantik einer Programmiersprache überhaupt in Verbindung mit der Umsetzung von Parallelität steht oder ob es eine reine Implementierungsfrage sein sollte.

Es soll in dieser Arbeit allerdings um die Möglichkeit des Vergleichs von Programmen gehen, mit Blick auf den Aufwand, zu einer Eingabe eine Ausgabe zu produzieren. Bei einer bedarfsgesteuerten Auswertung, die inhärent sequentiell ist, ist die Betrachtung einer möglicherweise unnötigen Kreierung von parallelen Prozessen und der Berechnung von unnötigen Ergebnissen nahe liegend. Dazu sollen drei verschiedene Aspekte betrachtet werden. Zum Einen die *Funktionalität* (der berechnete Ergebniswert), zum Anderen der *Parallelismus* (Prozesserstellungen, -verbindungen (Systemtopologie) und die Interprozessorkommunikation zur Berechnung), sowie als Letztes die *Aufteilung* (der Grad von Verdopplung von Arbeit und der Zunahme möglicher Berechnungen).
In diese Richtung wurden schon einige Forschungen durchgeführt für parallele bzw. nebenläufige funktionale Sprachen. So existieren für die Sprache CML (Concurrent Meta Language), die in [Rep92] beschrieben wird, zwei Ansätze für eine denotationelle Semantik, welche in [DB97] und [FH99] nachzulesen sind. Beide Ansätze gehen auf das Acceptance Tree Modell aus [Hen88] zurück. Da es sich bei CML um eine strikte Sprache handelt, ist die Semantisierung um einiges einfacher. Zudem werden nur die Punkte Funktionalität und Parallelismus betrachtet, da der Grad von Verdopplung von Arbeit und die Zunahme möglicher Berechnungen nicht beachtet werden muss.

1.1 Die Sprache Haskell und ihre parallelen Dialekte

Die eine bedarfsgesteuerte Auswertungsstrategie verfolgende funktionale Sprache Haskell ist breit gefächert und besitzt in der Gemeinschaft der funktionalen Programmierer einen hohen Stellenwert. Haskell wird in [Pey03] beschrieben und ist im Gegensatz zu anderen funktionalen Sprachen überdurchschnittlich gut dokumentiert und vollständig semantisiert. In den 90er Jahren wurden auf der Grundlage von Haskell und auch anderen funktionalen Sprachen neue Sprachen definiert, die zusätzlich parallele Sprachkonstrukte enthielten.
Dieser Arbeit wurden drei parallele Dialekte von Haskell zu Grunde gelegt, welche nach den oben bereits vorgestellten Umsetzungsarten ausgewählt wurden.
Als Erstes wäre da eine Sprache zu nennen, welche die Parallelität explizit umsetzt. Die hier gewählte Sprache heißt **Eden**. Eden wird in [BLOP98] beschrieben. Bei dieser Umsetzung werden Konstrukte für die explizite Erzeugung, Verwaltung und Kommunikation von Prozessen eingeführt, wie sie auch schon in ähnlicher Form in [KM77] beschrieben wurden. Zudem ist in Eden eine beschränkte Form von Nichtdeterminismus vorhanden um viele-zu-eins Kommunikationen zu modellieren.

Als Nächstes sollte hier, nun der Reihenfolge der möglichen Implementierungen von Parallelität folgend, eine Sprache genannt werden, welche die Parallelität semi-explizit einführt. Diese Sprache ist **GpH** (Glasgow parallel Haskell). In der Sprache GpH, welche in [THM+96] beschrieben wird, werden die zwei neuen Kombinatoren '`seq`' und '`par`' vom Typ a -> b -> b zur Verfügung gestellt. Mit '`seq`' wird sequentiell verknüpft. Mit '`par`' gibt es nun die Möglichkeit, dem Compiler mitzuteilen, dass die Auswertung parallel durchgeführt werden soll. Die Festlegung der Auswertungsstrategie wird also dem Programmierer überlassen. Die Idee der expliziten Anweisung einer gewollten parallelen Auswertung geht auf P. Hudak in [Hud86] zurück.
Zuletzt wird noch die Sprache **pH** (paralleles Haskell) aufgeführt. Sie wird beschrieben in [NA01] und [AAAH+95]. Diese Sprache macht den Versuch, die Parallelität implizit umzusetzen. In pH werden zusätzlich zu Datenkonstruktoren mit mehreren Argumenten und lokalen Definitionen besondere Datenstrukturen eingeführt, die es ermöglichen, parallel arbeitende Prozesse zu synchronisieren. Anders als beim seiteneffektfreien Haskell kann es bei pH zu Seiteneffekten und Nichtdeterminismus kommen.
Um einen ersten Eindruck der verschiedenen Umsetzungen zu bekommen sollen an einem kurzen Beispiel die verschiedenen Umsetzungen verglichen werden.
Es sollen die Summen zweier Listen miteinander multipliziert werden. In Haskell sieht diese Berechnung folgendermaßen aus: `let s1 = sum l1, s2 = sum l2 in s1 * s2`. Hierbei ist `sum` die Standardfunktion, alle Elemente einer Liste aufzusummieren. Zudem ist vorausgesetzt, dass die beiden Listen `l1` und `l2` bereits initialisiert wurden. In pH mit zwei verfügbaren Prozessoren würde die Auswertung von `sum l1` und `sum l2` parallel ablaufen, auch auf die Gefahr hin, dass die Ergebnisse nicht benötigt werden. Somit wird also strikt ausgewertet. In GpH würde der Aufruf wie folgt aussehen: `let s1 = sum l1, s2 = sum l2 in s2 'par' (s1 * s2)`. In diesem Aufruf werden `s2` und `(s1 * s2)` parallel ausgewertet und der Wert für `(s1 * s2)` wird auf jeden Fall zurückgegeben. Zum Schluss kommt noch die Formulierung dieses Beispiels in Eden: `let p = process list -> sum list, s1 = p # l1, s2 = p # l2 in s1 * s2`. Bei diesem Ausdruck wird eine Prozessabstraktion `p` definiert, die im Rumpf die Summe einer Liste enthält. Von ihr werden zwei Instanzen benötigt, die durch `#` erzeugt werden. Es werden zwei Prozesse geschaffen, die parallel zum Prozess der let-Anweisung, die Summe der jeweiligen Listen bilden. Dabei ist der erschaffende Prozess als Elternprozess zu sehen. Dieser Elternprozess muss die jeweilige Liste an die Kindprozesse übergeben und die Kindprozesse übermitteln dann ihr Ergebnis dem Elternprozess.

1.2 Eine Fortsetzungssemantik

Das größte Problem bei der Parallelisierung von Haskell ist seine charakteristische bedarfsgesteuerte Auswertungsstrategie. Bei der Einführung von Parallelität müssen Änderungen an der Auswertungsreihenfolge vorgenommen werden. In diesem Zusammenhang steht natürlich die Kombinierung von bedarfsgesteuerter und strikter Auswertung. Allgemein kann davon ausgegangen werden, dass viele verschiedene Berechnungen durchgeführt werden müssen, deren Verwaltung von der Anzahl der verfügbaren Prozessoren und der Geschwindigkeit von Basisoperationen abhängt. Bei der Definition einer formalen Semantik für solche Sprachen kann man eine Ansicht auswählen, die nur von der Durchführung notwendiger Berechnungen ausgeht bis hin zur Durchführung aller möglichen Berechnungen.
Für alle drei hier vorgestellten Sprachen sind bereits operationelle Semantiken verfügbar. Die Semantik für pH wird in [AAAM+95] auf der Basis der G-Maschine, die in [Pey87] erläutert wird, ausführlich beschrieben. Für GpH wurde eine solche Semantik in [BFKT00] gegeben, die als Einzelschrittsemantik für die lokalen Reduktionen und als Gesamtschrittsemantik für die Verwaltung und Parallelisierung gegeben ist. Eine ähnliche zweispurige Semantik wird in [HOM02] für Eden gegeben, wobei hier Transitionen für einzelne lokale Prozesse und das globale System unterschieden werden.
In dem mir vorliegenden Artikel wird ein allgemeiner Rahmen definiert, in dem alle Semantiken der aufgezählten Sprachen betrachtet und verglichen werden können. Aus diesem Grund fällt die operationelle Semantik weg, da sie zu nah an der jeweiligen Sprache ist. Um das Geforderte zu erreichen, wird eine Art der Semantik betrachtet, wie sie in [Stoy77] definiert wird. Es handelt sich dabei um die standardisierte denotationelle Semantik nach Scott und Strachey. Diese Art der Semantik befasst sich nur mit dem Ein-/Ausgabe-Verhalten von Programmen. Diese muss allerdings noch um eine Komponente erweitert werden, die für die Generierung des Prozesssystems verantwortlich ist. Die Behauptung der Zugehörigkeit des Prozesssystems zum Programm meint dabei nur, dass bei der Auswertung von Ausdrücken Seiteneffekte entstehen können, die gerade im Zusammenhang mit der Bedarfssteuerung beachtet werden müssen. Es könnte z.B. bei der Auswertung von (`\x.3`)y von einem Prozess p, y von der Berechnung anderer Werte abhängen. Dies kann allerdings unbeachtet bleiben, da nur das endgültige Ergebnis betrachtet wird und zudem in dieser Arbeit nicht die Modifizierung des Prozesssystems durchgeführt werden soll, auf Grund des Umstandes, dass die λ-Abstraktion nicht strikt in ihren Argumenten ist und somit die Berechnung von y nicht

gefordert wird. Deshalb sind die Prozesserstellung und die Kommunikation von Werten Seiteneffekte, die nur auf Anfrage stattfinden sollten. Es wird sogar darüber hinaus daran angeknüpft, dass bei Bedarfssteuerung die Argumente von Funktionsaufrufen höchstens einmal berechnet werden. Deshalb muss darauf geachtet werden, dass ein Seiteneffekt nur bei der ersten Anforderung eines Wertes produziert wird.

