Ein nichtdeterministischer call-by-need Lambda-Kalkül mit erratic Choice: Operationale Semantik, Programmtransformationen und Anwendungen

Dissertation
zur Erlangung des Doktorgrades
der Naturwissenschaften

vorgelegt beim Fachbereich 20
der Johann Wolfgang Goethe - Universität
in Frankfurt am Main

von
Arne Kutzner
aus Bielefeld

Frankfurt 1999
(DF1)

Danksagung

Die vorliegende Dissertation wurde von mir in den Jahren 1994 bis 1999 an der Professur für Künstliche Intelligenz und Softwaretechnologie der Johann Wolfgang Goethe-Universität Frankfurt am Main unter Herrn. Prof. Dr. Manfred Schmidt-Schauß angefertigt.

Herrn Prof. Schmidt-Schauß bin ich zu großem Dank verpflichtet, da ohne seine außerordentlich große Unterstützung und Mithilfe die vorliegende Arbeit nicht möglich gewesen wäre. Die vielen Stunden seiner Zeit, die er zum gemeinsamen Gespräch bereitgestellt hat, und die Fülle des Wissens, das er an mich weitergegeben hat, werde ich dankbar in Erinnerung behalten. Aber auch der von Achtung gekennzeichnete freundschaftliche Umgang ist zu erwähnen, der mir oft über Frustration und Ernüchterung im Verlauf meiner Arbeit hinweggeholfen hat.

Zu Dank verpflichtet bin ich auch meiner Lebensgefährtin Pok-Son Kim, die all meine Launen in dieser Zeit ertragen hat, und manch tröstendes Wort in schwerer Zeit sprach, oder aber in energischer Form das Weitermachen anmahnte, wenn mein Arbeitseifer wieder einmal zu wünschen übrig lies. Genauso bin ich meinen Eltern zu Dank verpflichtet. Auch sie haben mich häufig motiviert und standen unterstützend zur Seite, als dies erforderlich war.

Eine große Hilfe war das an der Professur über all die Jahre hinweg veranstaltete Diplomandenkolloqium, das einen gemeinsamen Ideenaustausch ermöglichte und einen Einblick in benachbarte Forschungsfelder gab. Zu erwähnen sind hier die Organisatoren Sven Eric Panitz und Marko Schütz, ohne deren Einsatz diese Veranstaltung nicht möglich gewesen wäre, aber auch all die Teilnehmer, die mit ihren Beiträgen manch gute Idee lieferten.

Dem Land Hessen und der Johann Wolfgang Goethe-Universität möchte ich für die Unterstützung der Promotion durch ein Stipendium in den Jahren 1994 bis 1996 danken, das mir eine große Hilfe war.

Frankfurt am Main im Oktober 1999

Inhaltsverzeichnis

Kapitel 1

Einleitung

Es ist unter Handwerken eine altbekannte Weisheit, daß das Ergebnis einer handwerklichen Arbeit in Korrelation zur Qualität des verwendeten Werkzeugs steht. Schlechtes Werkzeug führt schnell zu unbefriedigenden Ergebnissen, denn irgendwann kann ein auch noch so großes Maß an handwerklicher Erfahrung die mindere Qualität eines Werkzeugs nicht mehr kompensieren. Dieser Erfahrungswert läßt sich durchaus auf den Bereich der Softwareentwicklung übertragen, was ein Vorkommnis der jüngeren Zeit auf drastische Weise verdeutlicht hat.

Das spektakulärste und wohl auch teuerste Feuerwerk der letzten 5 Jahre war die Explosion einer Ariane 5 Trägerrakete 37 Sekunden nach dem Start am 4. Juni 1996. Der Fehler[1], der zu der kontrolliert herbeigeführten Sprengung der Trägerrakete führte, war der Verlust aller Steuerinformationen als Ergebnis des Versuches eine 64-Bit umfassende Fließkommazahl in einen 16-Bit Integer-Wert zu überführen. Die Verwendung einer Programmiersprache mit einem bezüglich Typkonversionen sensiblen Typsystem hätte geholfen, diesen sehr teuren und eigentlich primitiven Fehler zu verhindern.

Solche Vorkommnisse können gleichzeitig auch als das Resultat der heute immer noch üblichen Form der Softwareentwicklung angesehen werden, die auf die Kurzformel "hacken[2] und dann möglichst viel austesten" gebracht werden kann. Viele der heute eingesetzten Software-Werkzeuge lassen auch gar keine andere Arbeitsweise zu. Programmiersprachen wie C [KR77] oder C++ [ES91] verfügen über keinerlei mathematisch formale Grundlage, nichteinmal ihre genaue operationale Semantik ist festgelegt. Aber die genaue Kenntnis einer operationalen Semantik ist die Voraussetzung, um nichttriviale Programmanalysen oder aber auch Pro-

---

[1] Der Untersuchungsbericht des CNES (French National Center for Space Studies) und der ESA (European Space Agency) zu diesem Vorfall wird im Internet unter der Adresse `http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html` bereitgestellt.

[2] Ein umgangssprachlicher Begriff, der hier als Gleichnis für das Verb "programmieren" benutzt wird.

grammtransformationen vornehmen zu können.
Um teure Fehler wie den oben beschriebenen in Zukunft zu vermeiden, sollte anstelle des Austestens mehr und mehr die maschinelle Programmverifikation mittels automatischer Beweiser treten. Automatische Beweiser können vielerlei Analysen durchführen wie beispielsweise Terminierungsanalysen oder aber auch numerische Analysen. Wie bereits erwähnt, ist hierfür jedoch die genaue Kenntnis der operationalen Semantik einer Programmiersprache notwendig. An dieser Stelle ist der Bezug zu der vorliegenden Dissertation gegeben. Diese Arbeit soll einen Beitrag dazu leisten einer spezifischen Gattung funktionaler Programmiersprachen eine mathematisch-formale Grundlage in Form eines Kalküls zu verleihen.
Die nun folgenden Abschnitte der Einleitungen sollen das Umfeld, in dem sich die vorliegende Arbeit bewegt, genauer vorstellen. Dazu werden nach einigen einführenden Worten zum Thema funktionale Programmiersprachen einige Merkmale von diesen näher vorgestellt. In diesem Zusammenhang werden verschiedene bisher entwickelte I/O-Konzepte bei verzögernd auswertenden funktionalen Sprachen vorgestellt. Große Bedeutung besitzt in dieser Arbeit das Problem des Nichtdeterminismus bei funktionalen Sprachen, daher ist diesem Thema ein eigener Abschnitt gewidmet. Diese Ausführungen schaffen die Voraussetzungen, um in einem folgenden Abschnitt Motivation und Zielsetzung dieser Dissertation darzulegen. Mit einer Übersicht über verwandte Arbeiten sowie eine inhaltliche Übersicht wird die Einleitung abgeschlossen.

Einige Worte seien noch zu der Verwendung von Anglizismen in der vorliegenden Arbeit gesagt. Anstatt bei englischen Begriffen das deutsche Äquivalent zu benutzen, wird häufig das englische Original beibehalten. So wird beispielsweise in dieser Arbeit der Begriff "power domain" und nicht das deutsche Äquivalent "Potenzbereich" benutzt. Hintergrund dieser Entscheidung ist, daß in deutscher Sprache publizierte wissenschaftliche Arbeiten im Bereich der Informatik eher die Ausnahme sind. Um dem Leser das Studium dieser Dissertation nicht zusätzlich zu erschweren, werden daher in dieser Arbeit englische Fachbegriffe häufig beibehalten. Es sei explizit darauf hingewiesen, daß dies nicht als eine Mißachtung der deutschen Sprache verstanden werden soll.

1.1 funktionale Programmiersprachen

Eine der wohl bekanntesten und zugleich ältesten Programmiersprachen ist LISP [MAE^{+}62]. LISP wird häufig als Urvater allen funktionalen Programmierens angesehen, da es die erste Bekanntheit erlangende Programmiersprache war, die den Gedanken umsetzte, ein Programm ausschließlich mittels Funktionen zu modellieren und nicht auf imperativen Konstrukten wie beispielsweise bei COBOL [ANS85] oder FORTRAN [ISO97] aufzubauen. Pate bei der Entwicklung von LISP stand der klassische Lambda-Kalkül [Bar84], wenngleich einige Eigenschaften von LISP nicht mit diesem vereinbar sind, wie beispielsweise die Möglichkeit Zuweisungen vornehmen zu können sowie das Seiteneffekt-behaftete I/O von LISP. Motiviert durch LISP erfolgte in den 60'ziger Jahren eine intensive Untersuchung des Lambda-Kalküls als Grundlage funktionaler Programmiersprachen, zum Beispiel in [Mor68]. Als Ergebnis dieser Untersuchung entstanden über die Jahre eine Vielzahl verschiedener Gattungen funktionaler Sprachen, mit unterschiedlichen Charakteristika. In dieser Einleitung werden einige diese Charakteristika kurz betrachtet. Dies ist erforderlich, um in einem späteren Abschnitt die Hintergründe und Motivation für die Entwicklung des in dieser Arbeit vorgestellten Λ_{let}-Kalküls darlegen zu können.

Zum Verständnis der nun folgenden Abschnitte ist es erforderlich, über gewisse Grundkenntnisse bezüglich funktionalen Programmierens zu verfügen. Leser, die über keine solche Grundkenntnisse verfügen, seien folgende Einführungen aus dem Bereich der verzögernd auswertenden funktionalen Programmiersprachen empfohlen:
Eine kurze und kompakte Einführung in die Welt der verzögernd auswertenden funktionalen Programmiersprachen gibt [Hug89]. Eine weitaus umfassendere Einführung in Form eines ganzen Buches gibt [BW88] oder neuer und direkt an Haskell orientiert [Tho96]. Im Internet wird unter der Adresse `http:\\www.haskell.org` ein reichhaltiges Informationsangebot bereitgestellt, dem Verweise auf weitere einführende Literatur entnommen werden können.

1.1.1 call-by-value, call-by-name, call-by-need Auswertung

Das wohl wesentlichste Bewertungskriterium einer funktionalen Sprache ist ihre Strategie bei der Auswertung eines Programms (Ausdrucks). Hier kann zunächst eine Differenzierung nach call-by-value und call-by-name Auswertung vorgenommen werden. Um zu zeigen, was sich hinter den beiden Begriffen verbirgt, definieren wir drei Beispielfunktionen f, g, h wie folgt:

$$f\,x = x + x; \quad g\,x = x * x; \quad h = g(f\,1)$$

Offensichtlich ist h eine nullstellige Funktion, hinter der sich ein zu berechnender Wert verbirgt. Diesen Wert wollen wir nun berechnen. Dazu existieren zwei verschiedene Strategien:

- Zum einen können wir zunächst $f\,1$ auswerten und auf das erhaltene Ergebnis dann g anwenden, dies ist eine Methode, wie sie eine traditionelle schulmathematische Sicht nahelegen würde.
- Zum anderen können wir jedoch zuerst das Argument von g unausgewertet lassen und $g(f\,1)$ bei h durch $(f\,1) * (f\,1)$ ersetzen. Wir haben damit g ausgewertet ohne das Argument von g vorher auszuwerten. Danach berechnen

> wir zweimal den Wert von $f\,1$ und multiplizieren, um das Endergebnis zu erhalten.

Die erste Auswertungsstrategie wird call-by-value Auswertung genannt, die zweite Methodik call-by-name Auswertung. Bei call-by-value Auswertung werden zunächst die Argumente einer Funktion bis Normalform ausgewertet, bevor die Funktion auf die so ermittelten Werte angewendet wird. Normalform bedeutet hier, daß eine atomare Form wie beispielsweise eine Zahl vorliegt, also ein Objekt, das keine weiteren Berechnungen beinhaltet. Call-by-name Auswertung hingegen geht quasi genau invers vor, die Argumente werden zunächst nicht ausgewertet, stattdessen werden die Vorkommen eines Arguments mittels eines Substitutionsprozess durch das Argument ersetzt. Call-by-name ist offensichtlich sowohl bezüglich des Platzbedarfs wie auch bezüglich des Berechnungsaufwandes uneffizienter als call-by-value, da zweimal $f\,1$ berechnet werden muß. Es fragt sich also, ob diese Methodik einen anderen Vorteil besitzt, den obiges Beispiel nicht zeigt. Dazu betrachten wir ein weiteres Beispiel:

$$f\,x = f\,x; \quad g\,x = 1; \quad h = g(f\,1)$$

Die Definition $f\,x = f\,x$ erzeugt dabei eine Art schwarzes Loch (black hole), hinter ihr verbirgt sich eine niemals endende Berechnung, egal auf welches Argument f angewendet wird. Versuchen wir den Wert von h nun mittels der call-by-value Strategie zu berechnen, endet die Berechnung beim schwarzen Loch, d.h. wir werden nie ein Ergebnis erhalten. Bei der call-by-name Strategie hingegen wird bereits nach einem Schritt das Ergebnis 1 erhalten.
Der Vorteil von call-by-name Auswertung scheint offensichtlich ein signifikant besseres Terminierungsverhalten zu sein. Call-by-name Auswertung entspricht bezogen auf den klassischen Lambda-Kalkül der Reduzierung eines Ausdrucks durch wiederholte Wahl des am weitesten links stehenden Redex. Es ist seit langem bekannt ([Bar84] Lemma 13.2.2), daß diese Auswertungsstrategie normalisierend wirkt und somit eine Sonderstellung einnimmt.

Eine Fragestellung, die sich nun ergibt, lautete, ob und dann wie sich die Effizienzmängel der call-by-name Strategie überwinden lassen. Den Weg zu einer Lösung dieses Problems zeigt folgende Beobachtung:

> Die wiederholte Berechnung von $f\,1$ bei dem ersten Beispiel führt immer zum selben Ergebnis. Wird eine Modellierung von Seiteneffekten ausgeschlossen, läßt sich diese Aussage offensichtlich auf beliebig komplexe Ausdrücke ausdehnen.

Obige Beobachtung läßt die Schlußfolgerung zu, daß es eine legitime Methodik wäre, anstelle einer Substitution Sharing zu benutzen, um die wiederholte Auswertung zu verhindern und den Platzbedarf zu reduzieren. Was Sharing dabei bedeutet, soll an dem ersten Beispiel demonstriert werden:

> Anstelle der Durchführung einer Substitution wie bei obigem Beispiel bilden wir einen Ausdruck `let` $x = f\,1$ `in` $x * x$. Das Wortpaar `let` $\cdots$ `in` $\cdots$ soll dabei andeuten, daß alle Vorkommen der Variablen x bei $x * x$ an $f\,1$ gebunden werden.

Die Bezeichnung, die für diese Art der Auswertung gebildet wurde, lautet call-by-need Auswertung. Der Begriff call-by-need geht auf [Wad71] zurück[3]. Call-by-need Auswertung ist somit call-by-name Auswertung bei der Gegenwart von Sharing. Bezogen auf die Implementation von call-by-need Evaluatoren besteht ein sehr starker Bezug zu Graphen, da diese eine gute Möglichkeit bilden, das Sharing zu modellieren. Eine exzellente Darstellung dieser Zusammenhänge liefert [PJ87].

Wir haben hier die drei Begriffe call-by-value, call-by-name und call-by-need Auswertung vorgestellt. Zwei andere Begriffe, die im Zusammenhang mit Auswertungsstrategien in der Literatur vorkommen, sind strikte (eager) und verzögerte (lazy) Auswertung. Strikte Auswertung kann mit call-by-value Auswertung gleichgesetzt werden. Verzögerte Auswertung wird im allgemeinen als Äquivalent für call-by-need Auswertung [Lau93] benutzt. Irritierenderweise existiert auch ein "lazy lambda calculus" ([Abr90], [Ong92]), der jedoch auf call-by-name Auswertung aufbaut.
Sprechen wir im weiteren von einer "verzögernd auswertenden funktionalen Sprache", dann wird damit eine funktionale Sprache bezeichnet, die (zumindest im allgemeinen Verständnis[4]) call-by-need Auswertung benutzt.

1.1.2 pure und impure funktionale Sprachen

"Shall I be pure or impure ?", mit dieser Frage beginnt Wadler seine bekannte Arbeit [Wad95] über monadisches Programmieren. Pure funktionale Sprachen wie Miranda[5] [Tur90] oder Haskell verstehen sich in der Tradition des einfachen klassischen Lambda-Kalkül und erlauben nur solche Erweiterung, wie Konstruktoren u.ä., die mit diesem vereinbar sind. Im Kontext mit puren funktionalen Sprachen ist der Begriff der referentiellen Transparenz (referential transparency) von zentraler Bedeutung. Ein Ausdruck t ist referentiell transparent, wenn jeder Subausdruck von t und sein zugehöriger Wert (das Ergebnis der Auswertung des Subausdrucks) ausgetauscht werden können, ohne daß sich dadurch der (mittels einer festen Auswertungsstrategie berechnete) Wert von t ändert. Die Bewahrung

---

[3] siehe dazu [MOW98]

[4] Diese Anmerkung bezieht sich auf den auch in [Mor98] angemerkten Sachverhalt, daß beispielsweise die Spezifiktion der Programmiersprache Haskell [PHA$^+$97] keine Ausagen über die zu benutzende Auswertungsstrategie beinhaltet, im allgemeinen Verständnis jedoch davon ausgegangen wird, daß eine Haskell-Implementation call-by-need Auswertung benutzt.

