Philipps-Universität Marburg
Fachbereich Mathematik und Informatik
Seminararbeit
Seminar: *Konzepte von Programmiersprachen*
Betreuerin: Prof. Dr. Rita Loogen

Mixed Lazy/Strict Graph Semantics

Beatriz Guadalupe De La Rosa Mesino
Studienfach: Informatik
Sommersemester 2005

Inhaltsverzeichnis

1. Einleitung

Diese Seminararbeit ist im Rahmen des Seminars „Konzepte von Programmiersprachen“ entstanden. Ziel dieser Arbeit ist es, den Ansatz „Mixed Lazy/Strict Graph Semantics“[1] von Marko van Eekelen und Maarten de Mol zu erläutern.
Während der Ausarbeitung dieses Ansatzes erwies es sich allerdings als notwendig sich mit einem vorherigen Ansatz – „A Natural Semantics for Lazy Evaluation“ [1] von John Launchbury – auseinanderzusetzen. Der Grund hierfür war, dass dieser Ansatz als Grundlage für den oben genannten Ansatz diente.

Somit ergibt sich für vorliegende Seminararbeit folgende Struktur: In dem ersten Kapitel wird die Grundlage des Ansatzes von Launchbury kurz skizziert und wichtige Fachbegriffe werden erläutert, wie beispielweise die Motivation des Entwicklers, die Definition und die Eigenschaften der Semantik, die Reduktionsregeln und die Erklärung derselben anhand eines Beispiels.

Das Haupthema dieser Seminarabeit ist die Ausarbeitung des Ansatzes „Mixed Lazy/Strict Graph Semantics“, wo eine lazy-strikte Semantik definiert wird und die Kenntnisse – die so genannten „folklore“-Kenntnisse, die Programmentwickler über Striktheit haben –, formalisiert werden, da sie allein aufgrund von Erfahrungen gewonnen wurden. In diesem zweiten Kapitel wird außerdem anhand eines Beispiels erläutert, wie mit der lazy-strikten Semantik Beweise durchgeführt werden können.

Abschließend werde ich versuchen, eine Zusammenfassung der gewonnen Kenntnisse zu liefern.

[1] Bemerkung: Die Theoreme werden hier nicht noch einmal bewiesen. Die Beweise sind in den Originaldokumenten ausführlich dargelegt.

2. A Natural Semantics for Lazy Evaluation

In dem folgenden Kapitel werden die für ein besseres Verständnis dieser Arbeit relevanten Konzepte bzw. Grundlagen kurz skizziert, definiert und erläutert. Diese sollen hierbei nicht ausführlich und umfassend beleuchtet, sondern nur in dem für das Verständnis dieser Arbeit notwendigen Umfang dargelegt werden.

2.1 Lazy Evaluation

Bei der Lazy Evaluation handelt es sich um eine nicht-strikte Auswertung von Ausdrücken. Sie wird benutzt, um nicht-strikte Funktionen zu implementieren. Ziel dieser Auswertungstechnik ist es, unnötige Reduktionen bzw. Auswertungen zu vermeiden, weshalb die Ausdrücke auf ihre WHNF (weak head normal form) reduziert werden.
Mit Lazy Evaluation werden die Ausdrücke nur dann und soweit ausgewertet, wie es notwendig ist, das heißt, dass nur die Teile eines Ausdruckes ausgewertet werden, die für ein Ergebnis notwendig sind.
Mit Lazy Evaluation wird außerdem versichert, dass ein Ausdruck nur einmal ausgewertet wird. Dies ermöglicht es, eine Programmiersprache mit unendlichen Datenstrukturen zu implementieren, da nur bestimmte Teile dieser Datenstrukturen ausgewertet werden. Außerdem ist es dadurch möglich, dass bestimmte Reduktionen wiederverwendet (shared) werden.

2.2 Warum eine neue Semantik für Lazy Evaluation?

Bevor die Details dieses Ansatzes vertieft werden, ist es notwendig, die Vorgeschichte bzw. die vorhergehenden Ansätze, in denen auch eine Semantik für Lazy Evaluation definiert wurden, kurz zu erläutern:

Einige dieser Ansätze, wahrscheinlich auch die berühmtesten, sind die von Abramsky und Ong. Sie bieten eine bedeutende Theorie, die auf einem sehr hohen Abstraktionsniveau definiert wurde, sich auf die semantische Bedeutung der Lazy Evaluation konzentrierte und die theoretischen Auswirkungen der Behandlung von λ-Ausdrücken in WHNF als Werte untersuchte.

Es wurde außerdem eine operationelle Semantik entwickelt, die für die Definition des Verhaltens von abstrakten Machinen, wie z.B die Machinen G, STG, TIM und TIGRE, benutzt werden kann.

Ein anderer Versuch, eine Semantik für Lazy Evaluation zu definieren, ist die Semantik von Josephs. Diese Semantik berücksichtigt sowohl die Umgebung als auch die Speicherung der Ausdrücke. Damit wurde die notwendige Vorrichtung für die Modellierung von imperativen Sprachen mit „gotos“ angeboten.
Josephs' Semantik benötigt zusätzlich eine zwingende Funktion, um den Umfang der Auswertung kontrollieren zu können.