Fortsetzungen lösen die Probleme, die bei der Abfolge der Anweisungen in einer denotationellen Semantik entstehen, auf eine sehr vornehme Art. Die Bedeutung einer jeden Anweisung ist eine Funktion, die das endgültige Ergebnis des gesamten Programms wiedergibt. Es bedeutet, dass jede Anweisung ein zusätzliches Argument bekommt, welches einen Zustand in ein Endergebnis überführt. Dieses zusätzliche Argument, die Fortsetzung, beschreibt, wie der Rest der Berechnung auszuführen ist, wenn die Anweisung jemals ihre Berechnung beendet. Dies ist in dieser Art in [Rey98] beschrieben.

In dem Fall von funktionalen Programmiersprachen ist eine Fortsetzung eine Funktion, die Werte auf Endergebnisse abbildet. Die semantische Funktion für die Auswertung von Ausdrücken hängt von der Umgebung und einer Fortsetzung ab, so dass man das Ergebnis des gesamten Programms durch Anwendung der Fortsetzung auf den vom Ausdruck gelieferten Wert bekommt.

Damit Seiteneffekte in der funktionalen Programmierung handhabbar bleiben, kombinierte Josephs in [Jos86] zwei verschiedene Sichtweisen von Fortsetzungen, um eine Ausdrucksfortsetzung zu bekommen. Bei einer Ausdrucksfortsetzung handelt es sich um eine Funktion, die einen Wert bekommt und wieder eine Anweisungsfortsetzung zurückgibt oder eine Zustandstransformation bewirkt. Weiterhin wird in [Jos89] beschrieben, wie diese Ausdrucksfortsetzungen verwendet werden, um eine angemessene Auswertungsreihenfolge in einer denotationellen Semantik für einen bedarfsgesteuerten λ-Kalkül zu modellieren. In der mir vorliegenden Arbeit wird diese Idee mit dem Ziel verwendet, eine auf Fortsetzungen basierende denotationelle Semantik für parallele bedarfsgesteuerte Sprachen zu definieren, in denen Prozesserstellung, Interprozesskommunikation und Synchronisierung als Seiteneffekte aufgefasst werden, die während der Auswertung von Ausdrücken entstehen.

In [Em02] wird eine Mischung aus strikter und bedarfsgesteuerter Semantik präsentiert. Mal davon abgesehen, dass darin keine Parallelität enthalten ist, testet die strikte Semantik nur, ob ein Ausdruck zu einer Normalform ausgewertet wird oder ob die aktuelle Berechnung nur bei Bedarf durchgeführt wird. Auf dieser Grundlage war es nicht möglich, eine angemessene denotationelle Semantik zu definieren, wenn man bedenkt, welche Seiteneffekte die Ausdrücke haben können.

Soweit es den Autoren des mir vorliegenden Artikels und mir bekannt ist, ist die hier definierte denotationelle Semantik, die nicht nur ein Endergebnis beschreibt, sondern sich auch mit diesen Seiteneffekten beschäftigt, welche in einem bedarfsgesteuerten Zusammenhang entstehen, die Erste ihrer Art.
Es war nicht das Ziel eine vollständige denotationelle Semantik jeder Sprache zu definieren. Für das geplante Vorhaben reichte es aus, sich auf einen sehr einfachen funktionalen Kern zu beschränken. Damit ist ein bedarfsgesteuerter λ-Kalkül gemeint, der um einige sprachlich unterschiedliche Charakteristika ergänzt wird. Um die Vergleichbarkeit zu erhöhen, wird deshalb versucht, eine einheitliche Syntax zu finden.
Wie auch im vorliegenden Artikel gliedert sich meine restliche Arbeit nun in drei Teile. In jedem Kapitel wird jeweils eine der Sprachen Eden, GpH und pH zuerst mit ihren typischen Merkmalen vorgestellt. Dann wird die oben erwähnte Kern-Syntax definiert und danach die denotationelle Semantik durch die semantischen Bereiche und die Semantik der Auswertungsfunktionen dargestellt. Zum Schluss wird noch ein kurzes Resümee gezogen und es werden Ausblicke auf zukünftige Entwicklungen gegeben.

2. Die Sprache Eden

Bei Eden handelt es sich um ein durch eine Menge von Koordinierungsmerkmalen erweitertes Haskell, die einerseits auf dem Konzept der expliziten Verwaltung von Prozessen und andererseits auf dem Konzept der impliziten Kommunikation basieren. Funktionale Sprachen unterscheiden zwischen der Definition und der Applikation von Funktionen. Auf eine ähnliche Art bietet Eden eine Prozessabstraktion, mit abstrakten Schemata für das Verhalten von Prozessen, und eine Prozessinstanzierung für die eigentliche Erstellung von Prozessen und den dazu korrespondierenden Kommunikationskanälen an.
Die Kommunikation in Eden ist unidirektional, was bedeutet, dass genau von einem Erzeuger zu einem Verbraucher kommuniziert wird. Es können beliebige Kommunikationstopologien dadurch hergestellt werden, dass die Antwortkanäle dynamisch sind. Um zu kontrollieren, wo Ausdrücke ausgewertet werden, können nur vollständig ausgewertete Datenobjekte übermittelt werden. Dies erhöht den Parallelitätsgrad in Eden-Programmen und bewahrt die Bedarfssteuerung der ihr zu Grunde liegenden funktionalen Sprache. Listen werden in einem Strom, ein Element nach dem anderen, übermittelt. Nebenläufige Threads, die auf eine nicht verfügbare Eingabe warten, werden suspendiert. Dies ist die einzige Möglichkeit, in Eden Prozesse zu synchronisieren.

In Eden wird der Nichtdeterminismus durch eine vordefinierte Prozessabstraktion eingeführt, die eine Liste von Eingabekanälen zu einer einzelnen Liste zusammenfügt.
Wer vorhat, sich genauer mit der Sprache Eden auseinanderzusetzen, dem empfehle ich [BLOP95] und [BLOP96]. An dieser Stelle wird nur der Kern von Eden betrachtet:der ungetypte λ-Kalkül mit der Kategorie der Bezeichner ($x \in$ **Ide**) und der Ausdrücke ($E \in$ **Exp**). Dieser Kern wird in Abb. 1 zusammengefasst.

E ::=	x	Bezeichner
|	\x.E	λ-Abstraktion
|	x_1 $ x_2	bedarfsgesteuerte Applikation
|	x_1 $# x_2	parallele Applikation
|	let {x_i = E_i}$_{i=1}^{n}$ in x	lokale Definition
|	x_1 ▯ x_2	nichtdeterministische Auswahl

Abb. 1. Eden Kernsyntax

Der Kalkül mischt bedarfsgesteuerte und strikte Auswertung dadurch, dass zwei verschiedene Applikationen zur Verfügung gestellt werden. Zum Einen eine bedarfsgesteuerte Applikation (x_1 $ x_2), welche x_2 nur dann auswertet, wenn es benötigt wird, und eine parallele Applikation (x_1 $# x_2). Bei der parallelen Applikation wird ein neuer paralleler Prozess gestartet, der x_1x_2 auswertet. Dabei werden zwei Kanäle zwischen dem Elternprozess und dem Kindprozess eingerichtet, wobei auf dem einen Kanal der Wert von x_2 vom Elternprozess zum Kindprozess und auf dem anderen Kanal der berechnete Wert aus x_1x_2 vom Kindprozess zum Elternprozess gesendet wird. Zum Schluss stellt der Kern von Eden noch eine nichtdeterministische Auswahl zwischen zwei Argumenten zur Verfügung (x_1 ▯ x_2).
In [Lau93] wird ein Kalkül präsentiert, bei dem es sich um ein mit der (rekursiven) let-Anweisung erweiterten λ-Kalkül handelt. Dieses wird dann auf eine beschränkte Syntax normalisiert, in der alle Teilausdrücke, außer dem Rumpf einer λ-Abstraktion, mit Variablen assoziiert werden. Dieser Ansatz wurde auch in dieser Arbeit verfolgt, weil zum Einen auf diese Weise alle Teilausdrücke eines Ausdrucks gemeinsam genutzt werden können, was dazu führt, dass der Grad an gemeinsam benötigten Berechnungen innerhalb einer übergeordneten Berechnung maximiert wird und somit garantiert werden kann, dass jeder Teilausdruck höchstens einmal ausgewertet wird. Zum Anderen wird dadurch die Auswertung von bedarfsgesteuerten Applikationen einfacher, weil durch die Normalisierung die Einführung neuer Variablen für die Argumente einer Applikation wegfällt.