[5] Miranda is a trademark of Research Software Ltd.

der referentiellen Transparenz ist Voraussetzung dafür, daß eine funktionale Sprache als pur angesehen wird. Impure Sprachen wie Standard ML [MTH90] oder Scheme [RCe86] hingegen beinhalten Elemente, die mit dem klassischen Lambda-Kalkül bzw. der referentiellen Transparenz nicht vereinbar sind, wie beispielsweise Zuweisungen oder exceptions.
Pure funktionale Sprachen sind besser zu analysieren, können ohne Probleme call-by-need Auswertung benutzen, impure Sprachen hingegen besitzen Effizienzvorteile und lassen in bestimmten Bereichen eine "natürlichere Programmierung" zu. Einen Bereich, auf den dies insbesondere zutrifft, sind I/O-Operationen. I/O-Operationen sind mit dem klassischen Lambda-Kalkül bzw. der referentiellen Transparenz nur schwer vereinbar, da sie von ihrer Natur her Seiteneffekt-behaftet sind. So gibt die nullstellige Funktion `getchar`, wie wir sie aus den meisten Programmiersprachen her kennen, bei jedem Aufruf einen anderen Wert zurück. Zur Überwindung dieser Problematik wurden bei den puren Sprachen verschiedene Konzepte entwickelt, die später in diesem Abschnitt vorgestellt werden. Allen diesen Konzepten ist gemein, daß sie mehr oder minder große Kritikpunkte aufweisen, wie eine konterintuitive Programmierung oder Beschränkungen bei der Programmierung, die aus Entscheidbarkeitsproblemen resultieren. Aus der Motivation heraus, diese Kritikpunkte zu überwinden und zugleich die Vorteile von verzögernder Auswertung zu erhalten, wurde die impure verzögernd auswertende Sprache Natural Expert [HNSSH97] entwickelt.

Die vorliegende Arbeit stellt einen nichtdeterministischen call-by-need Lambda-Kalkül vor. Motivation für die Entwicklung dieses Kalküls war, eine formale Grundlage für verzögernd auswertende funktionale Sprache mit Seiteneffekt-behafteten I/O in der Tradition von Natural Expert zu schaffen. Der Nichtdeterminismus dient dabei dazu, Seiteneffekt-behaftetes I/O auf Kalkülebene repräsentieren zu können. Um diesen I/O-Gedanken im Umfeld bereits bekannter I/O-Konzepte einzuordnen zu können, wird im nun folgenden Abschnitt ein Überblick über I/O-Konzepte bei verzögernd auswertenden funktionalen Sprachen gegeben.

1.1.3 I/O bei verzögernd auswertenden funktionalen Sprachen

Bevor verschiedene Konzepte für I/O bei verzögernd auswertenden funktionalen Sprachen vorgestellt werden, ist es zunächst erforderlich, verschiedene I/O-Arten zu trennen, da die verschiedenen Konzepte unterschiedlich leistungsfähig sind. Wir führen hier eine grobe Differenzierung nach folgenden drei I/O-Formen durch:

1. *Terminal-I/O.*
   Die klassische Form von I/O, bei der I/O-Anweisungen als ein sequentieller Strom gekapselter Eingaben und Ausgaben angesehen werden. Eingaben erfolgen über die Tastatur, und Ausgaben über den Bildschirm oder Drucker.

Eingaben haben die einfache Form wie eine ja/nein-Abfrage oder die Eingabe einer Zahl oder Zeichenkette. In [Gor93] wird diese I/O-Form auch *teletype-I/O* genannt.

2. *Window-I/O.*
   Dies ist die Form von I/O, wie es beispielsweise bei dem aus der UNIX-Welt bekannten X-Window System (`http://www.x.org`) vorkommt. Diese Form ist im allgemeinen Verständnis sehr objektorientiert [Mey88] und beinhaltet standardmäßig folgende Komponenten:

   - Fenster (Windows), die als Eingabe/Ausgabe-Schnittstelle für den Benutzer dienen.
   - Nebenläufigkeit von Prozessen, da auf den verschiedenen Fenster unterschiedliche Aktivitäten gleichzeitig möglich sein sollen.
   - Interprozeßkommunikation bzw. Signalkommunikation, damit die Objekte wie Fenster u.ä. untereinander Nachrichten austauschen können.

3. *Server-I/O*
   Gedanklicher Hintergrund sind Datenbanken, WWW-Server und andere Systeme, die über keine direkte Schnittstelle zur Benutzerkommunikation verfügen, sondern auf einen client zur Kommunikation angewiesen sind. Anforderungen bei dieser I/O-Form sind:

   - Starke Nebenläufigkeit von Prozessen, da mehrere client-Anfragen gleichzeitig abarbeitbar sein müssen.
   - Synchronisation von Dateizugriffen.

   Da Window-Systeme teilweise auch eine client-server-Architektur besitzen, kann Server-I/O durchaus als Vorstufe von Window-I/O angesehen werden.

Wir werden nun zunächst sehen, welche I/O-Konzepte bei den puren Sprachen Haskell und Clean entwickelt wurden. Danach wird das I/O-Konzept der impuren Sprache Natural Expert kurz vorgestellt. I/O bei verzögernd auswertenden funktionalen Sprachen wird als eine sehr interessante Thematik angesehen und war bis in die jüngere Vergangenheit immer wieder Gegenstand von Dissertationen [Gor92], [Ach96], [CH98].

synchronised-stream I/O, continuation-passing I/O

Dies sind zwei Formen von Terminal-I/O, die als stream-based bezeichnet werden, da sie einen input stream in einen output stream transformieren. Beide I/O-Modelle waren Teil der frühen Versionen von Haskell [HPW^{+}92], sind jedoch ab der Version 1.3 [HAB$^+$96] der Sprachspezifikation von Haskell zugunsten des monadischen-I/O aufgegeben worden. Eine semantische Betrachtung dieser I/O-Formen gibt [Gor93]. Bereits in der Frühphase der Entwicklung von Haskell wurde erkannt, daß synchronised-stream I/O und continuation-passing I/O wechselseitig aufeinander abbildbar sind ([Gor93]).

monadisches I/O

Monadisches Programmieren ist ein Überbegriff für eine bestimmte Art funktionaler Programmierung, die in [Wad95] vorgestellt wird. Der Gedanke hinter monadischem Programmieren ist das Verstecken bzw. unsichtbare Weiterschieben von Funktionsargumenten, indem Funktionen mittels spezieller monadischer Kombinatoren zu neuen Funktionen zusammengefügt werden. In der Tradition dieser Programmierform wurde monadisches I/O entwickelt, das in [PJW93] beschrieben und dem continuation-passing I/O gegenübergestellt wird. Monadisches I/O ist in seiner Grundform Terminal-I/O, jedoch existieren mehrere Weiterentwicklungen, die deutlich darüber hinausgehen und im Anschluß vorgestellt werden. In [AP95] wird monadisches I/O als Umgebungs-basiert (environment based) eingeordnet und damit familiär dem später vorgestellten I/O der Programmiersprache Clean zugeordnet.

concurrent haskell

Concurrent haskell wird ausführlich in [JGF96] vorgestellt. Es ist eine monadische Erweiterung von Haskell, deren Motivation die Bereitstellung einer Umgebung zur Modellierung von Nebenläufigkeit in Haskell ist. Es ist somit geeignet, Server-Applikationen in Haskell zu entwickeln und damit der dritten der obigen I/O-Formen zuzuordnen. Concurrent haskell ist die Basis der nun folgenden zwei I/O-Konzepte.

fudgets und haggis

Fudgets [CH98] und Haggis [FPJ95] sind zwei sehr leistungsfähige I/O-Konzepte für Haskell, mit denen sich fensterbasierende Applikationen realisieren lassen und die somit der zweiten der oben aufgeführten I/O-Formen zuzuordnen sind. Haggis ist ein Aufsatz auf die concurrent-Erweiterung des Glasgow Haskell Compilers (concurrent haskell). Haggis versteht sich dabei vollständig in der Tradition des monadischen Programmierens und bildet die gesamte I/O-Funktionalität monadisch ab. Bei den Fudgets hingegen steht im Zentrum der Begriff des “stream processor”. Ein stream processor bildet einen Strom von input messages auf einen Strom von output messages ab. Eine Benutzerschnittstelle wird dann durch die Kombination verschiedener “stream processors” mittels vorgefertigter Operatoren erhalten. In [MSC99] wird das Konzept der Fudgets formal betrachtet. Interessant im Kontext der vorliegenden Arbeit ist, daß die formale Betrachtung der Fudgets erratic-Nichtdeterminismus benötigt, eine Form von Nichtdeterminismus, die wir später kennenlernen werden. Von ihrem Leistungsumfang sind Fudgets und

Haggis durchaus vergleichbar, beide verfolgen den Anspruch, daß sich mit ihnen real-world-Applikationen realisieren lassen.

clean

Clean [PvE98] ist im Gegensatz zu den bisher vorgestellten I/O-Konzepten eine eigene Programmiersprache, die zwar stark an Haskell angelehnt ist, sich jedoch als eigenständig versteht. Clean wird als Programmiersprache für real-world-Applikationen präsentiert und in diesem Kontext auch als kommerzielles Produkt vermarktet. Eine Besonderheit von Clean gegenüber anderen verzögernd auswertenden funktionalen Sprachen ist seine starke Graphen-Bezogenheit, die sich darin äußert, das Term Graph Rewriting [BEvG$^+$87] als Basis von Clean angesehen wird. Der I/O-Gedanke von Clean wird in [AP95] vorgestellt. Eine Kurzcharakterisierung lautet:

Die Auswertung eines Programms entspricht der schrittweisen Reduzierung eines Graphen. Bestimmte Subgraphen, deren Wurzel eine spezielle Eindeutigkeitseigenschaft erfüllen, werden als eindeutig (unique) angesehen. Um zur Übersetzungszeit bestimmen zu können, welche Knoten eines Graphen eindeutig bleiben, wird ein “Uniqueness Type System” gebildet. Die Eindeutigkeitskenntnis wird dann (mitunter) im Kontext von I/O ausgenutzt, um bei einem Environment-passing I/O eine ungewünschte Vervielfachung der Umgebung (äußeren Welt) zu unterbinden.

Die Schwachstelle dieses I/O-Ansatzes ist, daß das Problem des Beweises der Eindeutigkeitseigenschaft eines Knotens unentscheidbar ist. Dies hat zur Folge, daß der Programmierer teilweise von seinem Wunschprogramm abweichen muß, da das Uniqueness Type System, obgleich gegebener Eindeutigkeit, nicht in der Lage ist, diese zu beweisen. Angesicht dieser Einschränkung stellt sich die Frage, ob das explizite “rumschieben einer Welt” (in Clean hat die getchar-Operation die Form `getchar World`) gegenüber monadischem I/O, wo die Welt (Umgebung) ein ständig verstecktes virtuelles Argument ist, tatsächlich vorteilhaft ist.

natural expert

Natural Expert (NE) [HNSSH97] ist eine kommerzielle Entwicklung der Software AG und übernimmt in dieser Aufstellung eine Sonderstellung ein, da es eine impure verzögernd auswertende funktionale Sprache ist, die über Seiteneffekt-behaftetes I/O verfügt. Die I/O-Funktionalität von NE ist auf Terminal-I/O beschränkt. Zwar ist NE derart designt, daß der Zugriff auf ein weites Feld von Datenbanken möglich ist, jedoch existiert in NE selbst keine Möglichkeit zur Modellierung von Nebenläufigkeit. Eine weitere interessante Besonderheit von NE ist, daß es eine Entwicklungsumgebung umfaßt, Funktionen werden über feste Masken eingegeben, anstatt wie bei Haskell auf Basis reiner ASCII-Dateien kodiert zu werden.

Der I/O-Gedanke von NE lautet, daß I/O-Operationen bzw. ihr Wert durchaus geshart werden dürfen. So ist

```
($N, $N) WHERE $N IS ASK_INTEGER <'Pick a number'>
```
[6]

ein zulässiger Ausdruck in NE. Da Sharing dadurch in eine zentrale Position rückt, ist Vorsicht bei Programmtransformationen angebracht. Bei puren Sprachen wie Haskell sind Programmtransformationen wie die "common subexpression elimination" zulässig, die den folgenden Ausdruck

```
(ASK_INTEGER <'Pick a number'>, ASK_INTEGER <'Pick a number'>)
```

in den zuerst aufgeführten transformieren würden. Der erste Ausdruck liefert immer ein Tupel mit zwei gleichen Zahlen, während der letztere Ausdruck ein Tupel mit zwei beliebigen Zahlen liefert. Aus der Sicht einer puren Sprache ist die Elimination gemeinsamer Ausdrücke eine effizienzsteigernde Optimierung, da eine wiederholte Auswertung desselben Ausdrucks vermieden wird. Wegen der Gültigkeit der referentiellen Transparenz ist eine solche Optimierung bei puren Sprachen auch immer zulässig.

Bei NE hingegen verursacht diese Optimierung eine signifikante semantische Verschiebung. Eine wesentliche Optimierung, die solche semantischen Verschiebungen auch betrifft, ist fully-lazy Lambda-Lifting, eine Optimierung, die im Abschnitt über deterministische Subausdrücke betrachtet wird. Es gilt jedoch, daß "common subexpression elimination" bzw. fully-lazy Lambda-Lifting nicht immer zu einer semantischen Verschiebung führen, vielmehr ist darauf zu achten, ob sie im Kontext einer I/O-Operation vorkommen oder nicht. Da die Entwickler von NE keine formale operationale Semantik zu NE entwickelt haben, mußten sie sich auf Erfahrungswerte verlassen, um bestimmen zu können, welche Programmtransformationen zulässig sind und welche nicht. Eine Motivation für die vorliegende Arbeit war, diese Unsicherheit zu beseitigen und einen Kalkül zu entwickeln, der als eine formale Grundlage für eine verzögern auswertende funktionale Sprache mit einem I/O in der Tradition von NE benutzt werden kann.

1.2 Nichtdeterminismus bei funktionalen Sprachen

Der Begriff Nichtdeterminismus bezeichnet in der Informatik ein fundamentales Konzept, das auf eine lange Tradition seiner Betrachtung zurückblicken kann. Bereits in den späten 50'ziger Jahren haben Rabin und Scott [RS59] das Konzept des nichtdeterministischen endlichen Automaten vorgestellt. Einige Zeit später folgte die Einführung von Turingmaschinen mit Orakeln durch Kreider und Ritchie [KR64]. Zusätzlich wurden funktionale Sprachen sowie sequentielle imperative Sprachen entwickelt, die um eine Möglichkeit zur expliziten Spezifikation von

---

[6] Das Beispiel ist aus [HNSSH97] entnommen, eine Erklärung der Notation wird dort gegeben.

Nichtdeterminismus erweitert waren. Große Bekanntheit erlangte in diesem Zusammenhang [McC61], wo McCarthy eine um den nichtdeterministischen Operator *amb* erweiterte funktionale Sprache vorstellt. Weitere historische Anmerkungen können in [SS92] gefunden werden.

Die obige Einführung deutet bereits an, daß die Vorstellung, was Nichtdeterminismus modelliert, bei den einzelnen Autoren variierte. So führten Kreider und Ritchie Nichtdeterminismus im Kontext von Komplexitätsbetrachtungen ein, wohingegen McCarthy die Modellierung von Nebenläufigkeit interessierte. Mit der Zeit entstanden so eine Reihe von Formen und Anwendungsfelder von Nichtdeterminismus, die gegeneinander abgegrenzt werden können. In diesem Abschnitt soll ein kurzer Überblick über die verschiedenen Arten von Nichtdeterminismus gegeben werden, die zur Zeit differenziert werden. Zugleich soll dieser Abschnitt dem Leser ermöglichen, den später vorgestellten nichtdeterministischen call-by-need-Kalkül in seinem Umfeld einzuordnen.
Ausgangspunkt für diesen Abschnitt war [SS92], wo eine funktionale Miniatursprache eingeführt wird und insgesamt 12 verschiedene nichtdeterministische Semantiken für diese entwickelt werden. Zur Qualifizierung von Nichtdeterminismus werden von Sondergaard und Sestoft in [SS92] 4 Merkmale vorgestellt:

- Angelic, demonic und erratic Nichtdeterminismus mit den Unterformen global angelic, local angelic, global demonic sowie local demonic.
- Schwacher (weak) und starker (strong) Nichtdeterminismus.
- Beschränkter (bounded) und unbeschränkter (unbounded) Nichtdeterminismus.
- Eingeschränkter (restrained) und uneingeschränkter (unrestrained) Nichtdeterminismus.

Im Kontext der vorliegenden Arbeit sind die letzten drei Merkmale von nur geringer Bedeutung, für eine Erklärung der Begriffe sei daher auf [SS92] verwiesen. Wichtig hingegen ist, gerade im Bezug auf verwandte Arbeiten, die Differenzierung nach angelic, demonic und erratic Nichtdeterminismus. Es handelt sich hier um eine Qualifizierung von Nichtdeterminismus, bezüglich der Fragestellung von nichtterminierenden Berechnungen. Seien s, t zwei beliebige Ausdrücke, $\perp$ das Symbol, welches die nichtterminierende Berechnung repräsentiert und $\otimes$ ein Operator, der zwei Ausdrücke nichtdeterministisch miteinander verknüpft. Die nachfolgende Tabelle gibt die Semantik des Ausdrucks $s \otimes t$ bei den drei Arten von Nichtdeterminismus wieder:

	erratic	angelic	demonic
$s \otimes t$, wobei $s, t \neq \bot$	$s \vee t$	$s \vee t$	$s \vee t$
$s \otimes \bot$, wobei $s \neq \bot$	$s \vee \bot$	s	$\bot$
$\bot \otimes t$, wobei $t \neq \bot$	$\bot \vee t$	t	$\bot$
$\bot \otimes \bot$	$\bot$	$\bot$	$\bot$

Offensichtlich unterscheiden sich die drei Formen, wenn einer der beiden durch $\otimes$ verknüpften Ausdrücke eine terminierende, der andere hingegen eine nichtterminierende Berechnung besitzt. Diese Unterscheidung spiegelt ein spezifisches semantisches Verständnis wieder, was jeweils mit jeder der drei Formen verbunden ist. Erratic Nichtdeterminismus kommt die intuitive Vorstellung am nächsten, daß der nichtdeterministische Berechnungsschritt operational dem Wurf einer Münze gleichkommt. Abhängig von dem Ergebnis des Münzwurfs wird einer der zwei alternativen Berechnungswege weiterverfolgt. Bei angelic Nichtdeterminismus hingegen wird immer ein terminierender Berechungsweg ausgewählt, sofern dieser existiert. Gedanklich steht hinter dieser Form von Nichtdeterminismus die parallele Simulation der Auswertung beider Argumente von $\otimes$, wobei das Ergebnis einer der Berechnungen zurückgegeben wird, die terminiert. Demonic Nichtdeterminismus wiederum ist das Inversum von angelic Nichtdeterminismus. Hier wird zufällig einer von zwei Ausdrücken ausgewählt, wenn beide Berechnungswege terminieren, ansonsten wird die gesamte Berechnung als nichtterminierend angenommen. Die Begriffe angelic und demonic gehen auf C. A. R. Hoare zurück. Bei beiden Formen ist eine weitere Differenzierung in eine lokale und eine globale Form erforderlich, da die Konzepte unterschiedlich werden, sofern die Betrachtung des Operators $\otimes$ auf umgebendende Kontexte ausgedehnt wird. Dies verdeutlicht der Ausdruck

$$\texttt{if}\ (0 \otimes 1)\ \texttt{then}\ \bot\ \texttt{else}\ 1$$

Wird bei obigem Ausdruck die Semantik des Operators $\otimes$ als lokal angesehen, lautet die Menge möglicher Ergebnisse $\bot$ oder 1, egal ob angelic oder demonic Nichtdeterminismus als Semantik gewählt wird. Wird die semantische Sicht jedoch auf den umgebenden Kontext ausgedehnt, d.h. globalisiert, wird 1 bei angelic Nichtdeterminismus und $\bot$ bei demonic Nichtdeterminismus als Resultat erhalten. Bei lokal angelic Nichtdeterminismus wird der Operator $\otimes$ üblicherweise mit *amb* (eine Abkürzung von ambiguos) bezeichnet, dies geht auf [McC61] zurück, wo diese Form von Nichtdeterminismus zuerst untersucht wurde.