Andere Ansätze, die für die Entwicklung einer Semantik für Lazy Evaluation einen Beitrag geleistet haben, sind z.B die Ansätze von Purushothaman und Seaman, die eine operationelle Semantik entwickelt haben, oder auch der Ansatz von Maranget, dem eine Studie des λ-Kalkül zugrunde liegt, die mit Hilfe der Lazy Evaluation durchgeführt wurde, wobei sich die Lazy Evaluation als eine gute Strategie erwiesen hat.

Allerdings haben all diese Ansätze, obwohl sie die Grundlagen gelegt und darüber hinaus zur Entwicklung viel beigetragen haben, verschiedene Aspekte ausgelassen, die bei der Lazy Evaluation auf jeden Fall berücksichtigt werden müssen. Diese Aspekte sind zum Beispiel die Wiederverwendung (Sharing) von Ausdrücken bzw. Reduktionen: Die Semantik muss einfach entwickelt werden, so dass mit dieser verschiedene Beweise durchgeführt werden können; die Gestaltung der Semantik muss nicht so ausführlich sein, sodass deren Ergebnisse für verschiedene abstrakte Maschinen verwendbar sein können. Sie muss rekursiv sein, die Lets müssen in der Semantik berücksichtigt werden, da diese eine sehr große Rolle in jeder funktionalen Programmiersprache spielen.

2.3 Ziel des Ansatzes

Wie schon zuvor erwähnt gibt es verschiedene Ansätze, in denen eine Semantik für die Lazy Evaluation entwickelt wurde. Allerdings erfüllen diese leider nicht alle Aspekte. Mit dieser Motivation und mit dem Interesse zu wissen, wie sich ein ausgewerteter bzw. reduzierter Ausdruck während der Lazy Evaluation verhält, wurde dieser der Arbeit zugrunde liegende Ansatz entwickelt.

Somit war das Ziel dieses Ansatzes eine Semantik für Lazy Evaluation zu entwickeln, die ein Modell für die Wiederverwendung (Sharing) von Ausdrücken bietet, die zugänglich für die Durchführung von Beweisen, rekursiv und vor allem einfacher als die vorgehenden Semantiken ist, was heißt eine operationelle Semantik zwischen einer denotationellen Semantik und einer operationellen Semantik von einer abstrakten Maschine zu entwickeln, wobei nur ein Heap[2] als Datenstruktur eingeführt wird.

[2] Ein Heap ist eine partielle Funktion von Variablen zu Ausdrücken

2.4 Konventionen der Semantik

Die lazy Semantik ist durch die folgenden Konventionen definiert:

- Variable x, y, x_1, x_n
- Ausdrücke e, e′, e_1, e_n
- Werte z, z′
- α - Konversion $\hat{z}$
- Heapvariable Γ, Δ, Θ
- Bindungen $x \mapsto e$ [3]
- Beurteilung $\Gamma : e \Downarrow \Delta : z$ [4]
- Umgebung ρ, ρ', ρ_0 [5]

2.5 Methodik

Mit dieser Semantik wird Laziness nun in zwei Phasen erreicht: die statische Umwandlung der λ-Ausdrücke und die dynamische Semantik der umgewandelten λ-Ausdrücke.

[3] Die Variable x wird an den Ausdruck e gebunden

[4] Eine Beurteilung bedeutet, dass der Term in dem Kontext des Heaps Γ auf z mit der Reihe von Bindungen
Δ reduziert wird

[5] Eine Umgebung ist eine Funktion von Variablen zu Werten

✓ In der ersten Phase handelt es sich um eine statische Umwandlung der λ-Ausdrücke in einer regulären Form, das heißt die λ-Ausdrücke werden in einer eingeschränkten Syntax normalisiert. Diese Umwandlung führt zu normalisierten λ-Ausdrücken der Form:

$$
\begin{aligned}
& x \in Var \\
& e \in Exp ::= \lambda x.e \\
& \quad | \quad e\ x \\
& \quad | \quad x \\
& \quad | \quad \text{let } x_1 = e_1, \ldots, x_n = e_n \text{ in } e
\end{aligned}
$$

Die Umwandlung der Ausdrücke wird in zwei Phasen durchgeführt. In der ersten Phase handelt es sich um eine α-Konversion, die besagt, dass die Namen der gebundenen Variablen unwichtig sind, das heißt, dass alle gebundenen Variablen in e durch neue Variablen ersetzt werden. Ein Beispiel für eine α-Konversion ist:

$$\lambda x. * 2x \leftrightarrow_{\alpha} \lambda y. * 2y$$

Die zweite Phase der Umwandlung bewirkt, dass die Argumente der Funktionen immer Variablen sind.

$$
\begin{aligned}
& (\lambda x.e)^* = \lambda x.(e^*) \\
& \quad x^* = x \\
& (\text{let } x_1 = e_1, \ldots, x_n = e_n \text{ in } e)^* \\
& \qquad = \text{let } x_1 = (e^*_1), \ldots, x_n = (e^*_n) \text{ in } (e^*) \\
& (e_1\ e_2)^* = (e^*_1)\ e_2 \qquad \text{wenn } e_2 \text{ eine Variable ist} \\
& \qquad = \text{let } y = (e^*_2) \text{ in } (e^*_1)\ y \qquad \text{sonst}
\end{aligned}
$$