2.1. Semantische Bereiche von Eden

Die semantischen Bereiche, die zur Definition der Fortsetzungssemantiken benötigt werden, sind in Abb. 3 aufgeführt.

	Cont	=	**State** → **SState**	Fortsetzungen
$\kappa \in$	**ECont**	=	**EVal** → **Cont**	Ausdrucksfortsetzungen
$s \in$	**State**	=	**Env** × **SChan**	Zustände
$S \in$	**SState**	=	P_f(**State**)	Menge von Zuständen
$\rho \in$	**Env**	=	**Ide** → (**Val** ∪ {`not_ready`})	Umgebungen
Ch ∈	**SChan**	=	P_f(**Chan**)	Menge von Kanälen
	Chan	=	**IdProc** × **CVal** × **IdProc**	Kanäle
	CVal	=	**EVal** ∪ {`unsent`}	Kommunikationswerte
$\upsilon \in$	**Val**	=	**EVal** ∪ (**IdProc** × **Clo**) ∪ {`not_ready`}	Werte
$\varepsilon \in$	**EVal**	=	**Abs** × **Ides**	Ausdruckswerte
$\alpha \in$	**Abs**	=	**Ide** → **Clo**	Abstraktionswerte
$\nu \in$	**Clo**	=	**IdProc** → **ECont** → **Cont**	closures
$I \in$	**Ides**	=	P_f(**Ide**)	Menge von Bezeichnern
$p,q \in$	**IdProc**			Prozessbezeichner

Abb. 2. semantische Bereiche von Eden

Wie in der Einführung schon erläutert, handelt es sich bei der Bedeutung eines Programms, d.h. eines Ausdrucks, um eine Fortsetzung, hier eine Zustandstransformation. Um den Nichtdeterminismus in Eden zu behandeln, wird eine Menge von Zuständen betrachtet, so dass eine Fortsetzung einen Anfangszustand in eine Menge von möglichen Endzuständen umwandelt. Zur Notation sollte gesagt werden, dass mit P_f (M) die Menge gemeint ist, in der alle endlichen Teilmengen von M enthalten sind. In dem vorliegenden Fall besteht ein Zustand (s ∈ **State**) aus zwei Komponenten. Zum Einen besteht er aus einer Umgebung ($\rho \in$ **Env**). Eine Umgebung bildet Bezeichner auf Werte ab, weshalb man also eine Analogie zu Speicherplätzen imperativer Variablen sehen kann. Es wird hier nicht von der Möglichkeit gebraucht gemacht, die in [Stoy77] gegeben wird, die Umgebung von dem Zustand zu trennen, weil keine Auswertung in Eden unwiderrufliche Änderungen in einem Zustand vornimmt, so dass Erhaltung und Wiederherstellung einer Umgebung leicht durchzuführen sind. Zum Anderen besteht ein Zustand aus einer Menge von Kanälen (Ch ∈ **SChan**). Prozesse sind in Eden keine losgelösten Einheiten, sondern sie kommunizieren unter zur Hilfenahme von unidirektionalen und einwertigen Kanälen. Jeder Kanal ist durch ein Tripel <Sender, Wert, Empfänger> gekennzeichnet. Der Wert kann entweder der Ausdruckswert

sein, welcher mitgeteilt wurde oder '`unsent`', wenn keine Kommunikation über diesen Kanal stattgefunden hat.

Der Bereich der Werte (**Val**) beinhaltet endgültige denotationelle Werte (oder Ausdruckswerte $\varepsilon \in$ **EVal**). Zudem enthält dieser Bereich die closures ($\nu \in$ **Clo**) wie z.B. noch nicht vollständig ausgewertete Ausdrücke. Die Auswertung eines closures könnte zudem die Erstellung eines neuen Prozesses bedeuten. Aus diesem Grund ist es bei der Erstellung der Prozesssystemtopologie wichtig zu wissen, welcher Prozess der Elternprozess des neu Erstellten ist, weshalb dann auch die Prozessbezeichner mit closures assoziiert werden. Der zusätzliche Wert '`not_ready`' zeigt die gerade stattfindende Auswertung des jeweiligen closures an. Zudem wird dieser Wert benutzt um Selbstreferenzen zu entdecken.

Abstraktionswerte ($\alpha \in$ **Abs**) sind die einzige Art von Ausdruckswerten in diesem Kalkül. Jede Abstraktion wird dargestellt durch eine Funktion, die Bezeichner nach closures abbildet, zusammen mit einer Liste von freien Variablen, die ‚vor dem Übermitteln des Abstraktionswertes auf irgendeinem Kanal, berechnet werden müssen.

Ein closure ist wiederum eine Funktion, die abhängt von dem Prozess, in dem das jeweilige closure ausgewertet wird und der Ausdrucksfortsetzung, die den Rest des Programms bestimmt.

2.2. Anweisungssemantik von Eden

Bei der Verteilung einer Berechnung auf mehrere parallele Prozesse ist es notwendig, den Prozess kenntlich zu machen, in dem ein Ausdruck ausgewertet werden soll. Dazu sei angemerkt, dass der Prozessbezeichner nicht notwendigerweise seine Variablen von denen anderer Prozesse separiert, da jedem Prozess unterschiedliche Bezeichner zugewiesen werden. Allerdings kann dadurch die Eltern-Kind Beziehung bei der Prozessgenerierung bestimmt werden. Die semantische Funktion für Ausdrücke lautet in Eden:

$$\mathcal{L} : \textbf{Exp} \rightarrow \textbf{IdProc} \rightarrow \textbf{ECont} \rightarrow \textbf{Cont}$$

Nach dem Auswerten einer Anweisung wird die Fortsetzung durch Instanzierung der Ausdrucksfortsetzung mit dem Wert, der für den Ausdruck produziert wurde, begonnen, die dann weiterrechnet.

In Abb. 3 wird die Anweisungssemantik für den Eden-Kern ausführlich dargestellt. Es wird der Operator $\oplus$ verwendet um die Erweiterung oder die Aktualisierung einer Umgebung anzuzeigen, z.B. $\rho \oplus \{x \mapsto \nu\}$. Zudem soll $\oplus_{ch}$ selbiges nur für eine Menge von Kanälen eines Zustands bedeuten.

$\mathcal{L}[\![x]\!]p\kappa = \texttt{force}\ x\ \kappa$

$\mathcal{L}[\![\backslash x.E]\!]p\kappa = \kappa\langle\lambda x.\mathcal{L}[\![E]\!], fv(\backslash x.E)\rangle$

$\mathcal{L}[\![x_1\ \$\ x_2]\!]p\kappa = \mathcal{L}[\![x_1]\!]\,p\kappa'$

$\texttt{where}\ \kappa' = \lambda\langle\,\alpha,I\,\rangle.\lambda s.(\alpha x_2)\ p\kappa s$

$\mathcal{L}[\![x_1\ \$\#\ x_2]\!]p\kappa = \texttt{forceFV}\ x_1\ \kappa'$

$\texttt{where}\ \kappa' = \lambda\langle\,\alpha,I\,\rangle.\lambda s.(\alpha x_2)\ q\kappa''s$

$q = \text{newIdProc}\ s$

$\kappa''_{min} = \lambda\langle\alpha',I'\rangle.\lambda s'.\texttt{case}\ (\rho' x_2)\ \texttt{of}$
$\langle\alpha'',I''\rangle \in \mathbf{EVal} \rightarrow S_d \oplus_{ch} \{\langle q, \langle\,\alpha,I\,\rangle, p\rangle, \langle p, \langle\,\alpha'',I''\,\rangle, q\rangle\}$
$\texttt{otherwise} \rightarrow S_d \oplus_{ch} \{\langle q, \langle\,\alpha,I\,\rangle, p\rangle, \langle p, \texttt{unsent}, q\rangle\}$
$\texttt{endcase}$
$\texttt{where}\ (\rho', Ch') = s'$
$S_c = \texttt{mforceFV}\ I'\ s'$
$S_d = \bigcup_{s_c \in S_c} \texttt{mforceFV}\ I''\ s_c$