Nichtdeterministische Simulationstechniken, wie sie in der Komplexitätstheorie beispielsweise bei nichtdeterministischen Turingmaschinen verwendet werden, entsprechen somit gemäß dieser Einteilung global angelic Nichtdeterminismus. Nebenläufigkeit von Prozessen wiederum kann semantisch mittels *amb* modelliert werden. I/O-Operationen wie ja/nein-Abfragen können mittels erratic Nichtdeterminismus semantisch erfaßt werden.

Erratic Nichtdeterminismus nimmt gegenüber angelic und demonic Nichtdeterminismus eine Sonderstellung ein, da der Agent, so sei die Einheit bezeichnet, die nichtdeterministische Entscheidungen im Verlauf der Abarbeitung eines Programms trifft, dort keinerlei Eigenintelligenz besitzt, d.h. der Agent trifft seine Entscheidungen rein zufällig. Bei angelic wie demonic Nichtdeterminismus hingegen besitzt der Agent die Fähigkeit, eine kontrollierte Entscheidung zu treffen, d.h. er ist intelligent. Bei der Verwendung von Nichtdeterminismus im Zusammenhang mit Komplexitätsbetrachtungen ist die Intelligenz des Agenten sogar eine zentrale Eigenschaft.

In dieser Arbeit werden wir einen call-by-need Lambda-Kalkül mit einem Choice-Operator definieren, dessen Semantik erratic Nichtdeterminismus ist. Der nun folgende Abschnitt soll zeigen, daß erratic-Nichtdeterminismus bei verschiedenen Auswertungsstrategien unterschiedliche Eigenschaften aufweist und damit eine weitere Hilfe geben, die vorliegende Arbeit einzuordnen.

1.2.1 call-by-need, call-by-name, call-by-value und erratic Nichtdeterminismus

Wir haben bis jetzt die Begriffe call-by-need, call-by-name, call-by-value sowie erratic-Nichtdeterminismus kennengelernt. Nun soll die Beziehung zwischen diesen Begriffen genauer dargelegt werden.
Bereits in den 70'ziger Jahren hat Plotkin in [Plo75] die Beziehung zwischen call-by-name und call-by-value auf Basis einer simplifizierten Form der Programmiersprache ISWIM [Lan64] untersucht und gezeigt, daß sich die beiden Auswertungsstrategien wechselseitig simulieren lassen. Wie auch immer, es ist zu beachten, daß die beiden Auswertungsstrategien über ein signifikant differentes Terminierungsverhalten verfügen, was das Beispiel $(\lambda x.\lambda y.y)\bot$ zeigt. Ariola und andere haben in [AFM$^+$95] einen deterministischen call-by-need Lambda-Kalkül zusammen mit einer Standard-Reduktion vorgestellt und gezeigt, daß der daraus konstituierbare call-by-need Evaluator mit dem call-by-name Evaluator des klassischen Lambda-Kalküls austauschbar ist. Bei einer deterministischen Sicht sind somit die drei Auswertungsstrategien, abgesehen von unterschiedlichem Terminierungsverhalten, durchaus als gleichwertig ansehbar. Bei der Einführung von erratic Nichtdeterminismus beginnt sich dieses Bild jedoch zu wandeln. Dazu betrachten wir den folgenden Beispielausdruck:

$$\texttt{let}\ x = \texttt{choice}\ 0\ 1\ \texttt{in}\ x + x$$

Die Funktion `choice` wähle dabei zufällig eines ihrer beiden Argumente aus, sie entspricht somit dem zuvor betrachteten Operator $\otimes$ bei erratic Nichtdeterminismus.
Wir differenzieren zunächst zwischen call-by-name Auswertung auf der einen Seite, sowie call-by-value zusammen mit call-by-need Auswertung auf der anderen

Seite. Bei einer call-by-name Auswertungsstrategie würde zunächst der Subausdruck `choice` 0 1 an beide Positionen der Variable x kopiert werden, und anschließend zweimal zwischen 0 und 1 gewählt werden. Als Menge möglicher Resultate wird somit $\{0, 1, 2\}$ erhalten. Bei einer call-by-value wie call-by-need Auswertungsstrategie hingegen würde zunächst die nichtdeterministische Entscheidung `choice` 0 1 getroffen werden, und dann entweder 0 oder 1 verdoppelt werden. Die Menge möglicher Resultatwerte entspricht in diesem Fall somit $\{0, 2\}$ und ist zu der bei call-by-name erhaltenen signifikant verschieden. Diese Sachverhalte sind schon länger bekannt, auf den Unterschied zwischen call-by-need und call-by-name wurde bereits 1979 in [AC79] aufmerksam gemacht.

Wir werden nun in einem zweiten Schritt sehen, daß über das Terminierungsverhalten hinaus eine weitere Differenzierungsebene zwischen call-by-value und call-by-need Auswertung bei erratic Nichtdeterminismus existiert, obwohl das obige Beispiel andeutet, daß beide Evaluationsstrategien abgesehen von Terminierungsaspekten zu derselben Resultatmenge führen. Dazu ist es erforderlich den Begriff der denotationalen Semantik kurz vorzustellen.
Mittels einer denotationalen Semantik werden syntaktischen Elementen, die wir hier als Ausdrücke bezeichnen, Objekte einer abstrakten mathematischen Struktur, üblicherweise ein Domain, zugeordnet. Eine denotationale Semantik ist korrekt im Bezug auf einen gegebenen Evaluator, wenn alle Umformungen eines Evaluators auf Ausdrucksebene bezüglich der denotationalen Semantik die Äquivalenz bewahren. D.h. können zwei Ausdrücke durch einen Evaluator ineinander überführt werden, müssen sie auf dasselbe mathematische Objekt abgebildet werden. Zu einem Evaluator, können durchaus mehrere korrekte denotationale Semantiken mit unterschiedlichen Eigenschaften existieren. Als Beispiel sei hier auf die Vielzahl von denotationalen Semantiken (Modellen) für den klassischen Lambda-Kalkül hingewiesen. Der Begriff Evaluator kann dabei für die Reduktionsregeln eines Kalküls wie des klassischen Lambda-Kalküls stehen, oder aber auch für eine spezifische Auswertungsstrategie wie call-by-name oder call-by-value. Hier existiert ein weiter Rahmen verschiedener Variationsmöglichkeiten. Eine Einführung in diese komplexe Thematik gibt [Sto77]. Die dort vorgestellten Grundlagen seien im Rest dieses Abschnitts als bekannt vorausgesetzt.

Die Entwicklung von denotationalen Semantiken für nichtdeterministische Kalküle erfordert die Verwendung von power-domains, deren Bildung, vergleichbar der Potenzmengenbildung auf klassischen Mengen, auf Basis eines bereits existenten domains geschieht. Die Bildung von power-domains soll hier nicht weiter betrachtet werden, vielmehr soll auf eine wichtige Differenzierung von denotationalen Semantiken aufmerksam gemacht werden, die in diesem Bereich erfolgt. Denotationale Semantiken auf Basis von power-domains werden in singuläre und plurale Semantiken unterteilt. Manchmal ([Cli82], [SS92]) wird die Differenzierung zwischen pluralen und singulären Semantiken auch an den Begriffen call-time-choice und run-time-choice [HA77] verankert. Hier soll die Differenzierung

jedoch an der Art der Interpretation von Variablen verankert werden, dies ist jedoch bedeutungsidentisch zu der call-time-choice und run-time-choice Differenzierung. Wir bezeichnen eine Semantik als singulär, wenn während der Bildung der Interpretation eines Ausdrucks Variablen in der Umgebung immer eine einelementige Menge zugeordnet wird, ansonsten bezeichnen wir sie als plural. Es ist eine sehr subjektive Bewertung, plurale Semantiken als natürlicher und vorteilhaft gegenüber singulären Semantiken anzusehen, da erstere weniger restriktiv sind als letztere. Clinger [Cli82] beispielsweise ist genau umgekehrter Meinung.
Wir kehren nun zurück zu der oben angekündigten Differenzierung zwischen call-by-value und call-by-need.

Bei nichtdeterministischer call-by-value Auswertung ist eine zugehörige denotationale Semantik im allgemeinen singulär, da die Argumente bis zu Normalform ausgewertet werden, bevor die Anwendung der Funktion erfolgt. Bei call-by-need wie call-by-name Auswertung hingegen ist eine zugehörige denotationale Semantik im allgemeinen plural, da Variablen an nicht zu Normalform ausgewertete Ausdrücke gebunden werden können. Das folgende Beispiel soll dies verdeutlichen:

$$\texttt{let}\ y = \texttt{choice}\ 0\,1\ \texttt{in}\ (\texttt{let}\ x = \texttt{choice}\ y\,2\ \texttt{in}\ x + x)$$

Bei einer call-by-value Auswertung wird zunächst `choice` $0\,1$ ausgewertet. Bei der denotationalen Interpretation hat dies zur Folge, daß bei der Interpretation des Subausdrucks `let` $x = \cdots$ `in` $x + x$ die Variable y einmal mit 0 und einmal mit 1 belegt wird und die beiden Ergebnisse der weiteren Interpretation, die für beide Belegungen erhalten werden, vereinigt werden. Bei einer call-by-need Auswertung hingegen wird zunächst `choice` $y\,2$ ausgewertet. Denotational hat dies zur Folge, daß bei der Interpretation von `choice` $y\,2$ die Variable y an eine Menge, nämlich $\{0, 1\}$, gebunden wird. Die Semantik ist somit plural.
Diese kurzen Ausführungen geben zugleich einen Einblick in das filigrane Zusammenspiel zwischen operationaler und denotationaler Semantik, das es teilweise sehr schwer macht, eine passende denotationale Semantik zu einer operationalen Semantik zu finden. Im Zusammenhang mit dem in dieser Arbeit vorgestellten Λ_{let}-Kalkül ist dieses Vorhaben beispielsweise trotz großen Zeitaufwands gescheitert, da kein geeigneter Domain gefunden werden konnte.

Dieser Abschnitt soll mit einer abschließenden Gegenüberstellung beendet werden. Offensichtlich steht call-by-need Auswertung bei der Einführung von Nichtdeterminismus call-by-value Auswertung näher als call-by-name Auswertung, was das erste der obigen Beispiele zeigt. Dies ist um so erstaunlicher, da call-by-need bei rein deterministischer Sicht nur als eine mittels Sharing optimierte Form von call-by-name angesehen wird. call-by-need scheint bei der Gegenwart von Nichtdeterminismus dieselben Resultatmengen wie call-by-value zu liefern, jedoch verfügt es weiterhin über sein günstigeres Terminierungsverhalten. Zudem ist eine denotationale Semantik bei call-by-need Evaluation im allgemeinen plural, was subjektiv als vorteilhaft bewertet wurde.

1.3 Motivation und Kurzcharakterisierung

Indem in den letzten Abschnitten verschiedene Aspekte funktionaler Programmsprachen betrachtet wurden, ist die Basis geschaffen worden, um die Motivation dieser Dissertation darzulegen und eine Kurzcharakterisierung zu geben.

Die vorliegende Arbeit soll einen Beitrag dazu leisten, eine mathematisch formale Grundlage für verzögernd auswertende funktionale Programmiersprachen bereitszustellen, die über ein Seiteneffekt-behaftetes I/O in der Tradition von Natural Expert [HNSSH97] verfügen. Eine solche mathematisch formale Grundlage ist eine Grundvoraussetzung, um die Korrektheit wesentlicher Programmtransformationen wie Lambda-Lifting oder ähnliches zeigen zu können. Solch eine formale Grundlage kann ein Kalkül bilden, es sind jedoch auch Alternativen wie eine abstrakte Maschine denkbar. Wir werden hier einen call-by-need Lambda-Kalkül vorstellen, der um einen nichtdeterministischen choice-Operator erweitert ist, dessen Semantik erratic Nichtdeterminismus ist. Die Wahl fiel dabei auf erratic Nichtdeterminismus, da diese Form von Nichtdeterminismus zur Modellierung von Seiteneffekt-behafteten I/O gut geeignet scheint. Der vorgestellte Kalkül verfügt weder über rekursive Let-Strukturen noch über Konstruktoren, und ist wegen dieser Einschränkungen nur beschränkt praxistauglich. Eine wesentliche Motivation für diese Arbeit bildete [SS96], wo bereits Lösungsansätze dieser Problemstellung auf Basis eines Superkombinator-Kalküls vorgestellt werden.

1.4 verwandte Arbeiten

Der in dieser Arbeit vorgestellte Λ_{let}-Kalkül entspricht dem in [AFM$^+$95] vorgestellten call-by-need-Kalkül erweitert um eine nichtdeterministische `choice`-Regel, deren operationale Semantik erratic Nichtdeterminismus ist. Aus [AFM$^+$95] gingen [AF97] und [MOW98] hervor, die beide eine vertiefende Betrachtung von [AFM$^+$95] darstellen. [AF97] und [MOW98] unterscheiden sich mitunter dadurch, daß in [AF97] das in [MOW98] sowie [AFM$^+$95] vorkommende `let`-Konstrukt fortgelassen wird und alle Betrachtungen auf puren Lambda-Ausdrücken erfolgen. Dieser Übergang zu puren Lambda-Ausdrücken wäre auch beim Λ_{let}-Kalkül möglich gewesen, jedoch ist der Preis dieses Übergangs ein Verlust an Übersichtlichkeit, da Sharing dann nicht mehr auf Ausdrucksebene repräsentierbar ist.

Eine häufiger kritisierte Einschränkung ([MS99], [Mor98], [MSC99]) all dieser Arbeiten ist, daß die darin vorgestellten Kalküle keine rekursiven Let-Strukturen zulassen. Diese Kritik zielt auf wohlbekannte Effizienzmängel ([PJ87], Abschnitt

12.5), die die Benutzung eines Y-Kombinators zur Darstellung von Rekursion bei Kalkülen ohne rekursives `let` zur Folge hat. Ferner wird die Abwesenheit von Konstruktoren und einem `case`-Konstrukt kritisiert, die Bestandteil jeder modernen funktionalen Programmiersprache sind. Zur Überwindung dieser Kritikpunkte wird in [MS99] und [MSC99] auf die in [Ses97] vorgestellte abstrakte Maschine "mark 1" zurückgegriffen. Jedoch verfügt Sestofts "mark 1" wiederum über eine andere, bereits aus [Lau93] bekannte Einschränkung auf Ausdrucksebene. Applikationen müssen dort stets die Form $t\,x$ mit x eine Variable haben; Ausdrücke, die diese Eigenschaft nicht erfüllen, müssen zuvor entsprechend transformiert werden. Diese Einschränkung wird in [MS99] wie auch [MSC99] zwar als unproblematisch angesehen, jedoch spätestens der Abschnitt über die *ucp*-Reduktion dieser Arbeit wirft einen Schatten auf diese Aussage.
Moran, Sands und Carlson erweitern in [MSC99] Sestofts abstrakte Maschine "mark 1", um einen nichtdeterministischen Operator, dessen operationale Semantik erratic Nichtdeterminismus ist. Hintergrund dieser Arbeit ist eine semantische Betrachtung der bereits erwähnten Fudgets bzw. des Konzepts der Kombination von "stream-processors", auf dem diese aufbauen. Kapitel 6 zeigt, daß eine reduzierte Form der abstrakten Maschine "mark 1" mit dem Λ_{let}-Kalkül dieser Arbeit austauschbar ist. Diese Beobachtung erlaubt es, einen Teil der in [MSC99] gezeigten Ergebnisse als eine Bestätigung von Ergebnissen dieser Arbeit zu interpretieren. Dies trifft insbesondere auf die in Kapitel 3 vorgestellten Ergebnisse bezüglich der Erhaltung der kontextuellen Äquivalenz durch eine Reihe von Reduktionen zu. Einige dieser Reduktionen, wie beispielsweise die *lbeta*- und *ldel*-Reduktion, erscheinen auch als Regeln der "equational theory" aus [MSC99] und werden auf diesem Wege bestätigt.