✓ Die zweite Phase besteht aus einer dynamischen bzw. operationellen Semantik für die normalisierten λ-Ausdrücke. Hierbei wird folgende Konvention benutzt:

$$\Gamma, \Delta, \Theta \in \text{Heap} = \text{Var} \rightarrow \text{Exp}$$

$$z \in \text{Val} ::= \lambda x.e$$

Hierbei wird ein Heap (Γ, Δ, Θ) als eine ungeordnete Menge von Variablen bzw. Ausdrücken aufgefasst, der verschiedene Variablennamen um Ausdrücke bindet und einen Wert (Val) als einen Ausdruck in whnf betrachtet.

Nachdem die λ-Ausdrücke normalisiert wurden, können sie nun reduziert werden. Dafür werden verschiedene Reduktionsregeln benutzt. Diese Regeln werden nun im Folgenden erklärt.

2.5.1 Reduktionsregeln (Operationelle Semantik)

- Lambda- Regel

$$\Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e$$

Diese Regel besagt, dass die Ausdrücke, deren äußerste Komponente ein λ ist, wieder als solche geschrieben werden, ohne den Heap zu ändern, das heißt, dass diese Ausdrücke schon in whnf sind und deshalb keiner Auswertung mehr bedürfen.

- Applikation

$$\frac{\Gamma : e \Downarrow \Delta : \lambda y.e' \quad \Delta : e'[x/y] \Downarrow \Theta : z}{\Gamma : e\, x \Downarrow \Theta : z}$$

Hier wird der linke Term der Applikation reduziert, ersetzt das Argument in der λ-Variable und setzt die Reduktion fort.

- Variable

$$\frac{\Gamma : e \Downarrow \Delta : z}{(\Gamma, x \mapsto e) : x \Downarrow (\Delta, x \mapsto z) : \hat{z}}$$

Damit eine Variable x im Kontext eines Heaps ausgewertet werden kann, muss dieser Heap eine Bindung der Form $x \mapsto e$ haben. Wenn das der Fall ist, wird e ohne die Referenz zu x reduziert. Falls diese Reduktion eine WHNF z gibt, wird ein neuer Heap mit der Bindung $x \mapsto z$ erweitert und eine umbenannte Version von z ist das Ergebnis. Bei dieser Regel wird die Wiederverwendung (Sharing) der Ausdrücke ausgeführt.

- Let

$$\frac{(\Gamma, x_1 \mapsto e_1 \cdots x_n \mapsto e_n) : e \Downarrow \Delta : z}{\Gamma : let\ x_1 = e_1 \cdots x_n = e_n\ in\ e \Downarrow \Delta : z}$$

Die Bindungen können in dem Heap addiert werden, ohne dass es einen Namenskonflikt gibt.

2.5.2 Denotationelle Semantik

Die denotationelle Semantik ist in einem Definitionsbereich über mathematische Funktionen definiert. Solches Modell ermöglicht es zwischen definierten und undefinierten Ausdrücken zu unterscheiden.
Außerdem existiert eine Funktion, die als „Umgebung“ bezeichnet wird, mit der die Variablen auf Werten abgebildet werden. Diese Funktion ist definiert als:

$$\rho \in Env = Var \to Value$$

Die initialisierende Umgebung, ρ_0, ist eine Funktion, die alle Variablen auf $\perp$ abbildet.
Die „Bedeutung“ der Ausdrücke ist durch die semantische Funktion $[\![-]\!] : Exp \to Env \to Value$ definiert.

$$\begin{aligned}
[\![\lambda x.e]\!]_\rho &= Fn\left(\lambda v.[\![e]\!]_{\rho\ (x \mapsto v)}\right) \\
[\![e\ x]\!]_\rho &= \left([\![e]\!]_\rho\right)\downarrow_{Fn}\left([\![x]\!]_\rho\right) \\
[\![x]\!]_\rho &= \rho(x) \\
[\![let\ x_1 = e_1 \cdots x_n = e_n\ in\ e]\!]_\rho &= [\![e]\!]_{\{\{x1 \mapsto e1 \cdots xn \mapsto en\}\}\rho}
\end{aligned}$$

Hierbei definiert die semantische Funktion $\{\{ - \}\} : Heap \to Env \to Env$ eine rekursive Umgebung, in der die Rekursivität der Let-Ausdrücke aufgenommen wird. F_n und $\downarrow_{Fn}$ sind Funktionen, die als „Lifting“ und „Projektion“ bezeichnet werden. Das bedeutet, dass die Undefiniertheit ($\perp$) in die Umgebung hinzugefügt und eliminiert wird. Die semantische Funktion ist definiert durch:

$$\{\{x_1 \mapsto e_1 \cdots x_n \mapsto e_n\}\}_\rho = \mu\rho'.\rho\ \left(x_1 \mapsto [\![e_1]\!]_{\rho'\ \ xn \mapsto [\![en]\!]\rho'}\right)$$

wo μ den kleinsten Fixpunkt bezeichnet

2.6 Eigenschaften der Semantik

Wie schon erwähnt wurde, müssen die Ausdrücke normalisiert werden, bevor sie reduziert bzw. ausgewertet werden können. Deshalb ist eine Auswertung notwendig, um zu gewährleisten, dass die Eigenschaften der normalisierten Ausdrücke während der Reduzierung bzw. Auswertung erhalten bleiben. Diese Eigenschaften sind:

- Verschiedene Namen:

Diese Eigenschaft besagt, dass alle Bezeichner paarweise veschiden sind, wenn jede Bindung in Γ und e verschiedene Variablen binden.