$\kappa''_{max} = \lambda\langle\alpha',I'\rangle.\lambda s'.\ \bigcup_{s_c \in S_c} \texttt{forceFV}\ x_2\ \kappa_c\ s_c$
$\texttt{where}\ S_c = \texttt{mforceFV}\ I'\ s'$
$\kappa_c = \lambda\varepsilon''.\lambda s''.\{s'' \oplus_{ch} \{\langle q, \langle\alpha,I\rangle, p\rangle, \langle p, \varepsilon'', q\rangle\}$

$\mathcal{L}[\![\text{let}\ \{x_i = E_i\}_{i=1}^{n}\ \text{in}\ x\ ]\!]p\kappa = \lambda\langle\rho,Ch\rangle.\mathcal{L}[\![x]\!]p\kappa'\langle\rho',Ch\rangle$

$\texttt{where}\ \{y_1,\ldots,y_n\} = \text{newIde}\ n\ \rho$
$\rho' = \rho \oplus \{y_i \mapsto \langle\mathcal{L}[\![\,E_i[y_1/x_1,\ldots,y_n/x_n]]\!], p\rangle \mid 1 \le i \le n\}$

$\kappa'_{min} = \kappa$

$\kappa'_{max} = \lambda\varepsilon.\lambda s.\texttt{mforce}\ I\ s$
$\text{where}\ I = \{y_i \mid E_i \equiv x_1^i\ \$\#\ x_2^i \wedge 1 \le i \le n\}$

$\mathcal{L}[\![x_1\ \square\ x_2]\!]p\kappa = \lambda s.(\mathcal{L}[\![x_1]\!]p\kappa s) \cup (\mathcal{L}[\![x_2]\!]p\kappa s)$

Abb. 3. Auswertungsfunktion für die Anweisungen der Eden-Kernsyntax

Die Auswertung eines Bezeichners "erzwingt" die Auswertung des Wertes, mit dem dieser in der gegebenen Umgebung belegt ist. Die dazu benötigte Funktion `force` ist in Abb. 4 definiert und wird später erklärt.

Bei der Auswertung einer λ-Abstraktion wird der zugehörige Ausdruckswert, welcher ein Paar aus einer Funktion, welche einem gegebenen Bezeichner ein closure zuweist, und der Menge von freien Variablen der eigenen syntaktischen Konstruktion, erstellt.

Im Falle einer bedarfsgesteuerten Applikation wird die Auswertung von x_2 aufgeschoben, bis es tatsächlich benötigt wird. Die Ausdrucksfortsetzung κ, welche für die Auswertung der Applikation gegeben wurde, wird modifiziert zu κ'. Diese Ausdrucksfortsetzung κ' wird nun

benötigt, um die zur Applikationsabstraktion gehörige Variable x_1 auszuwerten. Dazu wird zuerst der Abstraktionswert benötigt. Dieser wird der Argumentvariablen x_2 zugewiesen und dann wird das zugehörige closure berechnet.

Es wird nun bei der Auswertung der parallelen Applikation `x1 $# x2` ein neuer Prozess q geschaffen, wobei q ein noch nicht benutzter Prozessbezeichner ist. Dann werden zwei neue Kanäle, ⟨q,_,p⟩ und ⟨p,_,q⟩ zwischen dem Elternprozess p und dem Kind q erstellt. Nun kommt es zu der Auswertung von x_1, um den Abstraktionswert zu bekommen zusammen mit seinen freien Variablen ⟨α,I⟩. Dazu wird die Funktion `forceFV` benutzt, die wie `force` ebenfalls in Abb. 4 definiert wird. Bekanntermaßen teilen sich in Eden die Prozesse nicht unbedingt einen Arbeitsspeicher. Deshalb müssen dem Heap, auf dem der neue Prozess ausgeführt wird, alle Bindungen in Bezug auf die freien Variablen der Wertabstraktion des korrespondierenden Prozessrumpfes enthalten. Jede dieser Informationen wird im Elternprozess ausgewertet und dann dem Kind mitgeteilt. Weiterhin wird x_2 ausgewertet. Wie in der Einführung bereits angesprochen, variiert der mutmaßliche Parallelismus von einem Minimum zu einem Maximum. Im Fall von Eden kann man eine minimale Semantik definieren, bei der x_2 nur bei Bedarf ausgewertet wird, wohingegen man in einer maximalen Semantik diese Auswertung immer stattfinden lassen würde. Der Wert von x_2 muss vom Eltern zum Kind kommuniziert werden zusammen mit allen Informationen die freien Variablen betreffend. Die zuvor angesprochene Funktion `forceFV` wird wieder benutzt um x_2 auszuwerten und seine freien Variablen zu bestimmen. Es kommt nun, auf Grund der definierten minimalen Semantik nicht notwendigerweise, zu einer Übermittlung des Wertes für x_2 vom Elternprozess p zum Kindprozess q. Danach wird die Applikation x_1x_2 in dem neuen Prozess q ausgewertet. Zum Schluss wird der Wert von x_1x_2 von q nach p übermittelt. Vor dieser Kommunikation müssen allerdings auch die freien Variablen mit der Funktion `forceFV` ausgewertet werden.

Zu der Auswertung des Rumpfes einer lokalen Definition ist zu sagen, dass alle lokalen Variablen x_i (mit frischen Bezeichnern y_i um Namenskonflikte zu vermeiden) in die Umgebung eingefügt werden. Jede neue lokale Variable y_i wird dem passenden closure $E_i[y_j/x_j]$, welches man sich vom korrespondierenden Ausdruck verschafft, zugeordnet. Die closures werden zusammen mit dem Prozessbezeichner p, in dem die Definition ausgewertet wird, gepaart. Wenn eine lokale Variable als parallele Applikation definiert wurde, sollte diese ebenfalls ausgewertet werden (im Falle einer maximalen Semantik).

Zum Schluss handelt es sich bei der Auswertung des Operators der nichtdeterministischen Auswahl um die Möglichkeit irgendeins der Argumente, welche wieder Ausdrücke sind,

auszuwerten. Dies schlägt sich in der Weise in der Semantik dieser Funktion nieder, dass die jeweiligen Auswertungen vereinigt werden.

Die oben verwendete Hilfsfunktion `force`, welche eine Auswertung erzwingt, hängt vom Wert ab, der mit dem Bezeichner verbunden ist. Im Falle eines Ausdruckswertes wird die Ausdrucksfortsetzung darauf angewendet, um die Berechnung fortzuführen. Im Falle eines closures, muss dieses ausgewertet werden um den Bezeichner dann mit dem Ergebnis in der Umgebung zu belegen. Während der Auswertung des closures, wird der Bezeichner mit `not_ready` belegt. Sollte jemals die Auswertung eines mit `not_ready` belegten Bezeichners erzwungen werden, deutet dies auf eine Selbstreferenz hin. Nach der Auswertung des closures wird die Ausdrucksfortsetzung auf den bekommenen Ausdruckswert und dem modifizierten Zustand angewendet. Selbstverständlich ist das Erzwingen der Auswertung einer Variablen mit der Belegung `undefined` oder `not_ready` ein Fehler.

Der Unterschied von der Funktion `forceFV` zu `force` liegt im Wesentlichen im Bereich, in dem ausgewertet wird. Während bei force nur das zur Variablen gehörige closure ausgewertet wird, berechnet `forceFV` die freien Variablen dieses closures und diejenigen, die mit den closures dieser freien Variablen korrespondieren und so weiter. Die Auswertung für eine Menge von Bezeichnern wird durch `mforceFV` zur Verfügung gestellt.