Wir haben zu Beginn dieses Abschnitts drei Arbeiten kennengelernt, deren Zugang zu call-by-need Auswertung wie bei beim Λ_{let}-Kalkül der vorliegenden Arbeit auf Basis eines Kalküls geschieht. Auf Kalkülebene existiert eine Differenzierung nach Kalkülen mit einer "small step semantic" und Kalkülen mit einer "natural semantic" bzw. "big step semantic". Kalküle mit einer small step semantic basieren auf der Vorstellung einer Regel-basierten Reduktion, die Ausdrücke schrittweise umformt, wobei dieser Prozess irgendwann terminiert oder auch nicht. Kalküle mit einer "natural semantic" hingegen basieren auf einem Beweissystem, das zum Beweis von Konvergenz oder Divergenzeigenschaften von Ausdrücken benutzt wird. Ein Nachteil von Kalkülen auf Basis einer natural semantic ist, daß sie keine Aussage über die Länge einer Reduktion erlauben. Ein Vorteil hingegen ist, daß die Formulierung denotationaler Aussagen erleichtert wird. In [HM95] werden Kalküle beider Formen im Zusammenhang mit Nichtdeterminismus auf der Basis von *amb* vorgestellt.
Sowohl der Λ_{let}-Kalkül dieser Arbeit, wie auch die Kalküle aus [AFM^{+}95], [AF97], [MOW98] verfügen über eine small step Semantik. Dieser Zugang zu call-by-need wird in [MS99] daher als "rewriting based" bezeichnet, da das Sharing auf syn-

taktischer Ebene behandelt wird und die Kalkülregeln gewährleisten, daß korrekt im Sinne von Sharing gehandelt wird.
Einen Zugang zu call-by-need Auswertung auf Basis einer natural Semantik gibt [Lau93]. Launchbary stellt dort eine operationale wie eine denotationale Semantik für einen erweiterten λ-Kalkül vor, der call-by-need modelliert. Die operationale Semantik ist dabei eine natural Semantik, d.h. wir wissen bei diesem Kalkül nur, daß ein Ausdruck in eine bestimmte Form überführbar ist, nicht jedoch wie dies geschieht. Anstelle des Begriffs call-by-need benutzt Launchbary den Begriff "laziness", was darin begründet ist, daß er seine Arbeit in der Tradition von [Abr90] betrachtet. Launchbary greift in [Lau93] auf einen "Trick" zurück, um Sharing im Kontext der β-Reduktion modellieren zu können. Er läßt nur eine auf die Substitution von Variablen durch Variablen beschränkte Form der β-Reduktion zu und benutzt call-by-name Auswertung. Auf diesem Wege werden niemals komplexe Ausdrücke wie Abstraktionen oder Applikationen kopiert, was das Sharing stören würde. Schwachstelle dieses Ansatzes ist, daß wie bei der abstrakten Maschine aus [Ses97] zunächst die Transformation eines gegebenen Ausdrucks in eine spezielle Form erforderlich wird. Von dieser Transformation ist jedoch ungewiß, welche semantischen Seiteneffekte sie im Bezug auf die Gesamtmenge aller Ausdrücke beinhaltet.

In [SI96] wird auf syntaktischer Basis der funktionalen Miniatursprache PCF [Plo77] eine operationale Semantik definiert, die call-by-need Auswertung modelliert. Der resultierende Kalkül wird entsprechend mit LAZY-PCF+SHAR bezeichnet. Die Betrachtungen beschränken sich jedoch auf die Untersuchung der Verträglichkeit zwischen dem Typsystem von LAZY-PCF+SHAR und der operationalen Semantik sowie auf die Korrektheit des Kalküls im Bezug auf call-by-name Auswertung, die als eine unoptimierte Form von verzögernder Auswertung angesehen wird. Untersuchungen in Richtung kontextuelle Äquivalenz oder einer denotationalen Semantik erfolgen jedoch nicht.

Die Frage nach Effizienzaspekten von Programmtransformationen bei call-by-need Auswertung ist Gegenstand von [MS99], einer Arbeit, die bereits früher in diesem Abschnitt referenziert, jedoch nicht näher betrachtet wurde. Als Bewertungsmaß für einen Ausdruck t wird in [MS99] die Anzahl der Berechnungsschritte der abstrakten Maschine mark-1 aus [Ses97] bis zum Erreichen einer Endkonfiguration ausgehend von einer Startkonfiguration basierend auf t vereinbart. Zwei Ausdrücke werden als gleich angesehen, wenn sie Beobachtungs-äquivalent (observationally equivalent) sind, die Beobachtungsäquivalenz entspricht semantisch der kontextuellen Äquivalenz dieser Arbeit, d.h. zwei Ausdrücke werden als gleich angesehen, wenn ihr Terminierungsverhalten in allen umgebenden Kontexten identisch ist. Als strengere Form der Beobachtungsäquivalenz wird zusätzlich eine Kostenäquivalenz definiert, die das oben erwähnte Bewertungsmaß mitbeinhaltet. Als technische Hilfsmittel werden ein Kontext-Lemma, vergleichbar dem in dieser Arbeit enthaltenen, bewiesen, sowie eine "Tick Algebra" definiert. Auf dieser Basis wird ein breiter Rahmen an Programmtransformationen untersucht und ein Improvement-Theorem bewiesen.

Alle bisher vorgestellten Publikationen betrachteten call-by-need Auswertung, d.h. Sharing war ein Bestandteil der darin vorgestellten Kalküle oder abstrakten Maschinen. call-by-need Auswertung wird jedoch, wie bereits früher erwähnt, bei Abwesenheit von Nichtdeterminismus als eine optimierte Form von call-by-name Auswertung angesehen. In diesem Kontext ist der "Lazy Lambda Calculus" aus [Abr90] zu erwähnen, der zwar das Wort "Lazy" in seinem Namen trägt, jedoch call-by-name Auswertung benutzt. Abramsky's Publikation ist ein häufig zitiertes Grundlagenpapier, dessen Methodiken und Begriffsvereinbarungen von anderen Autoren (z.B. Ong, [Ong92]) übernommen bzw. weiterentwickelt wurden. Die Benutzung des Begriffs WHNF in der vorliegenden Arbeit geht beispielsweise auch auf Abramsky's Publikation zurück. Es existieren einige Publikationen (z.B. [Ong93]), die den Lazy Lambda Calculus um nichtdeterministische Konstrukte erweitern, jedoch sind, wie wir bereits gesehen haben, Nichtdeterminismus in der Gegenwart von call-by-need Auswertung und Nichtdeterminismus in der Gegenwart von call-by-name Auswertung signifikant verschiedene Konzepte. Diese Arbeiten stehen daher in keinem engeren Bezug zu der vorliegenden Arbeit.

Wir haben in einem früheren Abschnitt dieser Einleitung gesehen, daß call-by-need und call-by-name Auswertung bei einem Wegfall der referentiellen Transparenz - hier durch die Einführung von Nichtdeterminismus - bei demselben Ausdruck unterschiedliche Ergebnisse liefern können. Ein interessanter Fragenkomplex ist nun, welche Folgen der Wegfall der referentiellen Transparenz bei call-by-name und call-by-value Auswertung hat.
Es sollen nun kurz einige Arbeiten vorgestellt werden, die der Untersuchung dieses Fragenkomplexes zugeordnet werden können. In [FF89] sowie [FH92] untersuchen Felleisen und Friedman bzw. Hieb, basierend auf der bereits früher in diesem Kapitel erwähnten Arbeit [Plo75] von Plotkin, call-by-value Auswertung bei der Gegenwart von imperativen Konstrukten wie Zuweisungen und Sprüngen. Den "praktischen Startpunkt" ihrer Arbeiten bildet die Untersuchung von impuren, strikten Sprachen wie Scheme und ML. Die Hinzunahme von imperativen Elementen ist dabei gegenüber Plotkin's Arbeit eine signifikante Erweiterung, da die bei Plotkin's λ_v-Kalkül gegebene referentielle Transparenz wegfällt. Die Hinzunahme der imperativen Elemente eröffnet zugleich eine weitere Differenzierungsebene in Form der benutzen Strategie zur Parameterübergabe. In [CF91] werden verschiedene Parameter-Übergabemechanismen wie Pass-by-worth und Pass-by-reference in der Gegenwart von call-by-value wie call-by-name Auswertung untersucht, wobei die benutzte operationale Grundlage die Formulierung von Zuweisungen erlaubt. Eines der gezeigten Ergebnisse ist, daß verschiedene Parameter-Übergabemechanismen in dieser nicht referentiell transparenten Umgebungen unterschiedliche Eigenschaften aufweisen, obwohl die Auswertungsstrategie beibehalten wird. Eine weitere interessante in [CF91] aufgeführte Beob-

achtung ist, daß call-by-value und call-by-name Auswertung bei der Gegenwart imperativer Elemente unterschiedliche Ergebnisse liefern können.

Im Umfeld des Themenkreises, call-by-value, call-by-name, call-by-need Auswertung und Nichtdeterminismus wurden in jüngster Zeit mehrere Dissertationen ([Las98], [Mor98], [Pit99]) angefertigt. Lassen operiert in seiner Dissertation [Las98] mit einer Variante von Plotkin's FPC, einem monomorph getypten Fragment von ML [MTH90]. Die operationale Semantik ist call-by-value. In einem zweiten Teil seiner Dissertation untersucht er eine um erratic Nichtdeterminismus wie auch *amb* erweiterte Form dieser Sprache. Interessant an Lassen's Dissertation ist ihr starker operationaler Bezug, den auch Moran's und diese Dissertation aufweisen. Ohne die Zuhilfenahme einer denotationalen Semantik werden auf Basis von Relationen, die eine rein operationale Basis aufweisen, starke Aussagen getroffen. Wie bei den meisten anderen Arbeiten besitzt die kontextuelle Äquivalenz dabei eine herausragende Bedeutung.
Moran sagt selbst von seiner Dissertation [Mor98], daß sie komplementär zu Lassen's Arbeit ist. Seine Untersuchungen sind im Bereich call-by-name und call-by-need in Kombination mit Nichtdeterminismus auf der Basis von *amb* angesiedelt. Viele seiner Methodiken hat er von Lassen übernommen. Moran trifft in seiner Dissertation bezüglich erratic Nichtdeterminismus die Aussage, daß sich erratic-choice mittels *amb* nachahmen ließe, ohne jedoch einen Beleg dafür zu geben.
Im Bereich von erratic Nichtdeterminismus im Zusammenhang mit call-by-name Auswertung ist Pitcher's Dissertation [Pit99] angesiedelt. Die operationale Basis dieser Arbeit ist eine erweiterte Version des berechenbaren Lambda-Kalküls von Moggi [Mog89][Mog91]. Die signifikanteste Erweiterung ist dabei die Einführung von erratic Nichtdeterminismus, der jedoch nicht wie üblich mittels eines binären `choice`-Operators dargestellt wird. Stattdessen werden nichtdeterministische Ausdrücke aus einer Menge von Ausdrücken durch Voranstellen eines Fragezeichens gebildet, z.B. ?{1,2,3}. Der Hauptgegenstand von Picher's Arbeit ist die Gegenüberstellung von Sprachen mit einem unterschiedlichen Maß an Nichtdeterminismus.

Ein eigenes Forschungsfeld, das im Bezug zur vorliegenden Arbeit zu erwähnen ist, sind "explicit substitutions". Der Begriff geht auf [ACCL90] zurück. In diesem Papier wird ein "explicit substitution calculus" eingeführt. Zusätzlich werden ein Typsystem sowie Anwendungen vorgestellt und eine abstrakte Maschine entwickelt. Eine Übersicht über alle Papiere, die in diesem Bereich bisher publiziert wurden, gibt [Ros98].

Die Idee hinter explicit substitutions ist, den Ersetzungsprozess der Beta-Reduktion des klassischen Lambda-Kalküls nicht direkt durchzuführen, sondern die Ersetzung in Form eines syntaktischen Zusatzes zu erfassen. Die Beta-Reduktion erhält das Aussehen

$$(\lambda x.s)t \xrightarrow{Beta} s[(t/x) \cdot id]$$

Dabei wird $[(t/x) \cdot id]$ als syntaktischer Bestandteil des Ausdrucks angesehen. Ein komplizierter Kalkül erlaubt es dann, Substitutionen auf syntaktischer Ebene zusammenzufassen, aufzusplitten und vieles mehr. Auf diese Weise können Aussagen über die β-Reduktion getroffen werden, ohne den expliziten Ersetzungsprozess auf den Ausdrücken durchzuführen. Dabei werden potentiell auftretende Namenskonflikte bei Variablen durch die Verwendung der De Bruijn-Notation [DB72] vermieden.

Der Bezug zur vorliegenden Arbeit wird dadurch hergestellt, daß die Schreibweise mit den angehefteten Substitutionen wie eine andere Darstellung der Let-Schreibweise des Λ_{let}-Kalküls und damit wie eine andere Darstellungsweise für Sharing aufgefaßt werden kann. Tatsächlich wird dieser Bezug auch von Autoren in diesem Forschungsfeld gesehen, wie die Aufführung von [AFM$^+$95] in [Ros98] zeigt. Jedoch ist anzumerken, daß die in [ACCL90] vorgestellten Kalkülregeln nicht mit call-by-need Auswertung vereinbar sind, da beispielsweise eine Regel vorkommt, die es erlaubt, eine Substitution über ein λ hinweg nach innen zu bewegen. Die Ursache hierfür ist, daß die Motivationen für die Betrachtung von explicit substitutions andere sind, als beim call-by-need Kalkül aus [AFM$^+$95]. Ein Anwendungsgebiet von explicit substitutions ist beispielsweise die Entwicklung optimaler Reduktionsstrategien beim klassischen Lambda-Kalkül.

1.5 Überblick

Die Einleitung soll mit einer Kurzübersicht über den Inhalt der einzelnen Kapitel dieser Dissertation abgeschlossen werden.

Im zweiten Kapitel werden zunächst einige grundlegende Definitionen sowie Lemmata eingeführt. Im Anschluß daran wird der Λ_{let}-Kalkül, ein call-by-need Lambda-Kalkül mit einer `choice`-Konstante, deren operationale Semantik erratic Nichtdeterminismus ist, definiert und für diesen gezeigt, daß er eine spezielle Form der Konfluenz erfüllt. Auf algorithmischer Basis wird dann für den Λ_{let}-Kalkül eine normal-order Reduktion (no-Reduktion) sowie eine WHNF (schwache Kopfnormalform) für Ausdrücke vereinbart. Am Ende des zweiten Kapitels werden verschiedene grundlegende Eigenschaften der no-Reduktion bewiesen.

Das dritte Kapitel führt eine kontextuelle Präordnung $\leq_c$ ein, auf deren Basis der Begriff der kontextuellen Äquivalenz definiert wird. Als Hilfsmittel zur Vereinfachung späterer Beweise wird ein Kontextlemma vorgestellt. Anschließend wird für verschiedene Reduktionen bewiesen, daß sie die kontextuelle Äquivalenz erhalten. Die so gezeigten Eigenschaften werden später für Optimierungen, Betrachtung von Rekursion sowie die Vorstellung einer abstrakten Maschine passend zum Λ_{let}-Kalkül genutzt. Zum Abschluß des dritten Kapitels wird gezeigt, daß Lambda-Lifting eine zulässige Optimierung beim Λ_{let}-Kalkül ist.

Im darauf folgenden vierten Kapitel werden deterministische Subausdrücke definiert und bewiesen, daß sich diese bei Erhaltung der kontextuellen Äquivalenz stets kopieren lassen. Auf dieser Basis wird die Zulässigkeit des Liftens von deterministischen Subausdrücken gezeigt. Es werden die Beschränkungen aufgezeigt, die das Liften deterministischer Subausdrücke gegenüber dem in [PJ87] beschriebenen fully-lazy Lambda-Lifting aufweist. Abschließend wird bewiesen, daß bei einer rein deterministischen Sicht die no-Reduktion des Λ_{let}-Kalküls mit dem call-by-name Evaluator des klassischen Lambda-Kalküls austauschbar ist.

Im Zentrum des fünften Kapitels stehen Betrachtungen von Rekursion beim Λ_{let}-Kalkül. Der Zugang zu Rekursion erfolgt dabei auf rein operationaler Ebene. Zunächst wird ein später benötigtes Hilfslemma gezeigt, das besagt, daß alle Ausdrücke ohne Reduktion zu WHNF kleinste Elemente bezüglich der Präordnung $\leq_c$ sind. Auf Basis der Präordnung $\leq_c$ werden die Begriffe kleinste obere Schranke und kleinste kontextuelle obere Schranke gebildet. Der Begriff des Fixpunktkombinators wird definiert und gezeigt, daß Turing's Fixpunktkombinator Y_T für Abstraktionen und die Konstante `choice` einen Fixpunkt bildet. Abschließend wird für Y_T ein "*club*-Beweis" geführt. Dieser zeigt, daß Y_T eine gesuchte kleinste obere Schranke bei Abstraktionen und der Konstante `choice` bildet.

Im sechsten Kapitel wird eine zur no-Reduktion des Λ_{let}-Kalküls passende abstrakte Maschine $M_{\Lambda_{let}}$ vorgestellt. Die entwickelte abstrakte Maschine wird verwandten abstrakten Maschinen gegenübergestellt und Unterschiede werden herausgearbeitet. In einem zweiten Teil wird eine optimierte Form von $M_{\Lambda_{let}}$ vorgestellt und gezeigt, daß die unoptimierte und optimierte Form wechselseitig austauschbar sind.

Das siebte Kapitel berichtet über ein maschinelles Werkzeug, das zur Beweisunterstützung im Zusammenhang mit der vorliegenden Arbeit entwickelt wurde. Es wird über den Aufbau und die Arbeitsweise des Werkzeugs berichtet, sowie der Wert des letzteren dargelegt. Die Ergebnisse einer maschinellen Gegenüberstellung der no-Reduktion des Λ_{let}-Kalküls und des call-by-need Kalküls aus [AFM$^+$95] werden vorgestellt.

Die Dissertation wird mit einer Zusammenfassung und einem Ausblick auf weitere Möglichkeiten der Forschung abgeschlossen.