- Korrektheit:

Die Bedeutung der Ausdrücke bleibt bei den Reduktionen erhalten. Diese ändert sich nur dann, wenn neue Bindungen dem Heap zugefügt werden.

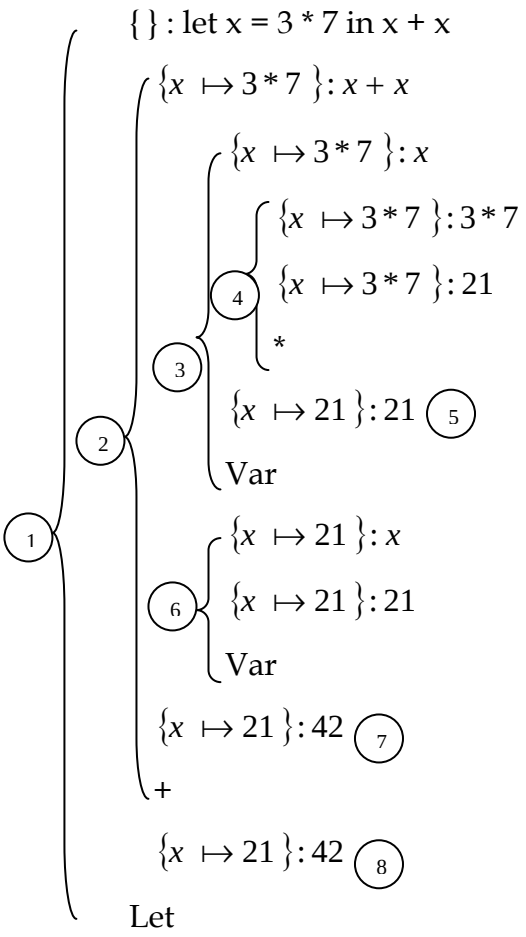
$$[\![e]\!]_{\{\{\Gamma\}\}\rho} = [\![z]\!]_{\{\{\Delta\}\}\rho} \wedge \{\{\Gamma\}\}_\rho \leq \{\{\Delta\}\}_\rho$$

- Rechnerische Angemessenheit:

Der Heap bzw. die Ausdrücke sind nur dann reduzierbar, wenn ihre Denotation definiert ist.

$$[\![e]\!]_{\{\{\Gamma\}\}\rho_0} \neq \bot \Leftrightarrow \left(\exists \Delta, z.\Gamma : e \Downarrow \Delta : z\right)$$

2.7 Beispiel



1. Der Ausdruck let x = 3 * 7 in x + x wird im Kontext von einem leeren Heap ausgewertet. Da es sich um einen Let-Ausdruck handelt, wird als Erstes mit der Let-Reduktionsregel gearbeitet. (x_1 = x, e_1 = 3 * 7, e = x + x)

2. So wird der Heap mit der Bindung der Form $\{x \mapsto 3 * 7\}$ erweitert. Als nächstes wird e ausgewertet. Hierbei wird die arithmetische Additionsregel angewendet. Dafür wird der Wert der Variablen x benötigt.

3. Da x eine Variable ist, wird nun die Variablen-Reduktionsregel benutzt. Um diese Reduktionsregel anwenden zu können, muss der Heap eine Bindung der Form $x \mapsto e$ haben. Da es der Fall ist, nämlich $\{x \mapsto 3 * 7\}$, wird nun 3 * 7 (e) ausgewertet.

4. Hierbei wird die arithmetische Multiplikationsregel angewendet. Der neue Wert ist nun 21.

5. Der Heap wird nun mit diesem neuen Wert erweitert.

6. Hier wird noch einmal die Variablenreduktionsregel angewendet, da x bereits ausgewertet wurde, wird dieser Wert nun wiederverwendet (shared), das heißt x wird nicht noch einmal ausgewertet.

7. Die Addition wird vollständigt.

8. Das Ergebnis ist 42 zusammen mit einem Heap, wo x zu 21 gebunden ist.

3. Mixed Lazy/Strict Graph Semantics

Wie bereits in der Einführung erwähnt wurde, wird in diesem Kapitel der Ansatz „Mixed Lazy/Strict Graph Semantics“ erläutert.
In diesem Ansatz wird eine operationelle und denotationelle Semantik für die Implementierung von Striktheit entwickelt, die auf dem Ansatz von Launchbury „A Natural Semantics for Lazy Evaluation“ basiert. Launchburys Ansatz bzw. Launchburys Semantik wird also zu einer lazy-strikten Semantik erweitert.