`force` :: **Ide → ECont → Cont**

`force` x κ = λ⟨ρ,Ch⟩.`case` (ρ x) `of`
  ε ∈ **EVal** → κε⟨ρ,Ch⟩
  ⟨p,ν⟩ ∈ (**IdProc** × **Clo**) → νpκ's'
    `where` κ' = λε'.λ⟨ρ',Ch'⟩.κε'⟨ρ'⊕{x ↦ ε'}, Ch'⟩
          s' = ⟨ρ⊕{ x ↦ `not_ready`}, Ch⟩
  `otherwise` → `wrong`

`forceFV` :: **Ide → ECont → Cont**

`forceFV` x κ = `force` x κ'
  `where` κ' = λ⟨α,*I*⟩.λs'. $\bigcup_{s''\in S''}$ κ ⟨α,*I*⟩ s''
        S'' = `mforceFV` *I* s'

`mforceFV` :: **Ides → Cont**

`mforceFV` {} = λs.{s}
`mforceFV` ({x} ∪ *I*) = λs. $\bigcup_{s'\in S'}$ `mforceFV` *I* s'
  `where` S' = `forceFV` x id_κ s

Abb. 4. Hilfsfunktionen für die Anweisungssemantik von Eden

3. Die Sprache Glasgow parallel Haskell

Die Sprache Glasgow parallel Haskell (GpH) benutzt zur Umsetzung des Parallelismus wie schon angesprochen eine semi-implizite Vorgehensweise. Dies wird erreicht, indem zwei neue Konstrukte der Sprache Haskell hinzugefügt werden. Dabei handelt es sich um '`par`' und '`seq`'. Mit '`par`' bewirkt man, dass e_1 und e_2 möglicherweise parallel ausgewertet werden. Dabei wird allerdings auf jeden Fall der berechnete Wert von e_2 zurückgegeben. Mit '`seq`' ist es erst möglich e_2 auszuwerten, wenn überhaupt notwendig, nachdem e_1 ausgewertet wurde. In Abb. 5 ist die für die Definition der denotationellen Semantik notwendige GpH Kernsyntax definiert. Dabei handelt es sich wie bei Eden zuerst um die allgemeinen Konstrukten wie ein Bezeichner, die λ-Abstraktion, die bedarfsgesteuerte Applikation und die lokale Definition. Dazu kommen dann noch die für GpH spezifischen Konstrukte der sequentiellen und der parallelen Komposition.

E ::=	x	Bezeichner
|	$\backslash x.E$	λ-Abstraktion
|	x_1 `$` x_2	bedarfsgesteuerte Applikation
|	`let` $\{x_i = E_i\}_{i=1}^{n}$ `in` x	lokale Definition
|	x_1`'seq'`x_2	sequentielle Komposition
|	x_1`'par'`x_2	parallele Komposition

Abb. 5. GpH Kernsyntax

3.1. Semantische Bereiche von Glasgow parallel Haskell

Die notwendigen semantischen Bereiche von GpH werden in Abb. 6 dargestellt. Da GpH weder Prozesse, noch Kommunikationen selbiger zur Verfügung stellt, besteht der Zustand eines Programms nur aus der Umgebung. Die restlichen semantischen Bereiche sind vereinfacht, z.B. sind Ausdruckswerte nur Abstraktionswerte ohne eine Menge von freien Variablen. Zudem besitzt GpH keine Konstrukte, die Nichtdeterminismus erzeugen, weswegen eine Fortsetzung nur eine Zustandsänderung herbeiführt, welche eine Umgebung samt ihrer Argumente in eine neue Umgebung transformiert.

Cont	=	**Env → Env**	Fortsetzungen
$\kappa \in$ **ECont**	=	**EVal → Cont**	Ausdrucksfortsetzungen
$\rho \in$ **Env**	=	**Ide → (Val** $\cup$ {`undefined`})	Umgebungen
$\upsilon \in$ **Val**	=	**EVal** $\cup$ **Clo** $\cup$ {`not_ready`}	Werte
$\varepsilon \in$ **EVal**	=	**Abs**	Ausdruckswerte
$\alpha \in$ **Abs**	=	**Ide → Clo**	Abstraktionswerte
$\nu \in$ **Clo**	=	**ECont → Cont**	closures

Abb. 6. semantische Bereiche von GpH

3.2. Anweisungssemantik von Glasgow parallel Haskell

Die Auswertungsfunktion für die Anweisungen in GpH gleicht der von Eden, nur fehlen die Prozessbezeichner. Sie hat folgende Form:

$$\mathcal{L}: \textbf{Exp} \rightarrow \textbf{ECont} \rightarrow \textbf{Cont}$$

und wird in Abb. 7 genauer definiert.

$\mathcal{L}[\![x]\!]\kappa = \texttt{force}\ x\ \kappa$

$\mathcal{L}[\![\backslash x.E]\!]\kappa = \kappa(\lambda x.\mathcal{L}[\![E]\!])$

$\mathcal{L}[\![x_1\ \$\ x_2]\!]\kappa = \mathcal{L}[\![x_1]\!]\kappa'$
  `where` $\kappa' = \lambda\varepsilon.\lambda\rho.\varepsilon x_2 \kappa\rho$

$\mathcal{L}[\![\text{let } \{x_i = E_i\}_{i=1}^{n} \text{ in } x]\!]\kappa = \lambda\rho.\mathcal{L}[\![x]\!]\kappa\rho'$
  where $\{y_1,\ldots,y_n\}$ = newIde n ρ
    $\rho' = \rho \oplus \{y_i \mapsto \mathcal{L}[\![E_i[y_1/x_1,\ldots, y_n/x_n]]\!] \mid 1 \le i \le n\}$

$\mathcal{L}[\![x_1\text{'seq'}x_2]\!]\kappa = \mathcal{L}[\![x_1]\!]\kappa'$
  `where` $\kappa' = \lambda\varepsilon.\lambda\rho.\mathcal{L}[\![x_2]\!]\kappa\rho$

$\mathcal{L}[\![x_1\text{'par'}x_2]\!]\kappa = \mathcal{L}[\![x_2]\!]\kappa'$
  `where` $\kappa' = \lambda\varepsilon.\lambda\rho.\kappa\varepsilon\rho_{par}$
    $\rho_{par} = \texttt{par}\ x_1\ \rho$

Abb. 7. Auswertungsfunktion für die Anweisungen der GpH-Kernsyntax

Die Auswertungen der Bezeichner, der λ-Abstraktion und der bedarfsgesteuerten Applikation genauso wie die Hilfsfunktion `force` ist dieselbe wie in Eden, nur ist sie um einiges einfacher, da Prozessbezeichner, Mengen von Kanälen und freie Variablen wegfallen.

Die Auswertung der lokalen Definition erzeugt keinen Parallelismus, da bei der Auswertung lediglich die Umgebung mit neuen Variablen, die mit den zugehörigen closures belegt sind, erweitert wird.

Bei der sequentiellen Komposition muss die Reihenfolge der Auswertungen eingehalten werden. Zuerst wird x_1 ausgewertet und erst dann wird x_2 von der Ausdrucksfortsetzung von x_1 ausgewertet. Dazu ist allerdings zu sagen, dass x_2 nur dann ausgewertet wird, wenn die Auswertung von x_1 ein Ausdruckswert ist.

Bei der parallelen Komposition wird x_2 immer ausgewertet, wohingegen die Auswertung von x_1 nur durchgeführt wird, wenn ausreichend Ressourcen zur Verfügung stehen. In einer minimalen Semantik wird x_1 also nicht ausgewertet, während in einer maximalen Semantik x_1 parallel ausgewertet wird. Dies wird dann durch die Funktion `par` aus Abb. 8 ermöglicht.

```
force :: Ide → ECont → Cont
force x κ = λρ.case (ρ x) of
   ε ∈ EVal → κ ε ρ
   ν ∈ Clo → ν κ' ρ'
     where κ' = λε''.λρ''.κε''ρ''⊕{x ↦ ε''}
           ρ' = ρ⊕{x ↦ not_ready}
   not_ready → wrong
endcase

par :: Ide → Cont
par_min x = λρ.ρ
par_max x = λρ.ℒ⟦ x ⟧ id_κ ρ
```

Abb. 8. Hilfsfunktionen für die Anweisungssemantik von GpH

4. Die Sprache parallel Haskell

In diesem Abschnitt wird pH nun als letzte Sprache etwas genauer betrachtet. Id, welches in [Nik91] definiert wird, ist eine parallele Datenflusssprache. Diese Sprache verwendet ein Typsystem, das dem von Haskell sehr ähnlich war. Die Sprache Id wurde dann weiterentwickelt zu pH, indem man die standardisierte Syntax und das Typsystem von Haskell mit der Auswertungsstrategie und den Operatoren für Seiteneffekte von Id verbunden hat. Diese Entwicklung sollte einen Versuch darstellen, die funktionale und Datenfluss Programmierung näher zusammenzubringen. Dadurch ist die Auswertungsstrategie eine Mischung aus Striktheit und Bedarfssteuerung. Für die Sprache pH ist die Einführung von I- und

M-Strukturen zur Synchronisierung charakteristisch. Außerdem wurden for- und while-Schleifenkonstrukte eingeführt, welche allerdings bei der hier vorliegenden Betrachtung wegfallen. Bei der I-Struktur handelt es sich um eine einmalige Zuweisungsstruktur, um Erzeuger-Verbraucher zu synchronisieren. Bei der M-Struktur handelt es sich im Gegensatz dazu um eine mehrfache Zuweisungsstruktur, die einen wechselseitigen Ausschluss modelliert. Durch die M-Struktur werden Seiteneffekte und Nichtdeterminismus eingeführt. Zudem berechnet pH alle reduzierbaren Ausdrücke, die nicht im Rumpf einer Bedingung oder λ-Abstraktion stehen.
Die Abb. 9 stellt die Kernsyntax von pH dar, auf deren Grundlage dann weitergearbeitet wird.