Kapitel 2

Der Λ_{let}-Kalkül

In diesem Kapitel werden wir einen nichtdeterministischen call-by-need Lambda-Kalkül kennenlernen, den wir als Λ_{let}-Kalkül bezeichnen werden. Der Λ_{let}-Kalkül entspricht dem in [AFM$^+$95] vorgestellten call-by-need Kalkül erweitert um eine `choice`-Regel, deren operationale Semantik erratic Nichtdeterminismus ist.
Zu Beginn werden zunächst eine Reihe von grundlegenden Begriffen vorgestellt, deren Kenntnis zum Verständnis des Λ_{let}-Kalküls erforderlich ist. So werden wir die Menge der Λ_{let}-Ausdrücke kennenlernen und den Begriff des Kontexts definieren. Wir werden in Anlehnung an [Bar84] einige Variablenkonventionen vereinbaren, um den Umgang mit Variablen zu erleichtern, und den Begriff der Substitution für Λ_{let}-Ausdrücke festlegen. Anschließend wird der Begriff der Reduktion vorgestellt und einige Eigenschaften von Reduktionen herausgestellt. Mit einer Kurzvorstellung von König's Lemma wird der Grundlagenteil abgeschlossen.
Im Anschluß wird der Λ_{let}-Kalkül formal definiert werden. Eine Eigenschaft, die bei Kalkülen von großem Interesse ist, ist ihr Konfluenzverhalten. Wir werden zeigen, daß der Λ_{let}-Kalkül eine spezielle Form der Konfluenz, die wir als Mengenkonfluenz bezeichnen, erfüllt. Danach wird auf Basis eines no-Markierungsalgorithmus eine normal-order Reduktion für den Λ_{let}-Kalkül definiert sowie zwei Ausdrucksformen als WHNF (schwache Kopfnormalform) und VWHNF herausgestellt. Der Begriff WHNF wurde dabei in Anlehnung an [Abr90] gewählt.
Der no-Markierungsalgorithmus sagt wenig über die Struktur der no-Redizes aus, d.h. es ist aus der algorithmischen Definition nicht ersichtlich, welches Aussehen die Kontexte aufweisen, in denen sich die verschiedenen Arten von no-Redizes befinden. Diese Kenntnis ist jedoch im Zusammenhang mit Beweisen von Eigenschaften der no-Reduktion von großem Interesse. In Satz 2.3.1 wird bewiesen, welches Aussehen die verschiedenen Typen von no-Redizes bzw. ein Ausdruck in WHNF oder VWHNF besitzt.
Wir werden anschließend zeigen, daß zwischen den Begriffen no-Reduktion und WHNF bei geschlossenen Ausdrücken eine enge Beziehung besteht. Existiert ausgehend von einem Ausdruck t irgendeine Reduktion zu einem Ausdruck t' in

WHNF, dann existiert auch eine no-Reduktion zu einem Ausdruck in WHNF. Die Existenzquantifizierung der no-Reduktion ist dabei eine Folge des Nichtdeterminismus, der bewirkt, daß die no-Reduktion nicht eindeutig ist.
In diesem Zusammenhang werden wir die Begriffe Gabel- und Vertauschungsdiagramm kennenlernen, die uns in vielen Beweisen begegnen werden.

2.1 Grundlagen

2.1.1 Λ_{let}-Ausdrücke

Zur Definition der Menge aller Λ_{let}-Ausdrücke übernehmen wir die in [Bar84] benutzte Methodik zur Definition aller Lambda-Ausdrücke.

Definition 2.1.1. *Folgende syntaktische Elemente seien gegeben:*

- *eine unendliche Menge von Variablen* $x, y, z, \ldots$
- *die Konstante '*`choice`*'*
- *der Lambda Abstraktor 'λ'*
- *die Klammersymbole '(' und ')'*
- *die beiden Wörter '*`let`*' und '*`in`*'*

Die Menge von Ausdrücken Λ_{let} wird wie folgt induktiv[1] *auf den obigen syntaktischen Elementen definiert:*

Variable: $x \in \Lambda_{let}$
nd-Konstante: $\texttt{choice} \in \Lambda_{let}$
Abstraktion: $s \in \Lambda_{let} \Rightarrow (\lambda x.s) \in \Lambda_{let}$
Applikation: $s, t \in \Lambda_{let} \Rightarrow (st) \in \Lambda_{let}$
let-Ausdruck: $s, t \in \Lambda_{let} \Rightarrow (\texttt{let}\ x = s\ \texttt{in}\ t) \in \Lambda_{let}$

Dabei ist x eine beliebige Variable.

Notation 2.1.1.

- *Die Symbole $r, s, t \ldots$ bezeichnen beliebige Λ_{let}-Ausdrücke.*

---

[1] Wird eine induktive Definition gegeben, so wird immer die schwächste Klasse definiert, die die gegebenen Bedingungen erfüllt.

- *Die Symbole* $x, y, z, \ldots$ *bezeichnen beliebige Variablen.*
- *Die äußersten Klammern können bei der Spezifikation eines* Λ_{let}*-Ausdrucks fortgelassen werden.*
- *Applikationen können ohne Angabe von Klammern aneinandergefügt werden. Die Applikationen sind dann per Vereinbarungen links-assoziativ geklammert.* $s_1 s_2 s_3$ *bezeichnet somit den Ausdruck* $((s_1 s_2) s_3)$.
- *Bindungsbereiche und Umbenennungen von Variablen seien genauso gewählt, wie in [Bar84] beim klassischen Lambda-Kalkül. Zwei Ausdrücke sind somit äquivalent, wenn durch eine konsistente Umbenennung von Variablen die syntaktische Äquivalenz beider Ausdrücke hergestellt werden kann, d.h. beide Ausdrücke* α*-konvertibel sind.*
  Zur Spezifikation der Äquivalenz wird der Operator $\equiv$ *benutzt.*
- $V(t)$ *bezeichnet die Menge aller Variablen, die in einem Ausdruck* t *vorkommen.*

Die fünfte der obigen Vereinbarungen, die ein Aussage bezüglich der Bindungsbereiche von Variablen trifft, besagt insbesondere, daß let-Ausdrücke keine zyklischen Bindungen formulieren können. Ferner sind die Ausdrücke aus Λ_{let} ungetypt, im Gegensatz zum Beispiel zu PCF [Plo77].

2.1.2 Kontexte

Passend zu den Λ_{let}-Ausdrücken werden wir nun den Begriff des Kontexts definieren.

Definition 2.1.2. *s sei ein beliebiger* Λ_{let}*-Ausdruck. Ein Kontext* C *ist ein* Λ_{let}*-Ausdruck mit einem Loch* $[\cdot]$ *und folgenden syntaktischen Bildungsgesetzen:*

$$C ::= [\cdot] \mid \lambda x.C \mid (Cs) \mid (sC) \mid (\texttt{let}\ x = s\ \texttt{in}\ C) \mid (\texttt{let}\ x = C\ \texttt{in}\ s)$$

Ein Kontext wird durch eines der Symbole $C, D, \ldots$ *bezeichnet.* $C[s]$ *entspricht dem Ausdruck, der erhalten wird, wenn das Loch* $[\cdot]$ *des Kontexts* C *durch den Ausdruck* s *ersetzt wird.* $C[D]$ *entspricht dem Kontext, der erhalten wird, wenn das Loch* $[\cdot]$ *des Kontexts* C *durch den Kontext* D *ersetzt wird.*

Beispiel 2.1.1. Gegeben sei der Kontext $C \equiv \texttt{let}\ x = \lambda z.z\ \texttt{in}\ \lambda y.[\cdot]$. Wenn $s \equiv xy$, dann entspricht $C[s]$ dem Ausdruck $\texttt{let}\ x = \lambda z.z\ \texttt{in}\ \lambda y.xy$. Dabei werden die in s freien Variablen x und y in $C[s]$ gebunden.

Obiges Beispiel zeigt, daß durch die Ersetzung des Loches eines Kontexts freie Variablen gebunden werden können. Die Begriffe freie und gebundene Variable werden in dem Abschnitt über Variablenkonventionen vorgestellt.

Es wird nun eine Möglichkeit bereitgestellt, bei der Spezifikation eines Kontexts gleichzeitig Informationen über dessen Struktur mitzuspezifizieren:

Definition 2.1.3. *s sei ein Λ_{let}-Ausdruck. Wir treffen folgende feste Zuordnung zwischen Symbolen und Kontexten:*

$$L_R ::= \texttt{let}\ x = s\ \texttt{in}\ [\cdot],\ L_L ::= \texttt{let}\ x = [\cdot]\ \texttt{in}\ s,\ A_L ::= ([\cdot]s),\ A_R ::= (s[\cdot])$$

S sei eines der Symbole L_R, L_L, A_L, A_R. Wir vereinbaren folgende zusätzliche Metasyntax:

$$S^* ::= [\cdot] \mid S[S^*], \qquad S^+ ::= S \mid S[S^+]$$

$$S^{0/1} ::= [\cdot] \mid S\ , \qquad S^0 ::= [\cdot], \qquad S^n ::= S[S^{n-1}]\ \textit{wobei}\ n > 0$$

Die oben eingeführte Syntax erlaubt auf einfache Weise auszudrücken, daß ein Kontext bestimmten strukturellen Anforderungen genügt.

Beispiel 2.1.2. Die folgende Tabelle erläutert an 4 Beispielen die eingeführte Syntax:

Beschreibung	Beispielkontext
$L_R[A_L]$	$\texttt{let}\ x = s\ \texttt{in}\ ([\cdot]t)$
L_R^*	$\texttt{let}\ x_1 = t_1\ \texttt{in}\ \ldots(\texttt{let}\ x_n = t_n\ \texttt{in}\ [\cdot])$ wobei $n \geq 0$
$A_L^+[A_R^+]$	$((\ (t_1(t_2 \ldots (t_m[\cdot])))\ s_1) \ldots s_n)$ wobei $m, n > 0$
A_L^3	$(([\cdot]t_1)t_2)t_3$

Die Eigenschaft von Symbolen wie beispielsweise L_R^*, gleichzeitig einen Kontext zu bezeichnen und eine Aussage über dessen Struktur zu treffen, benutzen wir zur Vereinbarung einer besonderen Sprechweise:

Notation 2.1.2. *S sei ein Symbol, das dazu benutzt, gleichzeitig einen Kontext zu spezifizieren und eine Aussage über dessen Struktur zu treffen. Wir sprechen von einem S-Kontext wenn ein Kontext der zu S vereinbarten Struktur gehorcht.*

Abschließend werden zwei weitere wichtige Kontextfamilien eingeführt.

Definition 2.1.4. *Den Symbolen R und W werden Kontexte zugeordnet, die folgenden Bildungsgesetzen gehorchen:*

$$R ::= [\cdot] \mid A_L[R] \mid A_R[R] \mid L_L[R] \mid L_R[R]$$

$$W ::= L_R^*[A_L^*]\ \mid L_R^*[\texttt{let}\ x = A_L^*\ \texttt{in}\ W[x]]$$

Eine alternative Beschreibung der W-Kontexte lautet

$W ::= L_R^*[L_L^{0/1}[A_L^*]]$ mit $L_L \equiv \texttt{let}\ x = [\cdot]\ \texttt{in}\ W[x]$,

was einfach nachprüfbar ist. Die Vorstellung der zweiten Schreibweisen für W-Kontexte geschieht alleinig aus beweistechnischen Gründen. In einigen der später geführten Beweise ist die erste Schreibweise vorteilhaft in anderen die zweite. W-Kontexte werden eine Schlüsselfunktion bei der späteren Betrachtung von no-Reduktionen einnehmen.
Jeder R-Kontext besitzt die Eigenschaft, daß sich das Loch nicht innerhalb einer Abstraktion befindet. Abstraktionen werden später als kopierbare Ausdrücke angesehen. Ein R-Kontext sichert somit, daß sich das Loch nicht in einem kopierbaren Subausdruck befindet.

Korollar 2.1.1. *Jeder W-Kontext ist zugleich ein R-Kontext.*

2.1.3 Variablenkonventionen

Wie beim klassischen Lambda-Kalkül ist auch beim Λ_{let}-Kalkül eine gewisse Vorsicht beim Umgang mit Variablen notwendig, um unerwünschte Seiteneffekte bei der Definition der operationalen Semantik auszuschließen. Wir werden zunächst nach freien und gebundenen Variablen differenzieren und dann Konventionen einführen, die den weiteren Umgang mit Variablen erleichtern.

Definition 2.1.5. *t sei ein beliebiger Λ_{let}-Ausdruck. $V_{free}(t)$ bezeichnet die Menge aller freien Variablen in t und ist wie folgt induktiv definiert:*

$$\begin{array}{rcl}
V_{free}(x) & := & \{x\} \\
V_{free}(\texttt{choice}\ ) & := & \emptyset \\
V_{free}(\lambda x.t) & := & V_{free}(t) - \{x\} \\
V_{free}(st) & := & V_{free}(s) \cup V_{free}(t) \\
V_{free}(\texttt{let}\ x = s\ \texttt{in}\ t) & := & V_{free}(s) \cup (V_{free}(t) - \{x\})
\end{array}$$

Ein Ausdruck t heißt geschlossen, wenn $V_{free}(t) = \emptyset$. Eine Variable x kommt in einem Ausdruck t frei vor, falls $x \in V_{free}(t)$.

Neben den freien Variablen werden nun die gebundenen Variablen eingeführt.

Definition 2.1.6. *t sei ein beliebiger Λ_{let}-Ausdruck. $\equiv_{id}$ bezeichne die syntaktische Äquivalenz zweier Ausdrücke unter Ausschluß jeglicher Umbenennung von Variablen. Es werden folgende Definitionen getroffen:*

$$\begin{array}{lll} V_{let}(t) & := & \{x \mid \exists \text{ Kontext } C : t \equiv_{id} C[\texttt{let } x = t_x \texttt{ in } s]\} \\ V_{abstr}(t) & := & \{x \mid \exists \text{ Kontext } C : t \equiv_{id} C[\lambda x.s]\} \\ V_{bound}(t) & := & V_{abstr}(t) \cup V_{let}(t) \end{array}$$

Eine Variable x kommt in einem Ausdruck t gebunden vor, wenn $x \in V_{bound}(t)$. $V_{let}(t)$ bezeichnen wir auch als die Menge aller Let-gebundenen Variablen.

Eine Variable kann in einem Ausdruck gleichzeitig gebunden und frei vorkommen, was das Beispiel `let` $x =$ (`let` $y = t_y$ `in` t_x) `in` y zeigt. Wenden wir auf den letzteren Ausdruck eine *llet*-Reduktion an (was eine *llet*-Reduktion ist, wird in Abschnitt 2.2 vorgestellt), so erhalten wir den Ausdruck `let` $y = t_y$ `in` (`let` $x = t_x$ `in` y). Offensichtlich hat sich durch den letzteren Schritt das hintere y von einer freien zu einer gebundenen Variable gewandelt. Ein derartiges Verhalten ist wie beim Lambda-Kalkül nicht erwünscht, da es nicht vereinbar mit dem Gedanken ist, daß Ausdrücke modulo konsistenter Umbenennung der gebundenen Variablen gleich sind. Aus diesem Grund werden die folgenden Vereinbarungen getroffen:

Konvention 2.1.1. *Wenn mehrere Λ_{let}-Ausdrücke $t_1, \ldots, t_n$ in einem bestimmten mathematischen Kontext (z.B. ein Definition oder ein Beweis) vorkommen, dann sind in diesen Ausdrücken alle gebundenen Variablen unterschiedlich zu den freien Variablen gewählt.*

Die obige Konvention wurde direkt aus [Bar84] übernommen.

Konvention 2.1.2. *Alle durch eine Abstraktion oder einen Let-Ausdruck formulierte Bindungen erfolgen durch unterschiedliche Variablen.*

Jeder Ausdruck kann durch eine einfache Umbenennung gebundener Variablen jederzeit in eine Form überführt werden, bei der die obigen Konventionen erfüllt werden.

Moral 2.1.1. *Die zwei Konventionen 2.1.1 und 2.1.2 erlauben einen naiven Umgang mit Λ_{let}-Ausdrücken.*

Der Begriff "naiv" ist dabei derart zu verstehen, daß Transformationen oder Substitutionen bei Λ_{let}-Ausdrücken durchgeführt werden können, ohne nach deren Zulässigkeit im Hinblick auf Bindungen von Variablen zu fragen. Die oben aufgeführte Moral wurde aus [Bar84] übernommen, wo diese für den Lambda-Kalkül in Anhang C ausführlich betrachtet wird.

Eine konsequente Anwendung von Konvention 2.1.2 ist nicht unproblematisch, da sie bei Beweisen, in denen komplexere Λ_{let}-Ausdrücke vorkommen, die Benutzung einer großen Zahl verschiedener Symbole für die einzelnen Variablen erforderlich macht. Dieses ist jedoch erfahrungsgemäß der Übersichtlichkeit und Lesbarkeit von Beweisen nicht zuträglich, weshalb die folgende Ausnahmeregelung getroffen wird:

Konvention 2.1.3. *Entstehen durch die Benutzung derselben Variable zur Formulierung mehrerer Bindungen in dem umgebenden mathematischen Kontext keine Mehrdeutigkeiten, d.h. für jede Variable ist zu jeder Zeit eindeutig wie und wo sie gebunden ist, so ist die mehrfache Benutzung in Abschwächung zu Konvention 2.1.2 zulässig.*

2.1.4 Substitution

Nachdem wir einige Variablenkonventionen getroffen haben, definieren wir den Begriff der Substitution.

Definition 2.1.7. *Das Ergebnis der Substitution aller freien Vorkommen einer Variablen x durch einen Ausdruck s in einem Ausdruck t wird durch $t[s/x]$ bezeichnet und wie folgt definiert:*

$$t[s/x] := \begin{cases} s & \text{falls } t \equiv x \\ y & \text{falls } t \equiv y \text{ und } y \not\equiv x \\ \lambda y.(t_1[s/x]) & \text{falls } t \equiv \lambda y.t_1 \\ (t_1[s/x])\,(t_2[s/x]) & \text{falls } t \equiv t_1\,t_2 \\ \texttt{let}\ y = (t_1[s/x])\ \texttt{in}\ (t_2[s/x]) & \text{falls } t \equiv \texttt{let}\ y = t_1\ \texttt{in}\ t_2 \end{cases}$$

Bei der obigen Definition ist es nicht notwendig bei einer Abstraktion wie einem Let-Ausdruck die Bedingung "sofern $x \not\equiv y$" aufzuführen. Gemäß der im vorherigen Abschnitt getroffenen Variablenkonventionen ist dies stets der Fall.

2.1.5 Reduktionen

Wir stellen nun den Begriff der Reduktion vor, der in dieser Arbeit von zentraler Bedeutung ist. Die nun folgenden Begriffsvereinbarungen erfolgen wie in den Abschnitten zuvor in der Tradition von [Bar84].