3.1 Funktionale Programmierung

Programmiersprachen haben sich im Laufe der Zeit stark verändert. Während bei den ersten Programmiersprachen der Rechner im Vordergrund stand – d.h die Befehle wurden mit Operationen des Rechners durchgeführt –, konzentrieren sich die neuen Programmiersprachen nur auf Operationen und wie diese durchgeführt werden. Diese Sprachen sind die so gennanten „höheren Programmiersprachen“. Zu diesen gehören auch die *funktionalen Programmiersprachen*.
Ein *funktionales Programm* beschreibt auf eine direkte Art, wie die Daten analysiert, verarbeitet und wieder zusammengefügt werden[6].

Funktionale Programmierung heißt „beschreibend“ programmieren. Es ist eine Art zu programmieren, in der die Verwendung von Funktionen hervorgehoben wird. Ausdrücke werden zu Werten ausgewertet. Mit der funktionalen Programmierung werden die Objekte allein nach ihrem Wert definiert. Die funktionale Programmierung ist in den letzten Jahren sehr beliebt geworden, da diese Art zu programmieren sehr leicht, präzise und deutlich ist.[7]

[6] Vgl. Thompson 1996: 5.
[7] Vgl. Hudak 2000: 1.

3.2. Striktheit

Striktheit ist eine Eigenschaft von Funktionen. Eine Funktion ist *strikt*, wenn ihr Ergebnis undefiniert ist, sobald eines ihrer Argumente undefiniert ist. Dass eine Funktion in einem ihrer Argumente undefiniert ist, kann bedeuten, dass die Auswertung in eine unendliche Schleife gerät.

Strikte Funktionen werden wie im Folgenden bezeichnet:

$$f \perp = \perp$$

Hierbei wird das Symbol $\perp$ als Bezeichnung für die „Undefiniertheit“ einer Funktion geschrieben. Dies ist allerdings kein Fachausdruck der Programmiersprachen. Wenn ein Argument einer Funktion undefiniert ist, bedeutet es, dass sowohl das Argument als auch die Applikation der Funktion keine Normalform haben.

Striktheit wird in der funktionalen Programmierung als Werkzeug implementiert, um zu versichern, dass die Applikationen die Zeit- und Platzanforderungen erfüllen.

In der funktionalen Programmierung wird Striktheit benutzt um:

- die Leistung der Datenstrukturen zu verbessern
- die Leistung der Auswertungen zu verbessern
- die Auswertungordnung zu modifizieren

3.3 Begründung für die Entwicklung dieses Ansatzes

Die Kenntnisse über Striktheit sind zwar viele und verschieden, basieren aber nur auf den Erfahrungen von Programmentwicklern und nicht auf ausführlichen Beweise, das heißt dass nur wenige Programmentwickler sich über die Konsequenzen der Ausführung von Striktheit in ihren Programmen bewusst sind.

In der funktionalen Programmierung dient die Striktheitsanalyse, um zu versichern, dass die Striktheit in den Eigenschaften der Programme berücksichtigt und erhalten wird.
Die Striktheitsanalyse spielt bei der Entwicklung von Programmen eine große Rolle, da ohne ihre Berücksichtigung die Programme bzw. Teile der Programme nicht korrekt funktionieren können.

Es gibt allerdings kein ofizielles Modell über die Implementierung von Striktheit, weder in der operationellen noch in der denotationellen Semantik. Dadurch wird es erschwert lazy- und strikte- Programme zu analysieren.

3.4 Ziel des Ansatzes

Mit der Motivation ein offizielles Modell für die Implementierung von Striktheit zu entwicklen und mit dem Interesse die verschiedenen Kenntnisse über diese zu normen, wurde dieser Ansatz entwickelt.

Somit war das Ziel dieses Ansatzes eine passende sowohl operationelle, als auch denotationelle Semantik für die korrekte Ausführung von Programmen in einem lazy-strikten Kontext zu entwicklen.

3.5 Konventionen der Semantik

Die Konventionen der lazy-strikten Semantik bleiben erhalten wie bei der Semantik für Lazy Evaluation.[8]

3.6 Methodik

Die Vorgehensweise mit der lazy-strikten Semantik ist dieselbe wie mit der Semantik für Lazy Evaluation. Allerdings werden die Ausdrücke um eine strikte Variante von rekursiven Let-Ausdrücken erweitert.[9]

$$
\begin{aligned}
& x \in Var \\
& e \in Exp ::= \lambda x.e \\
& \qquad \vdots \\
& \qquad \vdots \\
& \qquad \mid \text{let } !\ x_1 = e_1 \text{ in } e
\end{aligned}
$$

Bei diesen strikten Let-Ausdrücken kann nur eine Variable definiert werden, im Kontrast zu der Semantik für Lazy Evaluation, wo vielen Variablen definiert werden.