E ::=	x	Bezeichner
|	\x.E	λ-Abstraktion
|	x_1 $ x_2	bedarfsgesteuerte Applikation
|	x_1 $! X_2	strikte Applikation
|	let $\{x_i = E_i\}_{i=1}^{n}$ in x	lokale Definition
|	iCell x | mCell x	Cell-Erstellung
|	Fetch x	Cell-Leseoperator
|	Store(x_1,x_2)	Cell-Schreiboperator

Abb. 9. pH Kernsyntax

Zur Kernsyntax gehören wie auch schon bei den zwei vorherigen Sprachen die Bezeichner, die λ-Abstraktion, die bedarfsgesteuerte Applikation und die lokale Definition. Für pH kommen dann noch die strikte Applikation sowie die Erstellung von Cells und einfache Operationen zum Lesen und Schreiben von Cells.
Wie oben schon erwähnt, zeichnet sich pH durch die Verwendung von I- und M-Cells aus. Bei einer Cell handelt es sich allgemein um die Trennung von Erstellung und Wertzuweisung einer Variablen. Es ist somit erst möglich auf einen Wert zuzugreifen, wenn er berechnet und der Cell zugewiesen wurde, was dann wiederum bedeutet, dass man so lange an der Cell wartet, bis ihr Wert geschrieben wurde. Eine Cell kann mit dem jeweiligen Schlüsselwort iCell oder mCell und einem Bezeichner erstellt werden. Gespeichert wird dann in einer Cell mit iStore oder mStore, was hier allerdings vereinfacht wird zu Store, genau wie es eigentlich iFetch und mFetch heißt. Dabei ist der Parameter x_1 der Name der Cell und x_2 der zu speichernde Wert. Sollte versucht werden in eine gefüllte Cell zu speichern, gäbe es einen Fehler. Genauso wie beim Lesen einer leeren Cell. Eine Cell kann mit dem jeweiligen Befehl `Fetch` gelesen werden. Der große Unterschied zwischen der I- und der M-Cell besteht darin, dass eine I-Cell nach der ersten Belegung nicht mehr verändert, also nur noch gelesen werden kann

im Gegensatz zur M-Cell, bei der nach einmaliger Belegung durch ein `Fetch` die Cell als Seiteneffekt geleert wird und somit wieder ein neuer Wert mit `Store` in ihr gespeichert werden kann. Der Nichtdeterminismus durch Einführung von M-Cells ist dadurch gekennzeichnet, dass bei dem Ausdruck '`let t = mCell m, x = mStore m` v_1`, y = Store m` v_2`, z = mFetch m in z`' entweder der Wert von 'v_1' oder 'v_2' dem Bezeichner '*z*' zugewiesen wird.

4.1. Semantische Bereiche von parallel Haskell

Um eine Fortsetzungssemantik zu definieren ist es notwendig zuerst die semantischen Bereiche von pH darzustellen. Dies ist in Abb. 10 geschehen.

	Cont	=	**Store** → **SStore**	Fortsetzungen
κ ∈	**ECont**	=	**EVal** → **Cont**	Ausdrucksfortsetzungen
σ ∈	**Store**	=	**Loc** → (**Val** ∪ {`undefined`})	Speicher
Σ ∈	**SStore**	=	P_f(**Store**)	Menge von Speicherplätzen
ρ ∈	**Env**	=	**Ide** → **Loc**	Umgebungen
υ ∈	**Val**	=	**EVal** ∪ **Clo** ∪ **Cell** ∪ {`not_ready`}	Werte
ε ∈	**EVal**	=	**Abs** ∪ {`unit`}	Ausdruckswerte
α ∈	**Abs**	=	**Loc** → **Clo**	Abstraktionswerte
ν ∈	**Clo**	=	**ECont** → **Cont**	closures
	Cell	=	{I, M} × (**EVal** ∪ {`empty`})	Cells
l ∈	**Loc**			Speicherplätze

Abb. 10. semantische Bereiche von pH

Ähnlich wie in GpH und anders als in Eden besitzt pH keine Prozesse oder Kommunikationskanäle. Um allerdings das Charakteristikum der Trennung von der Erstellung und Wertbelegung von Cells zu beschreiben, werden bei pH Umgebungen in Verbindung mit Speichern benutzt. Es gibt also eine zweifache Belegungsstruktur. Die Speicherplätze eines Speichers beinhalten entweder einen Wert oder sind undefiniert, während Umgebungen Bezeichner auf Speicherplätze im jeweiligen Speicher abbilden. Deshalb wird der Zustand eines Programms dargestellt durch den globalen Speicher (σ ∈ **Store**). Um der weiter oben erwähnten Möglichkeit des Nichtdeterminismus Rechnung zu tragen, überführt eine Fortsetzung bei pH einen gegebenen Speicher in eine Menge von Speicher.
Wie auch bei den beiden vorherigen Umsetzungen von Parallelität beinhaltet der Bereich der Werte die Ausdruckswerte, closures und den zusätzlichen Wert `not_ready`. Zudem werden

bei pH noch die I-Cells und M-Cells hinzugefügt, die entweder leer sind oder einen Ausdruckswert beinhalten.

Die Abstraktionswerte ($\alpha \in$ **Abs**), die hier Speicherplätze auf closures abbilden, also der Bereich der Ausdruckswerte, werden vervollständigt mit dem Wert `unit`. Dieser Wert ist für Ausdrücke geschaffen worden, die nicht einen Wert produzieren, sondern den Zustand verändern (Seiteneffekte), eine Cell erstellen oder einen Wert in einer Cell speichern.

4.2. Anweisungssemantik von parallel Haskell

Zusätzlich zu der Ausdrucksfortsetzung benötigt die Auswertungsfunktion $\mathcal{L}$ für den pH-Kern eine Umgebung, um die Speicherplätze für die freien Variablen des Ausdrucks zu bestimmen. Die Definition der Funktion wird in Abb. 11 dargestellt und sie hat folgende Signatur:

$$\mathcal{L}: \textbf{Exp} \rightarrow \textbf{Env} \rightarrow \textbf{ECont} \rightarrow \textbf{Cont}$$

$\mathcal{L}[\![x]\!]\rho\kappa$ = `force` $(\rho\ x)\ \kappa$

$\mathcal{L}[\![\backslash x.E]\!]\rho\kappa = \kappa\lambda l.\mathcal{L}[\![E]\!](\rho \oplus \{x \mapsto l\}))$

$\mathcal{L}[\![x_1\ \$\ x_2]\!]\rho\kappa = \mathcal{L}[\![x_1]\!]\rho\kappa'$

`where` $\kappa' = \lambda\varepsilon.\lambda\sigma.$`case` ε `of`

$\varepsilon \in$ **Abs** $\rightarrow \varepsilon l \kappa \sigma'$

`where` $l =$ `freeloc` σ

$\sigma' = \sigma \oplus \{l \mapsto \mathcal{L}[\![x_2]\!]\rho\}$

`otherwise` $\rightarrow$ `wrong`

`endcase`

$\mathcal{L}[\![x_1\ \$!\ x_2]\!]\rho\kappa = \lambda\sigma. \bigcup_{\sigma_2 \in \Sigma_2} \kappa'(\sigma_2\ (\rho\ x_1))\ \sigma_2$

`where` $\Sigma_1 =$ `force` $(\rho\ x_1)$ $id_\kappa\ \sigma$

$\Sigma_2 = \bigcup_{\sigma_1 \in \Sigma_1}$ `force` $(\rho\ x_2)$ $id_\kappa\ \sigma_1$

$\kappa' = \lambda\varepsilon.\lambda\sigma'.$`case` ε `of`

$\varepsilon \in$ **Abs** $\rightarrow \varepsilon\ (\rho\ x_2)\kappa\sigma'$

`otherwise` $\rightarrow$ `wrong`

`endcase`

$\mathcal{L}[\![$ let $\{x_i = E_i\}_{i=1}^{n}$ in x $]\!]\rho\kappa = \lambda\sigma.\mathcal{L}[\![x]\!]\rho'\kappa'\sigma'$

where $\{l_1,\ldots,l_n\}$ = `freeloc` n σ

$\rho' = \rho \oplus \{x_1 \mapsto l_1, \ldots, x_n \mapsto l_n\}$

$\sigma' = \sigma \oplus \{l_1 \mapsto \mathcal{L}[\![E_1]\!]\rho', \ldots, l_n \mapsto \mathcal{L}[\![E_n]\!]\rho'\}$