Definition 2.1.8. *Eine Reduktion auf der Menge der Λ_{let}-Ausdrücke entspricht einer binären Relation auf der Menge der Λ_{let}-Ausdrücke. Den Begriff der Reduktion charakterisiert, daß die Definition der binären Relation auf Reduktionsregeln basiert, vergleichbar der β-Reduktion des klassischen Lambda-Kalküls.*

Notation 2.1.3. *Zur Spezifikation von Reduktionen wird das Pfeilsymbol $\rightarrow$ in Kombination mit einer infix-Schreibweise verwendet, wobei oberhalb des Pfeils Auskünfte über Name, Bildung und Aufbau der Reduktion spezifiziert werden können. Gegeben eine Reduktion $\rightarrow$ sowie zwei Ausdrücke s, t mit $s \rightarrow t$.*
Den Übergang von s zu t bezeichnen wir als Reduktionsschritt *oder abgekürzt* Reduktion. *Wir benutzen die Sprechweise "s reduziert zu t".*

Gemäß obiger Notation kann der Begriff Reduktion sowohl eine binäre Relation wie auch einen Reduktionsschritt bezeichnen. Die jeweils zutreffende Bedeutung des Begriffs Reduktion wird im weiteren stets aus dem Kontext ablesbar sein, in dem der Begriff benutzt wird.
Wie auch beim klassischen Lambda-Kalkül definieren wir eine Reihe von Abschlüssen für Reduktionen.

Definition 2.1.9. *Sei* $\rightarrow$ *eine Reduktion, die drei Abschlüsse,* $\xrightarrow{+}$, $\xrightarrow{0/1}$, $\xrightarrow{*}$ *werden wie üblich als transitiver* $(+)$*, reflexiver* $(0/1)$ *sowie transitiv reflexiver* $(*)$ *Abschluß definiert.*

Eine wichtige Eigenschaft von binären Relationen (Reduktionen) auf der Menge aller Λ_{let}-Ausdrücke ist ihre Ausdehnbarkeit auf umgebende Kontexte, da dies erlaubt, von der Beziehung zweier Ausdrücke auf das Verhalten einer viel größeren Menge von Ausdrücken zu schließen. Diese Eigenschaft wird in dem Begriff der Kompatibilität erfaßt.

Definition 2.1.10. *R sei eine binäre Relation auf der Menge aller Λ_{let}-Ausdrücke.*

1. *R ist* kompatibel mit allen Kontexten *wenn für alle Kontexte C:*
$$s\,R\,t \;\Rightarrow\; C[s]\,R\,C[t].$$
2. *R ist eine* Präkongruenz-Relation *bzw. (nach [Bar84])* Reduktions-Relation*, wenn R reflexiv, transitiv und kompatibel ist.*
3. *R ist eine* Kongruenz-Relation*, wenn R eine kompatible Äquivalenzrelation ist.*

Die β-Reduktion des klassischen Lambda-Kalküls ist kompatibel, die später in diesem Abschnitt definierte `choice` -Reduktion des Λ_{let}-Kalküls wird jedoch über keine Kompatibilität verfügen.

Ein weiterer wichtiger Begriff, der nun eingeführt wird, ist der des Redex. Informal gesprochen ist ein Redex ein reduzierbarer Subausdruck, der sich innerhalb eines umgebenden Kontexts befindet. Wir verankern die formale Definition des Begriffs Redex hier an der Form der Definition von Reduktionen, wie sie in der vorliegenden Arbeit erfolgt.

Definition 2.1.11. *Besitzt die Definition einer Reduktion die Form $S[t] \rightarrow S[t']$ für irgendeine Kontextform S, so bezeichnen wir t in einem Ausdruck $S[t]$ als Redex oder reduzierbaren Subausdruck.*

Konfluenz, Church-Rosser-Eigenschaft, Austauschbarkeit

Eine wichtiges Beurteilungskriterium von Reduktionen (Reduktionskalkülen) ist ihr Konfluenzverhalten. Was darunter zu verstehen ist, werden wir jetzt kennenlernen.

Definition 2.1.12. *Eine Reduktion* $\rightarrow$ *ist* lokal konfluent*, gdw. für beliebige Reduktionen* $a \rightarrow b$ *und* $a \rightarrow c$ *ein* d *existiert, so daß* $b \xrightarrow{*} d$ *und* $c \xrightarrow{*} d$ *gilt.*

Eine Erweiterung der lokalen Konfluenz ist die Church-Rosser-Eigenschaft.

Definition 2.1.13. *Eine Reduktion* $\rightarrow$ *ist* konfluent *bzw. besitzt die* Church-Rosser-Eigenschaft*, gdw. für beliebige Reduktionen* $a \xrightarrow{*} b$ *und* $a \xrightarrow{*} c$ *ein* d *existiert, so daß* $b \xrightarrow{*} d$ *und* $c \xrightarrow{*} d$ *gilt.*

Die nun definierte Austauschbarkeit ist geeignet, eine Verbindung zwischen zwei Reduktionen herzustellen.

Definition 2.1.14. *Zwei Reduktionen* $\rightarrow_1$ *und* $\rightarrow_2$ *sind* austauschbar*, wenn für beliebige Elemente* a, b, c *mit* $a \xrightarrow{*}_1 b$ *und* $a \xrightarrow{*}_2 c$ *ein Element* d *existiert, so daß* $b \xrightarrow{*}_2 d$ *und* $c \xrightarrow{*}_1 d$ *gilt.*

Alle obigen Definitionen sind nicht auf Ausdrücke beschränkt, sie sind vielmehr auf eine beliebige Menge, für die eine Reduktion definiert wurde, anwendbar. Die Begriffe lokal konfluent, konfluent, austauschbar sind somit auch im Falle einer Reduktion anwendbar, die nicht auf einzelnen Λ_{let}-Ausdrücken, sondern auf Mengen von Λ_{let}-Ausdrücke definiert ist. Wir werden die Austauschbarkeit zweier Reduktionen später in einem Konfluenzbeweis als Beweishilfsmittel benutzen.

2.1.6 Lemma von König

Ein in vielen Beweisen der vorliegenden Arbeit benutztes Lemma ist das aus der Graphentheorie wohlbekannte sogenannte Lemma von König. König's Lemma lautet

Lemma 2.1.1. *(König's Lemma)*

In jedem unendlichen, endlich verzweigenden Baum gibt es einen unendlichen Pfad.

wobei ein unendlicher Baum ein Baum mit einer nicht endlich beschränkten Zahl an Knoten ist. Der Beweis dieses Lemma's kann in Standardwerken der Logik

wie [Fit90] oder [Gal87] nachgeschlagen werden. König's Lemma verdankt seinen Namen vermutlich Denis König, der Autorin von [Kön36].
König's Lemma wird in der vorliegenden Arbeit zumeist im Kontext von Divergenzbeweisen benutzt. Häufig kann für einen Ausdruck t gezeigt werden, daß für jede natürliche Zahl n eine Reduktion ausgehend von t existiert, die über mindestens n Einzelreduktionen verfügt. Eigentliches Ziel ist jedoch, zu zeigen, daß ausgehend von t eine unendliche Reduktion existiert. Indem die no-Reduktion ausgehend von t als Baum interpretiert wird, kann mittels König's Lemma kann aus ersterem letzteres gefolgert werden, da

- der Verzweigungsgrad des Baumes endlich ist; jeder Ausdruck kann mittels einer no-Reduktion zu maximal zwei Nachfahren reduziert werden.
- die Anzahl der Knoten unendlich ist, da für jede natürliche Zahl n eine Reduktion angegeben werden kann, die über mindestens n Einzelreduktionen verfügt.

2.2 Einführung des Λ_{let}-Kalküls

In dem nun folgenden Abschnitt werden wir den Λ_{let}-Kalkül kennenlernen und zeigen, daß er eine spezielle Form der Konfluenz erfüllt. Wir beginnen, indem wir die Reduktionsregeln des Λ_{let}-Kalküls spezifizieren.

Definition 2.2.1. *Der nichtdeterministische Kalkül Λ_{let} umfaßt die folgenden Reduktionsregeln:*

$(llet)$ $C[(\texttt{let}\ x = (\texttt{let}\ y = t_y\ \texttt{in}\ t_x)\ \texttt{in}\ s)] \xrightarrow{llet} C[(\texttt{let}\ y = t_y\ \texttt{in}\ (\texttt{let}\ x = t_x\ \texttt{in}\ s))]$
$(lapp)$ $C[((\texttt{let}\ x = t_x\ \texttt{in}\ s)t)] \xrightarrow{lapp} C[(\texttt{let}\ x = t_x\ \texttt{in}\ (st))]$
$(lbeta)$ $C[((\lambda x.t)s)] \xrightarrow{lbeta} C[(\texttt{let}\ x = s\ \texttt{in}\ t)]$
$(nd, left)$ $R[\texttt{choice}\ s] \xrightarrow{nd,left} R[(\lambda x.(\lambda y.x))s]$ *wobei* $(*)$
$(nd, right)$ $R[\texttt{choice}\ s] \xrightarrow{nd,right} R[(\lambda x.(\lambda y.y))s]$ *wobei* $(*)$
(cp) $C[(\texttt{let}\ x = r\ \texttt{in}\ D[x])] \xrightarrow{cp} C[(\texttt{let}\ x = r\ \texttt{in}\ D[r'])]$ *wenn* $(**)$

$(*)$ *x, y sind frische Variablen*
$(**)$ *r ist eine Abstraktion oder die Konstante* `choice` *und r' ist eine umbenannte Kopie von r.*

Die Reduktionsregeln des Λ_{let}-Kalküls entsprechen bis auf die $(nd, left)$- und $(nd, right)$- Reduktionen den Reduktionsregeln des call-by-need Kalküls aus [AFM$^+$95]. Eine leichte Variation betrifft die *cp*-Reduktion, die hier auch die

Konstante `choice` kopieren kann. Für eine ausführliche Erläuterung der Reduktionen sei auf [AFM$^+$95] und [MOW98] verwiesen. Dort wird die Wirkungsweise der einzelnen Reduktionen ausführlich beschrieben und anhand von Grafiken erläutert.
Eine besondere Bedeutung kommt den beiden Reduktionen $(nd, left)$ und $(nd{,}right)$ zu, da diese den Nichtdeterminismus auf Kalkülebene wiedergeben. Offensichtlich ist jeder $(nd, left)$-Redex zugleich ein $(nd, right)$-Redex und umgekehrt, was eine Auswahlmöglichkeit bewirkt. Dies hat insbesondere zur Folge, daß die später in diesem Abschnitt definierte no-Reduktion zwar stets einen eindeutigen no-Redex bestimmt, jedoch in ihrer Gesamtheit nicht eindeutig ist, da im Falle eines no-Redex der Form $R[\texttt{choice}\ s]$ nicht bestimmt wird, welche der beiden möglichen Reduktionsregeln anzuwenden ist. Es ist ferner offensichtlich, daß die beiden nd-Reduktionen bewirken, daß die Reduktion des Λ_{let}-Kalküls in ihrer Gesamtheit nicht konfluent ist.
Einige Worte seien noch bezüglich der Einschränkung auf R-Kontexte bei den beiden nd-Reduktionen gesagt. Diese Einschränkung ist erforderlich, um die später in diesem Abschnitt vorgestellte Mengenkonfluenz zeigen zu können.[2] Mittels *lbeta*-Reduktionen ist es möglich neue nd-Redizes zu schaffen, da diese einen Ausdruck aus einer Abstraktion herausbewegen können. Somit impliziert das Vorkommen eines Subausdrucks der Form (`choice` s) außerhalb eines R-Kontext nicht, daß dieser nicht nach Anwendung anderer Reduktion irgendwann in einer R-Kontext hineinbewegt wird und einen nd-Redex bildet.

Wir führen nun eine zusätzliche Schreibweise ein, die eine einfache Spezifikation von zusammengesetzten Reduktionen beim Λ_{let}-Kalkül ermöglichen soll.

Notation 2.2.1. *Sei $\mathcal{R} = \{\xrightarrow{a_1}, \dots, \xrightarrow{a_n}\}$ eine Menge von n verschiedenen Reduktionen, wobei die einzelnen a_i Reduktionsnamen wie beispielsweise llet oder $(nd, left)$ entsprechen. Mit $\xrightarrow{a_1,\dots,a_n}$ bezeichnen wir die Reduktion die durch die Vereinigung aller Reduktionen der Menge $\mathcal{R}$ entsteht.*

Die neu eingeführte Schreibweise benutzen wir nun um zwei zusammengesetzten Reduktionen feste Symbole zuzuordnen.

Definition 2.2.2. *Zusätzlich vereinbaren wir folgende Reduktionen:*

- $\xrightarrow{nd} \;:=\; \xrightarrow{(nd,left),(nd,right)}$
- $\xrightarrow{let} \;:=\; \xrightarrow{llet,lapp,lbeta}$

---

[2] Die entstehende Problematik zeigt die Gegenüberstellung von call-by-name und call-by-need Auswertung bei der Anwesenheit von Nichtdeterminismus im Rahmen der Einleitung.

Im weiteren werden wir nun einige grundlegende Eigenschaften des Λ_{let}-Kalküls zeigen. Offensichtlich ist der Kalkül, wie bereits erwähnt, nicht konfluent, da durch Anwendung der beiden Alternativen einer nd-Reduktion zwei Ausdrücke t und t' erreicht werden können, die keinen gemeinsamen Nachfahren besitzen auf den sie reduziert werden können.

Es existiert jedoch eine Möglichkeit Konfluenz zu zeigen, wenn die Betrachtung auf Mengen von Ausdrücken ausgedehnt wird. Wir dehnen dazu die Reduktionen des Λ_{let}-Kalküls auf Mengen von Ausdrücken aus.

Definition 2.2.3. *Als Erweiterung der Reduktionen des Λ_{let}-Kalküls definieren wir folgende Reduktionen auf Mengen von Ausdrücken. M sei eine beliebige Menge von Λ_{let}-Ausdrücken, wir vereinbaren:*

1. *Für alle $\rho \in \{llet, lapp, lbeta, cp\}$: wenn $s \xrightarrow{\rho} t$, dann $M \cup \{s\} \xrightarrow{set,\rho} M \cup \{t\}$*
2. $M \cup \{R[\texttt{choice}\ s]\} \xrightarrow{set,nd} M \cup \{R[(\lambda x.\lambda y.x)s], R[(\lambda x.\lambda y.y)s]\}$
3. $\xrightarrow{set,let,nd} \quad := \quad \xrightarrow{set,let} \cup \xrightarrow{set,nd}$
4. $\xrightarrow{set,\Lambda_{let}} \quad := \quad \xrightarrow{set,let,nd} \cup \xrightarrow{set,cp}$

Offensichtlich bewirkt die Definition von $\xrightarrow{set,nd}$, daß beide Alternativen zusammengefaßt betrachtet werden, was die Überwindung der mit den nd-Reduktionen verbundenen Problematik bewirkt.

Notation 2.2.2. *Zum Zwecke der Vereinfachung sprechen wir im weiteren nicht mehr von Reduktionen auf Mengen, sondern benutzen auch bei diesen einfach nur den Begriff Reduktionen.*

2.2.1 Konfluenz von $\xrightarrow{set,\Lambda_{let}}$

Wir werden nun zeigen, daß die Reduktion $\xrightarrow{set,\Lambda_{let}}$ konfluent ist. Da $\xrightarrow{set,\Lambda_{let}}$ auf Mengen von Ausdrücken definiert ist, kann die Konfluenz von $\xrightarrow{set,\Lambda_{let}}$ auch als Mengenkonfluenz bezeichnet werden. Die Struktur des nun folgenden Konfluenzbeweises sieht wie folgt aus:

Wir werden zunächst zeigen, daß die Reduktionen $\xrightarrow{let}$ und $\xrightarrow{set,let,nd}$ lokal konfluent sind. Dies benutzen wir, um zu zeigen, daß die Reduktion $\xrightarrow{set,let,nd}$ konfluent ist. Nach genau demselben Schema wird gezeigt, daß $\xrightarrow{set,cp}$ konfluent ist. Die beiden so erhaltenen Lemmata setzen wir dann mittels eines Lemma's aus [Bar84] zusammen, um die Konfluenz von $\xrightarrow{set,\Lambda_{let}}$ zu erhalten.

Konfluenz von $\xrightarrow{set,let,nd}$

Lemma 2.2.1. *Die Reduktion $\xrightarrow{let}$ ist lokal konfluent.*

Beweis. Zu zeigen ist die lokale Konfluenz im Falle aller 6 möglichen Kombinationen von Redexarten. Dabei sind alle Redex-Kombinationen trivial mit Ausnahme der beiden folgenden:

1. *llet* überlappt mit *llet*

$$\begin{tikzcd}
& C[(\texttt{let}\ x = (\texttt{let}\ y = (\texttt{let}\ z = t_z\ \texttt{in}\ t_y)\ \texttt{in}\ t_x)\ \texttt{in}\ s)] \arrow[dl, "llet"'] \arrow[dr, "llet"] & \\
C[(\texttt{let}\ x = (\texttt{let}\ z = t_z\ \texttt{in}\ (\texttt{let}\ y = t_y\ \texttt{in}\ t_x)\ \texttt{in}\ s)] \arrow[d, "llet"] & & C[(\texttt{let}\ y = (\texttt{let}\ z = t_z\ \texttt{in}\ t_y)\ \texttt{in}\ (\texttt{let}\ x = t_x\ \texttt{in}\ s))] \arrow[dl, "llet"] \\
C[(\texttt{let}\ z = t_z\ \texttt{in}\ (\texttt{let}\ x = (\texttt{let}\ y = t_y\ \texttt{in}\ t_x)\ \texttt{in}\ s))] \arrow[r, "llet"] & C[(\texttt{let}\ z = t_z\ \texttt{in}\ (\texttt{let}\ y = t_y\ \texttt{in}\ (\texttt{let}\ x = t_x\ \texttt{in}\ s)))] &
\end{tikzcd}$$

2. *llet* überlappt mit *lapp*

$$\begin{tikzcd}
& C[(\texttt{let}\ x = (\texttt{let}\ y = t_y\ \texttt{in}\ t_x)\ \texttt{in}\ s)t] \arrow[dl, "llet"'] \arrow[dr, "lapp"] & \\
C[(\texttt{let}\ y = t_y\ \texttt{in}\ (\texttt{let}\ x = t_x\ \texttt{in}\ s))t] \arrow[d, "lapp"] & & C[\texttt{let}\ x = (\texttt{let}\ y = t_y\ \texttt{in}\ t_x)\ \texttt{in}\ (st)] \arrow[dl, "llet"] \\
C[\texttt{let}\ y = t_y\ \texttt{in}\ ((\texttt{let}\ x = t_x\ \texttt{in}\ s)t) \arrow[r, "lapp"] & C[\texttt{let}\ y = t_y\ \texttt{in}\ (\texttt{let}\ x = t_x\ \texttt{in}\ (st))] &
\end{tikzcd}$$

□

Der obige Beweis der lokalen Konfluenz der Reduktion $\xrightarrow{let}$ kann benutzt werden, um die lokale Konfluenz der Reduktion $\xrightarrow{set,let,nd}$ zu zeigen.