Die Methodik[9] des Launchburys Ansatzes bleibt in diesem Ansatz erhalten, die Ausdrücke müssen zuerst normalisiert werden, bevor die operationelle lazy-strikte Semantik definiert wird, das heißt, dass die Semantik nur auf normalisierten Termen definiert wird.

$$(\text{let! } x_1 = e_1 \text{ in } e)^* = \text{let! } x_1 = (e_1)^* \text{ in } (e)^* \quad (11)$$

[8] siehe Kapitel 2.4 dieser Arbeit.
[9] siehe Kapitel 2.5 dieser Arbeit

3.6.1 Reduktionsregeln

Mit der lazy-strikten Semantik bleiben die Reduktionsregeln der Semantik für Lazy Evaluation erhalten.[10] Allerdings werden sie um eine Regel bzw. eine rekursive strikte Let-Regel erweitert.

$$\frac{(\Gamma, x_1 \mapsto e_1) : x_1 \Downarrow \Theta : z_1 \quad \Theta : e \Downarrow \Delta : z}{\Gamma : let!\ x_1 = e_1\ in\ e \Downarrow \Delta : z}\ Str$$

Anders als bei der Let-Regel der Semantik für Lazy Evaluation hat diese Regel eine Bedingung, um die Wiederverwendung (Sharing) von Ausdrücken zu gewährleisten.
Bei dieser Let!-Regel müssen zwei Ausdrücke vorhanden sein, e_1 und e. Der Ausdruck e_1 wird als erstes ausgewertet, der Ausdruck e repräsentiert den Wert, der für die Lazy Evaluation notwendig ist. Bei der Auswertung von e_1 wird der Heap Γ mit der Bindung $x_1 \mapsto e_1$ erweitert, und um die Wiederverwendung (Sharing) von Ausdrücken zu erreichen, wird x_1 als der auszuwertende Term genommen.

In der sich ergebenden Umgebung Θ wird eine Bindung zu x_1 erhalten bleiben, allerdings wird diese auf das ausgewertete Ergebnis hinweisen. Diese Umgebung wird dann als die Umgebung für die Auswertung von e genommen.

[10] Siehe Kapitel 2.5.1 dieser Arbeit.

3.6.2 Denotationelle Semamtik

Da die operationelle Semantik erweitert wurde, muss nun auch die denotationelle Semantik um die strikten Let!-Ausdrücke erweitert werden.

$$\vdots$$
$$\vdots$$
$$[\![let!\, x_1 = e_1 \; in \; e]\!]_\rho \quad = \bot, \; wenn \; [\![x_1]\!]_{\{\{x1 \mapsto e1\}\}\rho} = \bot$$
$$= [\![e]\!]_{\{\{x1 \mapsto e1\}\}\rho}$$

Für die Bedeutung der Let!-Ausdrücke werden wir zwei Fälle betrachten:

- wenn die Bedeutung des wiederverwendeten Ausdrucks undefiniert ist, dann wird die Bedeutung des Let!-Ausdrucks auch undefiniert sein.

- andersfalls bleibt die Bedeutung des Let-Ausdrucks erhalten.

3.7 Eigenschaften der Semantik

Die Eigenschaften der lazy-strikten Semantik sind nicht anders als die von der Semantik für Lazy Evaluation, daher werden sie hier nicht noch einmal erläutert.[11]

[11] Siehe Kapitel 2.6 dieser Arbeit.

3.8 Formalisierung der Kenntnisse

Um das Ziel dieses Ansatzes zu erreichen, nämlich ein offizielles Modell für die Implementierung von Striktheit zu entwickeln, ist es notwendig, die verschiedenen Kenntnisse derselben zu normen.
Deshalb ist es nötig eine Verbindung bzw. eine Relation zu der Semantik für Lazy Evaluation festzulegen. Um das zu erreichen müssen die Ausdrücke, die in strikte Ausdrücke umgewandelt wurden, wieder lazily sein.

- ✓ Entfernung der Striktheitsanotationen in Ausdrücken

$(x)^{-!}$	$= x$
$(\lambda x.e)^{-!}$	$= \lambda x.\ (e)^{-!}$
$(e\ x)^{-!}$	$= (e)^{-!}\ (x)^{-!}$
$(\text{let } x_1 = e_1 \ldots x_n = e_n \text{ in } e)^{-!}$	$= \text{let } x_1 = e_1^{-!} \ldots x_n = e_n^{-!} \text{ in } e^{-!}$
$(\text{let! } x_1 = e_1 \text{ in } e)^{-!}$	$= \text{let } x_1 = e_1^{-!} \text{ in } e^{-!}$

Die Funktion -! ist für Ausdrücke definiert, wo $e^{-!}$ ein Ausdruck e ist, in dem jeder Let! durch den entsprechenden Let-Ausdruck ersetzt wird.

- ✓ Entfernung der Striktheitsanotationen in Umgebungen

$$(\Gamma,\ x \mapsto e)^{-!} = \left(\Gamma^{-!},\ x \mapsto e^{-!}\right)$$
$$\{\quad\}^{-!} = \{\quad\}$$

Die Funktion -! ist für Umgebungen definiert, wo $\Gamma^{-!}$ die Umgebung Γ ist, in der jeder Ausdruck e durch den entsprechenden Ausdruck $e^{-!}$ ersetzt wird.