$\kappa' = \lambda\varepsilon.\lambda\sigma''. \bigcup_{\sigma_d \in \Sigma_d} \kappa\ \varepsilon\ \sigma_d$

where Σ_d = `decls` $\{x_1,\ldots,x_n\}\rho'\sigma''$

$\mathcal{L}[\![$ iCell x $]\!]\rho\kappa = \lambda\sigma.\kappa$ `unit` $(\sigma \oplus \{(\rho\ x) \mapsto \langle I,$ `empty`$\rangle\})$

$\mathcal{L}[\![$ mCell x $]\!]\rho\kappa = \lambda\sigma.\kappa$ `unit` $(\sigma \oplus \{(\rho\ x) \mapsto \langle M,$ `empty`$\rangle\})$

$\mathcal{L}[\![$ Fetch x $]\!]\rho\kappa =$ $\mathtt{in}_{\mathtt{Cont}}$ $(x,\lambda\varepsilon.\lambda\sigma.\kappa\varepsilon\sigma')$

where σ' = `case` $\sigma\ (\rho\ x)$ `of`

$\langle I, \varepsilon'\rangle \rightarrow \sigma$

$\langle M, \varepsilon'\rangle \rightarrow \sigma \oplus \{(\rho\ x) \mapsto \langle M,$ `empty`$\rangle\}$

`otherwise` → `wrong`

`endcase`

$\mathcal{L}[\![$ Store(x_1,x_2) $]\!]\rho\kappa = \mathcal{L}[\![x_2]\!]\rho\kappa'$

where $\kappa' = \lambda\varepsilon.\lambda\sigma.(\mathtt{out}_{\mathtt{Cont}}(x_1,\varepsilon,\kappa$ `unit`$)\sigma')$

σ' = `case` $\sigma\ (\rho\ x_1)$ `of`

$\langle I,$ `empty`$\rangle \rightarrow \sigma \oplus \{(\rho\ x_1) \mapsto \langle I, \varepsilon\rangle\}$

$\langle M,$ `empty`$\rangle \rightarrow \sigma \oplus \{(\rho\ x_1) \mapsto \langle M, \varepsilon\rangle\}$

`otherwise` → `wrong`

`endcase`

Abb. 11. Auswertungsfunktion für die Anweisungen der pH-Kernsyntax

Die Auswertung eines Bezeichners erzwingt die Auswertung des Wertes, der im zugehörigen Speicherplatz abgelegt wurde. Die Hilfsfunktion `force`, die in Abb. 12 definiert wird, ist der in Eden und GpH definierten sehr ähnlich.

Bei der λ-Abstraktion wird der zugehörige Abstraktionswert erzeugt, auf welchen dann die Ausdrucksfortsetzung angewendet wird.

Bei der bedarfsgesteuerten Applikation wird die Auswertung von x_1, welches die Abstraktion liefert, erzwungen. Dabei wird x_2 als closure in einem neuen Speicherplatz abgelegt, um bei Bedarf ausgewertet werden zu können.

Die Auswertung der strikten Applikation ist gleich der bedarfsgesteuerten Applikation in Hinblick auf das Erzwingen der Auswertung von x_1, welches die Abstraktion liefert. Der Unterschied liegt in der Erzwingung der Auswertung von x_2. Es wird bei der strikten Applikation ausgewertet bevor die Applikation angewendet wird.

Um die Auswertung der Cells zu beschreiben, sollte man sich nochmals vor Augen führen, dass es drei verschiedene Operationen gibt. Zum Einen die *Erstellung* einer Cell. Dazu wird

eine leere Zelle in einem Speicherplatz gespeichert, der mit der Variablen verbunden ist. Es wird unterschieden zwischen iCell und mCell. Zum Anderen die *Leseoperation*. Wenn eine Cell leer ist, entsteht beim Lesen immer ein Fehler. Zudem wird beim Lesen des Wertes einer mCell selbige geleert. Zum Schluss gibt es noch die *Schreiboperation*. Hierbei wird der bekommende Wert in der Cell gespeichert. Beim Schreiben auf eine nicht-leere Cell, generiert dieses einen Fehler.

Es gibt eine unübersehbare Parallele des Cell-Konzepts zu dem Konzept der Konstruktion von Kanälen in Eden. Das Speichern in einer Cell ist vergleichbar mit der Übermittlung eines Wertes. Genauso wie erst nach dem Senden eines Wertes auf diesen zugegriffen werden kann, ist es erst möglich auf den Wert einer Cell zuzugreifen, nachdem etwas in ihr gespeichert wurde. Das Zugreifen auf die Cells wird durch die wie in [HI93] beschriebenen üblichen Funktionen out_{Cont} und in_{Cont} durchgeführt.

Die Auswertung der lokalen Definition findet in zwei Schritten statt. Als Erstes wird die Umgebung und der Speicher um die Informationen der neuen lokalen Variablen erweitert. Der zweite Schritt besteht dann aus dem Erstellen paralleler Threads, um jede Variable auszuwerten. Dieses wird dann durch die Hilfsfunktion `decls` realisiert, die in Abb. 12 definiert ist.

`force` :: **Loc → ECont → Cont**
`force` $\alpha\ \kappa = \lambda\sigma.$`case` $(\sigma\ \alpha)$ `of`
  $\varepsilon \in$ **Abs** $\rightarrow \kappa\ \varepsilon\ \sigma$
  $\nu \in$ **Clo** $\rightarrow \nu\ \kappa'\ \sigma'$
    `where` $\kappa' = \lambda\varepsilon''.\lambda\sigma''.\kappa\varepsilon''\sigma'' \oplus \{l \mapsto \varepsilon''\}$
      $\sigma' = \sigma \oplus \{l \mapsto$ `not_ready`$\}$
  `otherwise` → `wrong`
`endcase`

`decls` :: P_f**(Ide) → Env → Cont**
`decls` $\{\}\ \rho = \lambda\sigma.\{\sigma\}$
`decls` $I\ \rho = \lambda\sigma. \bigcup_{x \in I} \left(\bigcup_{\sigma_x \in \Sigma_x}\right.$ `decls` $(I \setminus \{x\})\rho\sigma_x)$
  `where` $\Sigma_x = \mathcal{L}[\![x]\!]\ \rho\ \text{id}_\kappa\ \sigma$

Abb. 12. Hilfsfunktionen für die Anweisungssemantik von pH

Das Beenden eines rechnenden Programms ist in pH nur möglich, wenn jeder Thread vollständig ausgewertet wurde. Dazu ist es nicht notwendig zwischen einer minimalen und einer maximalen Semantik zu unterscheiden. Trotz dessen ist die hier gewählte Form die der maximalen Semantik.

5. Schlussbemerkung

In der mir zu Grunde liegenden Arbeit wurde der Versuch unternommen Fortsetzungen zu benutzen, um Bedarfssteuerung, Parallelität, Nichtdeterminismus und Seiteneffekte in einer denotationellen Semantik zu modellieren.

Der Unterschied zwischen den drei Umsetzungsmöglichkeiten (implizit, semi-implizt, explizit) für Parallelität schlägt sich in den semantischen Bereichen der jeweiligen Sprachen nieder. Nur im Falle des expliziten Parallelismus war eine zusätzliche Notation für Prozesse nötig. Darüber hinaus verkompliziert das System der Verteilung von Eden die Semantik etwas, weil Belegungen der Variablen von einem Prozess zu einem Anderen kopiert werden müssen, während bei den beiden anderen Sprachen keine Einschränkungen in dieser Hinsicht durch die gemeinsame Nutzung eines Arbeitsspeichers bestehen.

Der explizite Parallelismus von Eden benötigt besondere Bereiche für Prozesse. Dazu gehören zum Einen die Prozessbezeichner (IdProc) und zum Anderen die Menge der Kanäle (SChan). Diese beiden Bereiche werden von GpH und pH nicht benötigt. Es sind allerdings nicht nur die Bereiche, in denen die Programmiersprachen voneinander abweichen. Die Semantik der Funktionen ist ein weiteres Unterscheidungsmerkmal. Der explizite Parallelismus von Eden liegt in der Auswertung der parallelen Applikation (`$#`), bei der ein neuer Prozess und dazu eine Struktur erstellt werden, um z.B. mit Anderen über Kanäle zu kommunizieren. Ein weiteres eindeutiges Unterscheidungsmerkmal ist die Auswertung der lokalen Definition. Während GpH nur die Umgebung aktualisiert und die Variablen somit nur bei Bedarf ausgewertet werden, ist es in pH hingegen so, dass die lokalen Variablen parallel ausgewertet werden, während gleichzeitig der Ausdruck, welcher die lokale Definition aufruft, weiterläuft. Dazu wertet Eden im Fall einer minimalen Semantik nur die Variablen aus, die in Verbindung zur stattfindenden Prozesserstellung stehen. Diese Aufgabe wird durch die Ausdrucksfortsetzung bewältigt.