Lemma 2.2.2. *Die Reduktion $\xrightarrow{set,let,nd}$ ist lokal konfluent.*

Beweis. Die lokale Konfluenz von $\xrightarrow{let}$ impliziert sofort die lokale Konfluenz von $\xrightarrow{set,let}$. Durch das Hinzukommen der Überlappungsdiagramme für $\xrightarrow{set,nd}$ entstehen keine weiteren nichttrivialen Fälle. Das einzige benötigte, nichttriviale Argument ist, daß durch eine *let*-Reduktion kein `choice` aus einem R-Kontext herausbewegt werden kann. □

Wir werden nun zeigen, daß jede Reduktion, die nur aus *let*- und *nd*-Reduktionen besteht, nach endlich vielen Reduktionsschritten terminiert. Dazu definieren ein Bewertungsmaß φ für Λ_{let}-Ausdrücke.

Definition 2.2.4. *Das Bewertungsmaß φ, das jedem Λ_{let}-Ausdruck eine natürliche Zahl zuordnet, wird wie folgt definiert:*

$$\varphi(s) := \begin{cases} 1 & \text{falls } s \text{ eine Variable} \\ 2 & \text{falls } s \text{ die Konstante } \texttt{choice} \\ 2 * (\varphi(t_1) + \varphi(t_2)) & \text{falls } s \equiv (t_1\, t_2) \\ \varphi(t) & \text{falls } s \equiv \lambda x.t \\ (2 * \varphi(t_1)) + \varphi(t_2) & \text{falls } s \equiv \texttt{let } x = t_1 \texttt{ in } t_2 \end{cases}$$

Wir werden zunächst ein Lemma beweisen, das die Erhaltung von ungleichen Bewertungen durch das Bewertungsmaß φ beim Einschluß in umgebende Kontexte aufzeigt.

Lemma 2.2.3. *Gegeben zwei Ausdrücke t, s mit $\varphi(t) > \varphi(s)$. Dann gilt $\varphi(C[t]) > \varphi(C[s])$ für alle Kontexte C.*

Beweis. Eine einfache strukturelle Induktion über den Aufbau des Kontext C.
Induktionsbeginn:
Der Kontext entspricht dem leeren Kontext. Die Gültigkeit der Hypothese ist in diesem Fall trivial.

Induktionsschritt:
Wir differenzieren nach dem Aufbau des Kontext C. Stellvertretend betrachten wir die Zerlegung $C \equiv C'[\cdot]\, t'$:
Wir wenden die Definition des Bewertungsmaßes φ an und erhalten $\varphi(C'[t]\, t') = 2 * n_t + c$ sowie $\varphi(C'[s]\, t') = 2 * n_s + c$, wobei $n_t = \varphi(C'[t])$, $n_s = \varphi(C'[s])$ und $c = 2 * \varphi(t')$. Gemäß Induktionsvoraussetzung gilt $n_t > n_s$. Daraus folgt direkt $2 * n_t + c > 2 * n_s + c$ und wir haben die Hypothese für die Zerlegung gezeigt.

Bei allen anderen Zerlegungen ist eine ähnliche Argumentation möglich. □

Genauso läßt sich zeigen, daß das Bewertungsmaß φ für alle Ausdrücke positive Werte liefert.

Lemma 2.2.4. *Für alle $t \in \Lambda_{let}$ gilt: $\varphi(t) > 0$*

Beweis. Durch Induktion über die Struktur von t. □

Nun kann gezeigt werden, das jede *let*-Reduktion zu einer Verringerung der Bewertung eines Ausdrucks mittels φ führt.

Lemma 2.2.5. *Gegeben zwei Ausdrücke* t, s. *Wenn* $t \xrightarrow{let,nd} s$ *dann* $\varphi(t) > \varphi(s)$, *d.h. jede let- oder nd-Reduktion erniedrigt die Bewertung eines Ausdrucks.*

Beweis. Lemma 2.2.3 zeigt, daß eine Verkleinerung der Bewertung eines Ausdrucks sich stets in gleicher Weise auf die Bewertung bei Einschluß in einen umgebenden Kontext auswirkt. Gemäß Lemma 2.2.4 liefert φ für alle Ausdrücke eine Bewertung, die echt größer 0 ist. Es bleibt somit zu zeigen, daß die Reduzierung eines Redex vom Typ $llet, lapp, lbeta$ oder nd stets zu einer Reduzierung der Bewertung führt:

$llet$:

$$\begin{array}{rl} & \varphi(\texttt{let}\ x = (\texttt{let}\ y = t_y\ \texttt{in}\ t_x)\ \texttt{in}\ s) = 4\varphi(t_y) + 2\varphi(t_x) + \varphi(s) \\ > & \varphi(\texttt{let}\ y = t_y\ \texttt{in}\ (\texttt{let}\ x = t_x\ \texttt{in}\ s)) = 2\varphi(t_y) + 2\varphi(t_x) + \varphi(s) \end{array}$$

$lapp$:

$$\begin{array}{rl} & \varphi((\texttt{let}\ x = t_x\ \texttt{in}\ s)t) = 4\varphi(t_x) + 2\varphi(s) + 2\varphi(t) \\ > & \varphi(\texttt{let}\ x = t_x\ \texttt{in}\ (st)) = 2\varphi(t_x) + 2\varphi(s) + 2\varphi(t) \end{array}$$

$lbeta$:

$$\begin{array}{rl} & \varphi((\lambda x.s)t) = 2\varphi(t) + 2\varphi(s) \\ > & \varphi(\texttt{let}\ x = t\ \texttt{in}\ s) = 2\varphi(t) + \varphi(s) \end{array}$$

$nd, left$:

$$\begin{array}{rl} & \varphi(\texttt{choice}\ s) = 4 + 2\varphi(s) \\ > & \varphi((\lambda x.\lambda y.x)s) = 2 + 2\varphi(s) \end{array}$$

Für $nd, right$ kann dieselbe Argumentation erfolgen. □

Korollar 2.2.1. *Die Reduktionen* $\xrightarrow{let,nd}$ *und* $\xrightarrow{set,let,nd}$ *sind terminierend.*

Beweis. Die Terminierung von $\xrightarrow{set,let,nd}$ wird erhalten, indem eine Menge von Ausdrücken M mittels der Multimenge aller Bewertungen $\varphi(s)$ aller Ausdrücke s aus M bewertet und die Ordnung für Multimengen[3] angewendet wird. □

Wir sind nun in der Lage zu folgern, daß die Reduktion $\xrightarrow{set,let,nd}$ konfluent ist.

Satz 2.2.1. *Die Reduktionen* $\xrightarrow{let}$ *und* $\xrightarrow{set,let,nd}$ *sind konfluent.*

Beweis. Aus Lemma 2.2.2 und Korollar 2.2.1 folgt in Kombination mit dem Newmann-Lemma [New42] (siehe auch [Hue80]) die Konfluenz der Reduktion $\xrightarrow{let}$ und $\xrightarrow{set,let,nd}$. □

---

[3] Bezüglich Ordnungen auf Multimengen und ihre Eigenschaften sei auf [DM79] verwiesen.

Konfluenz von $\xrightarrow{cp,nd}$

Die Vorgehensweise ist dieselbe wie beim Beweis der Konfluenz von $\xrightarrow{set,let,nd}$.

Lemma 2.2.6. *Die Reduktion $\xrightarrow{cp}$ ist lokal konfluent.*

Beweis. Alleinig wenn zwei *cp*-Redizes überlappen entsteht kein Standarddiagramm:

1. Innerhalb des zu kopierenden Subausdrucks eines *cp*-Redex befindet sich ein anderer *cp*-Redex. t_x sei ein beliebiger Λ_{let}-Ausdruck, mit $t_x \xrightarrow{cp} t'_x$.

$$\begin{array}{c}
C[(\texttt{let}\ x = t_x\ \texttt{in}\ D[x])]\\
\begin{array}{c|c}
\xrightarrow{cp} & \xrightarrow{cp}\\
C[(\texttt{let}\ x = t_x\ \texttt{in}\ D[t_x])] & \\
\xrightarrow{cp} & C[(\texttt{let}\ x = t'_x\ \texttt{in}\ D[x])]\\
C[(\texttt{let}\ x = t'_x\ \texttt{in}\ D[t_x])] & \\
\xrightarrow{cp} & \xrightarrow{cp}
\end{array}\\
C[(\texttt{let}\ x = t'_x\ \texttt{in}\ D[t_x])]
\end{array}$$

2. Das Kopierziel eines *cp*-Redex wird durch einen anderen *cp*-Redex kopiert.

$$\begin{array}{c}
C[\texttt{let}\ x = t_x\ \texttt{in}\ D[\texttt{let}\ y = t_y[x]\ \texttt{in}\ E[y]]]\\
\begin{array}{c|c}
\xrightarrow{cp} & \xrightarrow{cp}\\
C[\texttt{let}\ x = t_x\ \texttt{in}\ D[\texttt{let}\ y = t_y[x]\ \texttt{in}\ E[t_y[x]]]] & \\
\xrightarrow{cp} & C[\texttt{let}\ x = t_x\ \texttt{in}\ D[\texttt{let}\ y = t_y[t_x]\ \texttt{in}\ E[y]]]\\
C[\texttt{let}\ x = t_x\ \texttt{in}\ D[\texttt{let}\ y = t_y[t_x]\ \texttt{in}\ E[t_y[x]]]] & \\
\xrightarrow{cp} & \xrightarrow{cp}
\end{array}\\
C[\texttt{let}\ x = t_x\ \texttt{in}\ D[\texttt{let}\ y = t_y[t_x]\ \texttt{in}\ E[t_y[t_x]]]]
\end{array}$$

□

Wir werden nun zeigen, daß jede Folge von *cp*-Reduktionen terminiert. Um das dazu erforderliche Bewertungsmaß definieren zu können, benötigen wir Umgebungen, die eine Zuordnung zwischen Variablen und Bewertungen festhalten:

Definition 2.2.5. *Eine* Variablenumgebung ρ *ist eine partielle Abbildung, die Variablen natürliche Zahlen zuordnet.*
Gegeben eine Variable x sowie eine natürliche Zahl n. $\rho[n/x]$ bezeichnet die Umgebung ρ', die wie folgt aus ρ erhalten wird:

$$\rho'(y) := \begin{cases} n & \text{falls } y \equiv x \\ \rho(y) & \text{sonst} \end{cases}$$

Definition 2.2.6. *Wir definieren eine Interpretation $[\![s]\!]_\psi\rho$, die jedem Ausdruck $s \in \Lambda_{let}$ eine natürliche Zahl zuordnet:*

$$\begin{array}{rcl}
[\![x]\!]_\psi\rho & := & \left\{\begin{array}{ll} \rho(x) & \text{falls } \rho(x) \text{ definiert} \\ 0 & \text{falls } \rho(x) \text{ undefiniert} \end{array}\right. \\
[\![\texttt{choice}\,]\!]_\psi\rho & := & 1 \\
[\![\lambda x.t]\!]_\psi\rho & := & [\![t]\!]_\psi\rho[0/x] \\
[\![st]\!]_\psi\rho & := & [\![s]\!]_\psi\rho + [\![t]\!]_\psi\rho \\
[\![\texttt{let}\ x = s\ \texttt{in}\ t]\!]_\psi\rho & := & [\![s]\!]_\psi\rho + [\![t]\!]_\psi\rho' \\
 & & \text{wobei } \rho' := \rho[\,[\![s]\!]_\psi\rho + 1\,/x]
\end{array}$$

Für einen Λ_{let}-Ausdruck definieren wir die Bewertungsfunktion $\psi(s) := [\![s]\!]_\psi\emptyset$.

Wir zeigen nun, daß jede *cp*-Reduktion die Bewertung eines Ausdrucks mittels ψ verringert.

Lemma 2.2.7. *Für beliebige Λ_{let}-Ausdrücke t, t': $t \xrightarrow{cp} t' \;\Rightarrow\; \psi(t) > \psi(t')$.*

Beweis. Gegeben eine beliebige Variablenumgebung ρ. Durch strukturelle Induktion über den Aufbau einen Kontexts C zeigen wir zunächst $[\![C[x]]\!]_\psi\rho' > [\![C[s]]\!]_\psi\rho'$, sofern $\rho' := \rho[\,[\![s]\!]_\psi\rho + 1\,/x]$.

Induktionsanfang $C \equiv [\;]$:

Trivialerweise gilt $[\![s]\!]_\psi\rho > [\![s]\!]_\psi\rho + 1$.

Induktionsschritt

Die Hypothese ist für alle Bildungsgesetze eines Kontext C zu zeigen. Wir betrachten stellvertretend den Fall $C \equiv \texttt{let}\ y = t_y\ \texttt{in}\ D[\cdot]$; alle übrigen Fälle werden genauso gezeigt:

Wir betrachten die folgenden beiden Entwicklungen

$$\begin{array}{rl|rl}
 & [\![\texttt{let}\ y = t_y\ \texttt{in}\ D[x]]\!]_\psi\rho' & & [\![\texttt{let}\ y = t_y\ \texttt{in}\ D[s]]\!]_\psi\rho' \\
= & [\![t_y]\!]_\psi\rho' + [\![D[x]]\!]_\psi\rho'' & = & [\![t_y]\!]_\psi\rho' + [\![D[s]]\!]_\psi\rho''
\end{array}$$

mit $\rho'' := \rho'[[\![t_y]\!]_\psi\rho + 1/y]$. Die Hypothese folgt direkt aus der Induktionsvoraussetzung $[\![D[x]]\!]_\psi\rho'' > [\![D[s]]\!]_\psi\rho''$.

Die Hypothese des Lemmas folgt in Kombination mit dem oben gezeigten daraus, daß die Bewertung eines Ausdrucks strikt monoton in Abhängigkeit der Bewertung der Subausdrücke ist. □

Korollar 2.2.2. *Die Reduktionen $\xrightarrow{cp}$ und $\xrightarrow{set,cp}$ sind terminierend.*

Beweis. Die Terminierung von $\xrightarrow{set,cp}$ wird erhalten, indem eine Menge von Ausdrücken M mittels der Multimenge aller Bewertungen $\psi(s)$ aller Ausdrücke s aus M bewertet und die Ordnung für Multimengen angewendet wird. □

Wir sind nun in der Lage zu folgern, daß die Reduktion $\xrightarrow{set,cp}$ konfluent ist.

Satz 2.2.2. *Die Reduktionen* $\xrightarrow{cp}$ *und* $\xrightarrow{set,cp}$ *sind konfluent.*

Beweis. Aus Lemma 2.2.6 und Korollar 2.2.2 folgt in Kombination mit dem Newmann-Lemma [New42] die Konfluenz der Reduktionen $\xrightarrow{cp}$ und $\xrightarrow{set,cp}$. □

Austauschbarkeit von $\xrightarrow{set,let,nd,*}$ und $\xrightarrow{set,cp,*}$

Lemma 2.2.8. *Für beliebige drei Mengen von* Λ_{let}*-Ausdrücken* M_1, M_2, M_3 *mit* $M_1 \xrightarrow{set,let,nd} M_2$ *und* $M_1 \xrightarrow{set,cp} M_3$ *existiert eine Menge von Ausdrücken* M_4*, so daß* $M_2 \xrightarrow{set,cp,\leq 1} M_4$ *und* $M_3 \xrightarrow{set,let,nd,\leq 2} M_4$ *gilt.*

Beweis. Sofern die Redizes nicht überlappen ist die Korrektheit der Hypothese offensichtlich. Ist eine Reduktion eine *set*, *nd*-Reduktion so entsteht kein Problem die beiden erhaltenen Ausdrücke in einem Schritt zu einem gemeinsamen Nachfahren zu reduzieren, da die *set*, *cp*-Reduktion den *nd*-Redex nicht vervielfachen kann. Ist eine Reduktion eine *set*, *let*-Reduktion, so kann der Redex der *set*, *let*-Reduktion durch die *set*, *cp*-Reduktion vervielfacht werden. In diesem Fall benötigen wir zwei *set*, *let*-Reduktion und eine *set*, *cp*-Reduktion um zu einem gemeinsamen Nachfahren zu reduzieren. □

Lemma 2.2.9. *Die Reduktionen* $\xrightarrow{set,let,nd,*}$ *und* $\xrightarrow{set,cp,*}$ *sind austauschbar.*

Beweis. Wir benutzen Lemma 3.3.6 in [Bar84], das für zwei beliebige Reduktionen $\rightarrow_1$ und $\rightarrow_2$ folgendes aussagt:
Wenn für alle Elemente a, b, c mit $a \rightarrow_1 b$ und $a \rightarrow_2 c$ ein Element d existiert, so daß $b \xrightarrow{*}_2 d$ und $c \xrightarrow{0/1}_1 d$, dann sind $\xrightarrow{*}_1$ und $\xrightarrow{*}_2$ austauschbar.