✓ Formalisierung der Kenntnissen

Die Kenntnisse, die die Programmentwickler über Striktheit haben, können mit den folgenden Theoremen formalisiert werden, wo die denotationelle Bedeutung durch $[\![\ ]\!]^{lazy}$ und die operationelle Bedeutung durch $\Downarrow^{lazy}$ bezeichnet wird[12].

- Ausdrücke, die bei der Lazy Evaluation nicht definiert sind, also $\perp$, werden auch undefiniert bleiben, wenn sie zu strikten Ausdrücken umgewandelt werden.

$$[\![e^{-!}]\!]^{lazy}_{\{\{\Gamma^{-!}\}\}\rho_0} = \perp \quad \Rightarrow \quad [\![e]\!]_{\{\{\Gamma\}\}\rho_0} = \perp$$

- Wenn ein Ausdruck zu einem strikten Ausdruck umgewandelt wird, der bei der Lazy Evaluation nicht undefiniert ist, wird das Ergebnis unverändert bleiben oder undefiniert werden.

$$[\![e^{-!}]\!]^{lazy}_{\{\{\Gamma^{-!}\}\}\rho_0} \neq \perp \quad \Rightarrow \quad \left([\![e]\!]_{\{\{\Gamma\}\}\rho_0} = \perp \vee \exists z, \Delta, \Theta\left[\Gamma : e \Downarrow \Delta : z \wedge \Gamma^{-!} : e^{-!} \Downarrow^{lazy} \Theta : z^{-!}\right]\right)$$

- Ausdrücke, die mit der Anwendung von Striktheit nicht undefiniert sind, werden (nach der Entfernung von Striktheit) mit der Lazy Evaluation auch nicht undefiniert sein und dasselbe Ergebnis haben.

$$[\![e]\!]_{\{\{\Gamma\}\}\rho_0} \neq \perp \quad \Rightarrow \left([\![e^{-!}]\!]^{lazy}_{\{\{\Gamma^{-!}\}\}\rho_0} \neq \perp \wedge \exists z, \Delta, \Theta\left[\Gamma : e \Downarrow \Delta : z \wedge \Gamma^{-!} : e^{-!} \Downarrow^{lazy} \Theta : z^{-!}\right]\right)$$

[12] Vgl Mixed Lazy/Strict Graph Semantics.

3. 9 Beispiel

Das folgende Beispiel soll zeigen, wie mit einer lazy-strikten Semantik Beweise durchgeführt werden können. Mit der Durchführung von Beweisen kann gezeigt werden, dass es mit einer lazy-strikten Semantik möglich ist, Ausdrücke zu unterscheiden, die bei einer lazy Semantik nicht unterscheidbar scheinen.
Seien:

$$\Omega \equiv (\lambda x.xx)(\lambda x.xx) \text{ und } f \equiv \lambda x.(let!\ y = x\ in\ 42)$$

Mit der lazy-strikten Semantik wird zwischen den Ausdrücken λx.Ω und Ω unterschieden. Aus diesem Grund werden zwei Eigenschaften bewiesen:

1. $\exists \Delta, z.\{\ \}: f\ \Omega \Downarrow \Delta : z$

Die erste Eigenschaft besagt, dass es unmöglich ist, eine endliche Ableitung mit der operationellen Semantik zu bilden.

$$[\![f\ \Omega]\!]_\rho = [\![(\lambda x.let\ !\ y\ =\ x\ in\ \ 42\ )\ (\Omega\ )]\!]_\rho$$

1. Als erstes wird *f* durch den Ausdruck λx.let! y = x in 42 ersetzt, wobei sowohl *f* als auch Ω Abstraktionen sind.

$$= ([\![\lambda x.let!\ y = x\ in\ 42]\!]_\rho)\downarrow Fn\ ([\![\Omega]\!]_\rho) \quad (2)$$

2. Aus zwei Abstraktionen ergibt sich eine Applikation. Damit wird die Applikationsregel der denotationellen Semantik angewendet.

$$= (Fn\ \underbrace{(\lambda v.[\![let!\ y = x\ in\ 42]\!]_{\rho\ \ (x\mapsto v)})}_{(3)})\downarrow Fn\ ([\![\Omega]\!]_\rho) = (\underbrace{\lambda v.[\![let\ !\ y = x\ in\ 42]\!]_{\rho\ \ (x\mapsto v)}}_{(4)})\ [\![\Omega]\!]_\rho$$