Wie man leicht erkennen kann, hat GpH die einfachste Semantik. Es gibt keine Prozesse, Kommunikation, keinen Nichtdeterminismus und keine Seiteneffekte. Da diese Semantik so einfach ist, lässt sich mit ihr leicht herausfinden, was nun wirklich parallel ausgeführt wird und was nicht, indem man die endgültige Umgebung unter der minimalen und maximalen Semantik betrachtet. Dies kann bei GpH auch ohne Fortsetzungen durchgeführt werden.

Die Semantiken, die in der mir zu Grunde liegenden Arbeit definiert wurde, ermöglichen es, den Grad an Parallelität und die Menge von vermuteten Berechnungen zu bestimmen. Für Eden z.B. definiert eine Menge von Kanälen einen gerichteten Graphen, dessen Knoten mit den Prozessbezeichnern und die Kanten mit den übermittelten Werten beschriftet werden.

Daher sind die Zahl der Knoten des Graphen und die Zahl der Prozesse des zugehörigen Systems gleich. Durch eine Änderung an der Definition der Ausdrucksfortsetzung für die parallele Applikation wird es möglich, auch andere Ansätze zwischen minimaler und maximaler Semantik in diesem Rahmen zu realisieren. Die Spekulation über den vermuteten Grad an Parallelität kann man in diesem Zusammenhang als Differenz der Anzahl der Knoten des minimalen und des nicht-minimalen Graphen sehen.

Bei pH Programmen handelt es sich bei einem spekulativen Speicherplatz um einen, der nicht benötigt wurde um ein Ergebnis zu speichern. Es könnte auch eine alternative minimale Definition für Ausdrucksfortsetzungen der lokalen Definition angegeben werden, welche die Abschlüsse (closures) an den Speicher übergibt, allerdings dabei nicht auswertet. Vergleicht man den endgültigen Speicher aus Kapitel 4 mit dem nun Definierten, so weiß man, dass die nun nicht vorhandenen Speicherplätze spekulativ sind, genauso wie Speicherplätze, die zwar in beiden Definitionen deklariert wurden, aber in der nicht minimalen Semantik mit einem Ausdruckswert oder einer Cell belegt sind, im Gegensatz zur Belegung mit einem Abschluss (closure) in der minimalen Semantik.

Betrachtet man die Spekulationen in GpH, so benutzt man zur Analyse dieselben Ideen, wie zuvor in pH. Dabei müssen dann allerdings die Speicherplätze im Speicher durch Variablen in der Umgebung ersetzt werden.

Andererseits lässt es das Abstraktionsniveau der präsentierten denotationellen Modellierung nicht zu, die Verdopplung von Arbeit zu bemerken, welche beim Kopieren von Variablen von einem Prozess zu einem Anderen entstehen kann.

Eine der, in der mir zu Grunde liegenden Arbeit, angesprochenen zukünftigen Aufgaben ist die Nutzung der erarbeiteten denotationellen Semantik um die behandelten Sprachen formaler fassen zu können. Als Beispiel wurde genannt, dass angedacht war, das Verhalten der Kommunikationskanäle in Eden mit dem Cell-Konzept in pH zu vergleichen. Dies ist von Interesse, da es sich bei einer iCell um eine dem Kanal ähnliche Konstruktion handelt. Bei der iCell kann ein Thread den Wert schreiben, welchen andere Threads dann später lesen können. Ist der Wert einer iCell erstmal gesetzt, dann kann dieser nicht mehr verändert werden. Ähnlich ist es bei einem Kanal, der nur einmal benutzt werden kann, wobei allerdings genau ein Leser eindeutig bestimmt ist.

Literaturverzeichnis

[AAAM+95] S. Aditya, Arvind, L. Augustsson, J. Maessen, R. S. Nikhil. Semantics of pH: A parallel dialect of Haskell. In P. Hudak, Herausgeber, Haskell Workshop, Seiten 35-49, La Jolla, Cambridge, MA, USA, YALEU/DCS/RR-1075, Juni 1995.

[AAAH+95] S. Aditya, Arvind, L. Augustsson, J. Hicks, J. Maessen, R. S. Nikhil, Y. Zhou. pH Language Reference Manual, Version 1.0 - preliminary. Technical Report CSG Memo 369, Laboratory for Computer Science, MIT, Cambridge, MA, USA, 1995.

[BFKT00] C. Baker-Finch, D. King, P. Trinder. An operational semantics for parallel lazy evaluation. In ACM-SIGPLAN International Conference on Functional Programming (ICFP'00), Seiten 162-173, Montreal, Canada, September 2000.

[BLOP95] S. Breitinger, R. Loogen, Y. Ortega-Mallén, R. Peña-Marí. Towards a Declarative Language for Parallel and Concurrent Programming. In Glasgow Workshop on Functional Programming, Workshops in Computing, Springer, 1995.

[BLOP96] S. Breitinger, R. Loogen, Y. Ortega-Mallén, R. Peña-Marí. Eden - The Paradise of Functional Concurrent Programming. In EUROPAR'96: European Conference on Parallel Processing, pages 710-713. LNS 1123, Springer, 1996.

[BLOP98] S. Breitinger, R. Loogen, Y. Ortega-Mallén, R. Peña-Marí. Eden - Language Definition and Operational Semantics, Bericht 96-10, Reihe Informatik FB Mathematik, Universität Marburg, 1998.

[DB97] M. Debbabi, D. Bolognano. ML with Concurrency: Design, Analysis, Implementation and Application, Kapitel 6: A semantic theory for ML higher-order concurrency primitives, Seiten 145-184. Monographs in Computer Science. Ed. F. Nielson. Springer, 1997.

[Em02] M. van Ekelen, M. de Mol. Reasoning about explicit strictness in a lazy language using mixed lazy/strict semantics. In Draft Proceedings of the 14th International Workshop in Implementation of Functional Languages, IFL'02, Seiten 357-373. Dept. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, 2002.

[FH99] W. Ferreira, M. Hennessy. A behavioral theory of first-order CML. Theoretical Computer Science, 216:55-107, 1999.

[Pey03] S. Peyton Jones. Haskell 98 language and libraries: the Revised Report. Cambridge University Press, 2003.

[Hen88] M. Hennessy. Algebraic Theory of Processes. MIT Press, 1988.

[HI93] M. Hennessy, A. Ingólfsdóttir. A theory of communicating processes with value passing. Information and Computation, 107:202-236, 1993.

[HOM02] M. Hidalgo-Herrero, Y. Ortega-Mallen. A operational semantics for the parallel language Eden. Parallel Processing Letters. World Scientific Publishing Company, 12(2):211-228, 2002.

[Hud86] P. Hudak. Para-functional programming. IEEE Computer, 19:60-71, 1986.

[Jos86] M. B. Josephs. Functional programming with side-effects. PhD thesis, Oxford University, 1986.

[Jos89] M. B. Josephs. The semantics of lazy functional languages. Theoretical Computer Science, 68:105-111, 1989.

[KM77] G. Kahn, D. MacQueen. Coroutines and networks of parallel processes. In IFIP'77, Seiten 993-998.Herausgeber B. Gilchrist. North-Holland, 1977.

[Lau93] J. Launchbury. A natural semantics for lazy evaluation. In POPL'93, Charlston, 1993.

[LOO99] R. Loogen. Research Directions in Parallel Functional Programming, Kapitel 3: Programming Language Constructs. Herausgeber K. Hammond, G. Michaelson. Springer, 1999.

[NA01] R. S. Nikhil, Arvind. Implicit Parallel Programming in pH. Academic Press, 2001.

[Nik91] R. S. Nikhil. Id (version 90.1) language reference manual. Technical Report CSG Memo 284-2, Laboratory for Computer Science, MIT, Cambridge, MA, USA, 1991.

[Pey87] S. Peyton Jones. Implementation of Functional Programming Languages. Prentice Hall, 1987.

[Rep92] J. H. Reppy. Higher-Order Concurrency. PhD thesis, Cornell University (Department of Computer Science), 1992.

[Rey98] J. C. Reynolds. Theories of Programming Languages. Cambridge University Press, 1998.

[Stoy77] J. E. Stoy. Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press, Cambridge, MA, 1977.

[THM+96] P. Trinder, K. Hammond, J. Mattson Jr., A. Partridge, S. Peyton Jones. GUM: a portable implementation of Haskell. In Proceedings of Programming Language Design and Implementation, Philadelphia, USA, Mai 1996.