3. Die Projektionsfunktion wird eliminiert und die Varregel wird zunächst angewendet.

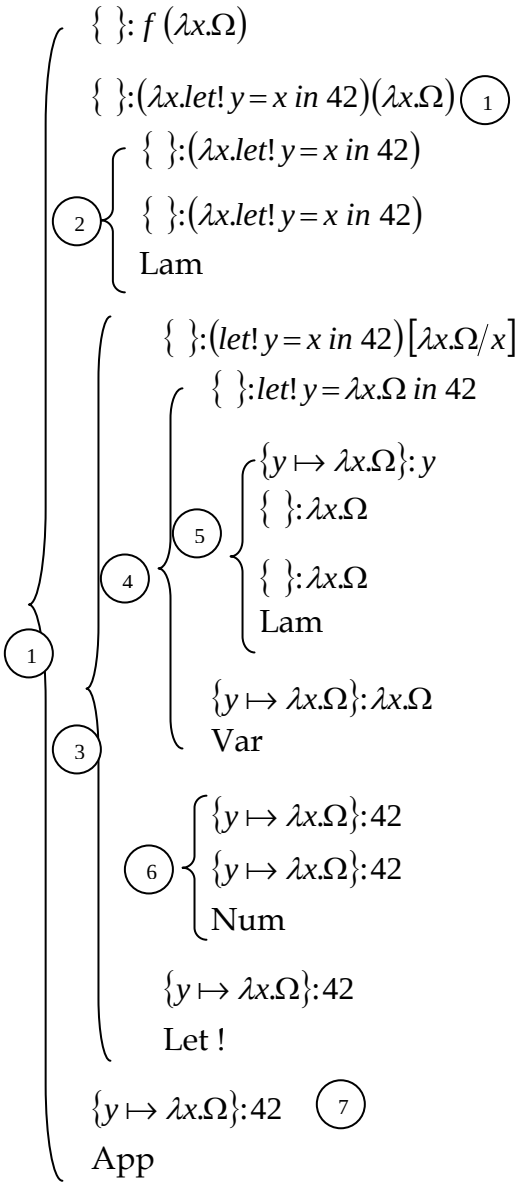
4. Der Ausdruck wird auf Ω angewendet und v durch $[\![\Omega]\!]$ ersetzt.

$$= [\![let\ !\ y = x\ in\ 42]\!]_{\rho\ \ (x\mapsto [\![\Omega]\!]\rho)} = \bot \quad (5)$$

5. Der Ausdruck ist nun ein strikter Let-Ausdruck, womit die Let!-Regel angewendet und das ⊥ Ergebnis erhalten wird.

2. $\exists \Delta.\{\}: f\ (\lambda x.\Omega) \Downarrow \Delta : 42$

Die zweite Eigenschaft wird mit den Reduktionsregeln bewiesen.



1. *f* wird durch den Ausdruck λx.let! y = x in 42 ersetzt. Die zwei Abstraktionen ergeben eine Applikation, so dass die Applikationregel angewendet werden muss.
2. Die Lambdaregel wird angewendet. Da es aber keinen weiteren Ausdruck gibt, der ausgewertet werden muss, wird der Ausdruck als solcher wieder geschrieben.
3. Der Let!-Ausdruck wird durch die strikte Letregel ausgewertet.
4. Die Varregel wird als nächstes angewendet. Hierbei wird die Funktion auf das Argument (λ.Ω) angewendet. Daraus erfolgt eine β-Reduktion, wo x durch λ.Ω ersetzt wird.
5. y wird ausgewertet. Der Heap wird mit der Bindung $\{y \mapsto \lambda.\Omega\}$ erweitert. Der Ausdruck λ.Ω muss als nächster ausgewertet werden, so wird die Lambdaregel angewendet.
6. Hierbei wird eine numerische Reduktionsregel angewendet, die besagt, dass jede Nummer zu sich selbst reduziert wird.
7. Das Ergebnis von *f* auf λ.Ω ist 42. Die Reduktion wird nicht weiter fortgesetzt, da kein weiterer Ausdruck reduziert werden muss.

4. Schlussbetrachtung

Mit dieser Seminararbeit konnte ich zeigen, dass der Ansatz von van Eekelen und de Mol tatsächlich eine sinnvolle Erweiterung bzw. Ergänzung der bisherigen im Bereich der funktionalen Programmierung entwickelten Ansätze darstellt, weil mit diesem Ansatz erstmals die Anforderung von Striktheit an funktionale Programmiersprachen berücksichtigt wurde.
Häufig traten und treten bei Programmen basierend auf einer funktionalen Programmiersprache, bei denen das Kriterium der Striktheit nicht berücksichtigt wurde, Fehler auf, wenn die Programme manuell geändert oder transformiert werden.
Zudem haben van Eekelen und de Mol die bis dahin allein auf persönlichen Erfahrungen und Erkenntnissen basierende Semantik konventionalisiert und normiert und haben damit die Semantik wissenschaftlich begründet.

Literatur:

Hudak, Paul (2000): The Haskell School of Expression. Cambrigde University Press.

Launchbury, John (1993): A Natural Semantics for Lazy Evaluation. Computing Science Department, Glasgow University.

Thompson, Simon (1996): Haskell: The Craft of Functional Programming. Addison-Wesley.

van Eekelen, Marko / **de Mol**, Maarten (2005): Mixed Lazy/Strict Graph Semantics. Departament of Software Technology, Nijmegen University.

Quellen im Internet:

Metzner, André: Aspekte funktionaler Sprachen: Einführung in den λ-Kalkül.

URL: http://uebb.cs.tu-berlin.de/sem_afs.vortraege/lambda.ps
[Stand: 02.01.2001]

URL:http://www..dcs.stand.ac.uk/research/architecture/more_functional
[Stand:21.09.2004]