Indem wir bei diesem Lemma $\rightarrow_1 := \xrightarrow{set,cp}$ und $\rightarrow_2 := \xrightarrow{set,let,nd}$ wählen, wird gezeigt, daß die beiden Reduktionen $\xrightarrow{set,let,nd,*}$ und $\xrightarrow{set,cp,*}$ austauschbar sind. □

Satz 2.2.3. *Die Reduktion* $\xrightarrow{set,\Lambda_{let},*}$ *ist konfluent.*

Beweis. Jede *set*, Λ_{let}, $*$-Reduktion läßt sich in Abschnitte zerteilen, die nur aus *set*, *let*, *nd*- oder *set*, *cp*-Reduktionen bestehen. Daraus folgt in Kombination mit den Sätzen 2.2.1, 2.2.2 sowie Lemma 2.2.9 und einer einfachen induktiven Überlegung die Hypothese. □

Nachdem wir gezeigt haben, daß die Reduktion des Λ_{let}-Kalkül konfluent im Bezug auf Mengen von Λ_{let}-Ausdrücken ist, werden wir nun zu einem anderen Betrachtungsgegenstand übergehen.
Wie bereits in der Einleitung erwähnt können verschiedene Reduktionsstrategien (Auswertungsstrategien), trotz Konfluenz, zu einem unterschiedlichen Terminierungsverhalten führen. Reduktionsstrategie bedeutet dabei, daß für einen gegebenen Ausdruck aus der Menge aller seiner Redizes einer herausgestellt wird oder der ganze Ausdruck irgendeine Normalform besitzt.
Wir werden in dem nun folgenden Abschnitt eine Reduktionsstrategie für den Λ_{let}-Kalkül vorstellen, die als normal-order Reduktion bezeichnet wird und in der gesamten weiteren Arbeit von zentraler Bedeutung sein wird.

2.3 normal-order Reduktion

Wir definieren nun die Begriffe normal-order Reduktion, normal-order Redex sowie den Begriff der schwachen Kopfnormalform. Im Anschluß werden wir zeigen, daß durch wiederholte Reduktion des normal-order Redex eine schwache Kopfnormalform erreicht wird, sofern diese existiert.

Definition 2.3.1. *Der normal-order Redex eines Λ_{let}-Ausdrucks t wird unter Zuhilfenahme einer Menge von Regeln definiert, die ein Label E (E für Evaluation) bis zu einem finalen Subausdruck in t auf und ab bewegen. Wir beginnen mit t^E, wobei t unmarkiert ist:*

i) $C[s^E]$ *und s ist eine Abstraktion oder die Konstante* `choice` *. Wir stoppen, der Ausdruck t ist in schwacher Kopfnormalform.*

ii) $C[(rs)^E]$ *und r ist eine Abstraktion. Wir stoppen, der markierte lbeta-Redex ist der normal-order Redex.*

iii) $C[(\texttt{choice}\ s)^E]$. *Wir stoppen, der markierte nd-Redex ist der normal-order Redex.*

iv) $C[(rs)^E]$ *und r ist eine Variable oder eine Applikation. Wir fahren fort mit* $C[(r^E s)]$.

v) $C[(rs)^E]$ *und r ist ein Let-Ausdruck. Wir stoppen, der markierte lapp-Redex ist der normal-order Redex.*

vi) $C[(\texttt{let}\ x = s\ \texttt{in}\ t)^E]$. *Wir fahren fort mit* $C[(\texttt{let}\ x = s\ \texttt{in}\ t^E)]$.

vii) $C[(\texttt{let}\ x = (\texttt{let}\ y = t_y\ \texttt{in}\ t_x)\ \texttt{in}\ D[x^E])]$. *Wir stoppen, der innerhalb des C-Kontext vorkommende llet-Redex ist der normal-order Redex.*

viii) $C[(\texttt{let}\ x = r\ \texttt{in}\ D[x^E])]$ *und* r *ist eine Abstraktion oder die Konstante* `choice` . *Wir stoppen, der vom Kontext eingeschlossene cp-Redex ist der normal-order Redex, mit der markierten Variable* x^E *als Kopierziel.*

ix) $C[\texttt{let}\ x = r\ \texttt{in}\ D[x^E]]$ *und* r *ist eine Applikation oder eine Variable. Wir fahren fort mit* $C[\texttt{let}\ x = r^E\ \texttt{in}\ D[x]]$.

x) $C[x^E]$ *und* x *ist eine* ***freie*** *Variable. Wir stoppen, der Ausdruck* t *ist in schwacher Variablen-Kopfnormalform.*

Eine normal-order Reduktion erfolgt, indem der als normal-order Redex markierte Subausdruck reduziert wird. Der Begriff normal-order Reduktion kann transitiv erweitert benutzt werden, d. h. auch eine Folge von Reduktionen, bei der stets der normal-order Redex reduziert wird, bezeichnen wir als normal-order Reduktion.

Bei geschlossenen Ausdrücken ist das Erreichen einer schwachen Variablen-Kopfnormalform wegen des Fehlens freier Variablen nicht möglich.

Konvention 2.3.1. *"normal-order" wird durch "no" abgekürzt. "schwache Kopfnormalform" wird durch "WHNF" abgekürzt, wobei WHNF eine abgekürzte Schreibweise von "weak head normal form" darstellt, des englischen Äquivalent des Begriffs "schwache Kopfnormalform". Entsprechend kürzen wir "schwache Variablen-Kopfnormalform" durch "VWHNF" ab.*

Beispiel 2.3.1. Der Ausdruck $((\texttt{let}\ x = (\texttt{let}\ y = t_y\ \texttt{in}\ t_x)\ \texttt{in}\ x)s)$ besitzt einen *llet-* wie einen *lapp*-Redex. Bei Anwendung des no-Markierungsalgorithmus wird der *lapp*-Redex als no-Redex markiert.

An dieser Stelle sei erwähnt, daß die in [AFM$^+$95] vorgestellt Verfahrensweise zur Markierung des no-Redex bei dem obigen Beispiel den *llet*-Redex als no-Redex wählen würde. Eine Erklärung für diese Vorgehensweise ist, daß die *llet*-Reduktion sowieso erforderlich wird, um die direkte Bindung des Ausdrucks t_x zu erhalten, und deshalb auch vorgezogen werden kann. In Kapitel 7, dessen Gegenstand die Darstellung der benutzen maschinellen Beweishilfsmittel ist, wird dargelegt, daß diese veränderte Wahl jedoch durchaus zu einer Verlängerung der no-Reduktion führen kann.

Definition 2.3.2. *Gegeben ein Ausdruck s. Jeden Redex in s, der keinen no-Redex bildet, bezeichnen wir als* internen Redex. *Entsprechend bezeichnen wir die Reduktion eines internen Redex als* interne Reduktion. *Der Begriff interne-Reduktion kann wie der Begriff no-Reduktion auch transitiv verwendet werden.*

Notation 2.3.1. *Wir schreiben:*

$s \xrightarrow{no} t$, *wenn* s *durch eine no-Reduktion zu* t *reduziert wird.*

$s \xrightarrow{i} t$, *wenn* s *durch eine interne Reduktion zu* t *reduziert wird.*

$s \xrightarrow{i,llet,*} t$ bezeichnet somit beispielsweise, daß der Ausdruck s durch eine Folge interner *llet*-Reduktionen in den Ausdruck t überführt werden kann. Die Folge kann auch leer sein, d.h $s \equiv t$ gelten.

Es ist nicht offensichtlich, ob der oben spezifizierte no-Markierungsalgorithmus immer terminiert. Dies soll nun gezeigt werden.

Definition 2.3.3. *Mittels der Funktion Ecode ordnen wir wie folgt jedem* Λ_{let}-*Ausdruck mit genau einem E-Label ein Wort aus der Sprache* $\{0,1,2\}^*$ *zu:*

$$Ecode(st) := \begin{cases} 1.Ecode(s) & \text{falls } (st)^E \\ 0.Ecode(s) & \text{sonst} \end{cases}$$

$$Ecode(\texttt{let } x = s \texttt{ in } t) := \begin{cases} 2.w & \text{falls } (\texttt{let } x = s \texttt{ in } t)^E \\ 1.w & \text{falls das } E - \text{Label in } t \\ 0.w & \text{sonst} \end{cases}$$
$$\textit{wobei } w := Ecode(s).Ecode(t)$$

$$Ecode(s) := \varepsilon \textit{ falls } s \textit{ keine Applikation oder Let-Ausdruck}$$

Dabei entspricht '.' dem Konkatenations-Operator und 'ε*' dem leeren Wort.*

Eine einfache Überlegung zeigt, daß bei jedem Ausdruck s die Länge des Wortes $Ecode(s)$ genau der Summe der Anzahl von Applikationen und Let-Ausdrücken entspricht, die sich nicht innerhalb einer Abstraktion befinden.

Lemma 2.3.1. *Der no-Markierungsalgorithmus angewendet auf einen Ausdruck t stoppt immer und markiert entweder einen eindeutigen no-Redex oder den Gesamtausdruck als WHNF oder VWHNF.*

Beweis. Wir benutzen die Funktion *Ecode*, um jedem Ausdruck ein Wort zuzuweisen, und ordnen die erhaltenen Worte lexikographisch. Wir zeigen daß jede der drei rekursiven Regeln *iv), vi)* oder *ix)* des no-Markierungsalgorithmus zu einem kleineren Wort führt:

iv)
$$\begin{array}{ll} & Ecode(C[(st)^E]) = w_1 1 w \\ > & Ecode(C[(s^E t)]) = w_1 0 w' \end{array}$$

vi)
$$\begin{array}{ll} & Ecode(C[(\texttt{let } x = s \texttt{ in } t)^E]) = w_1 2 w \\ > & Ecode(C[\texttt{let } x = s \texttt{ in } t^E]) = w_1 1 w' \end{array}$$

ix) $\quad Ecode(C[\texttt{let}\ x = s\ \texttt{in}\ D[x^E]]) = w_1 1 w$
$\quad > \quad Ecode(C[\texttt{let}\ x = s^E\ \texttt{in}\ D[x]]) = w_1 0 w'$

Dabei kann $w > w'$ gelten, jedoch haben in allen drei Fällen die Worte w und w' dieselbe Länge.

Da der no-Markierungsalgorithmus für alle möglichen Formen eines gelabelten Subausdrucks genau eine Regel formuliert und alle rekursiven Regeln zu einer kleineren Bewertung führen, trifft der no-Markierungsalgorithmus nach endlicher Zeit für alle Ausdrücke eine eindeutige Entscheidung. □

2.3.1 Eigenschaften von no-Redizes, WHNF, VWHNF

Wir werden in dem nun folgenden Abschnitt einige Eigenschaften von no-Redizes sowie WHNF und VWHNF zeigen, die wir in späteren Beweisen benötigen werden. So wird beispielsweise von Interesse sein, welche Form die Kontexte haben, in denen sich die verschiedenen Arten von no-Redizes befinden bzw. wie Ausdrücke in WHNF oder VWHNF aussehen.

Definition 2.3.4. *Wir unterscheiden folgende 11 Zustände des no-Markierungsalgorithmus:*

q_1: *Das E-Label ist an einen let-Ausdruck geheftet.*

q_2: *Das E-Label ist an eine Applikation geheftet.*

q_3: *Das E-Label ist an eine Variable geheftet.*

q_4: *Das E-Label ist an einer Abstraktion oder der Konstante* `choice` *geheftet.*

q_W: *Eine Markierung als WHNF ist erfolgt.*

q_{VW}: *Eine Markierung als VWHNF ist erfolgt.*

q_{tp}: *Die Markierung eines no-Redex vom Typ tp ist erfolgt. Ist der no-Redex beispielsweise vom Typ lbeta, so wird für tp lbeta eingesetzt und wir erhalten* q_{lbeta}.

Lemma 2.3.2. *Geht der no-Markierungsalgorithmus durch eine Regelanwendung von einem Zustand q in einen Zustand q′ über, so existiert in dem Graphen* $\mathcal{G}_{no}$ *aus Abbildung 2.1 eine mit der Nummer der angewendeten Regel beschriftete Kante von dem Knoten q zu dem Knoten q′.*

Beweis. Ein einfacher Abgleich zwischen dem no-Markierungsalgorithmus und dem Graphen $\mathcal{G}_{no}$ für alle Zustände. □

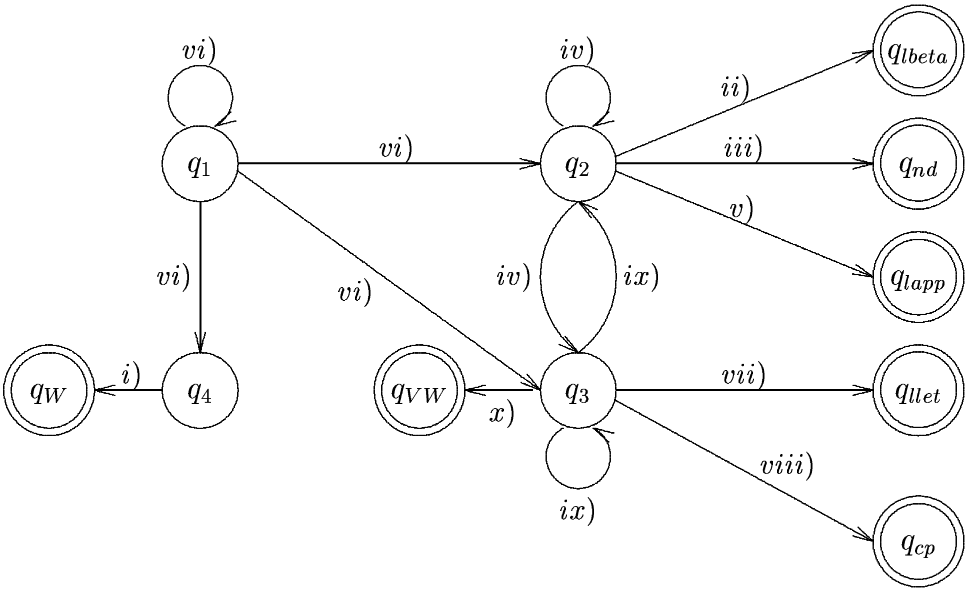


Abbildung 2.1: Der Graph $\mathcal{G}_{no}$.

Lemma 2.3.3. *Der no-Markierungsalgorithmus werde auf einen beliebigen Ausdruck t (offen oder geschlossen) angewendet. Bei allen dabei erreichten Zuständen wird die untenstehende Zuordnung zwischen Zustand und Kontext, in dem sich der gelabelte Subausdruck befindet, eingehalten:*

Zustand	$t \equiv Kontext[t_s^E]$
q_1	$L_R^*[(\texttt{let}\ x = t_x\ \texttt{in}\ t)^E]$
q_2, q_{lbeta}, q_{lapp}, q_{nd}	$W[(rs)^E]$
q_3, q_{VW}	$W[x^E]$
q_4, q_W	$L_R^*[r^E]$ *wobei* r *eine Abstraktion oder die Konstante* `choice`
q_{cp}, q_{llet}	$L_R^*[\texttt{let}\ x = t_x\ \texttt{in}\ W[x^E]]$

Beweis. Sei $\mathcal{F}$ die Folge von Zuständen, die bei der Anwendung des no-Markierungsalgorithmus auf t durchwandert werden. Der Beweis erfolgt durch Induktion über alle der Länge nach aufsteigend geordneten Präfixfolgen von $\mathcal{F}$. Die Induktionshypothese lautet, daß die Zuordnung zwischen Zuständen und Zerlegungsmustern für alle Zustände die im Rahmen einer Zustandsfolge erreicht werden korrekt ist.

Induktionsanfang:
Nach dem Anbringen des E-Label in t sind die vier Zustände q_1, q_2, q_3, q_4 möglich. Die den Zuständen q_1, q_2, q_3, q_4 zugeordneten Kontextklassen enthalten alle den Kontext $[\cdot]$.

Induktionsschritt:

Es ist zu zeigen, daß die Induktionshypothese durch keinen Zustandsübergang des no-Markierungsalgorithmus verletzt wird. Die verschiedenen Möglichkeiten von Zustandsübergängen gibt der Graph $\mathcal{G}_{no}$ wieder. Wir betrachten stellvertretend eine Auswahl der 15 bestehenden Möglichkeiten:

$q_1 \mapsto q_1$, $(q_1 \mapsto q_2)$, $(q_1 \mapsto q_3)$, $(q_1 \mapsto q_4)$:

$$\begin{array}{ll}
 & L_R^*[(\texttt{let } x = t_x \texttt{ in let } y = t_y \texttt{ in } s)^E] \\
\mapsto & L_R^*[\texttt{let } x = t_x \texttt{ in } (\texttt{let } y = t_y \texttt{ in } s)^E] \\
\equiv & L_R^*[L_R[(\texttt{let } y = t_y \texttt{ in } s)^E]] \\
\equiv & L_R^*[(\texttt{let } y = t_y \texttt{ in } s)^E]
\end{array}$$

$q_3 \mapsto q_2$, $(q_3 \mapsto q_3)$: Wir betrachten den schwierigeren Fall $W[x^E] \equiv L_R^*[\texttt{let } y = A_L^*[x^E] \texttt{ in } W[y]]$

$$\begin{array}{ll}
 & L_R^*[\texttt{let } y = A_L^*[x^E] \texttt{ in } W[y]]] \\
\equiv & L_R^*[\texttt{let } x = (st) \texttt{ in } L_R^*[\texttt{let } y = A_L^*[x^E] \texttt{ in } W[y]]]] \\
\mapsto & L_R^*[\texttt{let } x = (st)^E \texttt{ in } L_R^*[\texttt{let } y = [A_L^*[x]] \texttt{ in } W[y]]]] \\
\equiv & L_R^*[\texttt{let } x = (st)^E \texttt{ in } W[x]
\end{array}$$

$q_2 \mapsto q_3$, $(q_2 \mapsto q_2)$:

$$\begin{array}{ll}
 & W[(xs)^E] \\
\mapsto & W[(x^E s)] \\
\equiv & W[A_L[x^E]] \\
\equiv & W[x^E]
\end{array}$$

$q_3 \mapsto q_{llet}$, $(q_3 \mapsto q_{cp})$: Wir betrachten stellvertretend $W[x^E] \equiv L_R^*[\texttt{let } y = A_L^*[x^E] \texttt{ in } W[y]]$

$$\begin{array}{ll}
 & L_R^*[\texttt{let } y = A_L^*[x^E] \texttt{ in } W[y]] \\
\equiv & L_R^*[\texttt{let } x = \texttt{let } z = t_z \texttt{ in } L_R^*[\texttt{let } y = A_L^*[x^E] \texttt{ in } W[y]]] \\
\equiv & L_R^*[\texttt{let } x = \texttt{let } z = t_z \texttt{ in } W[x^E]]
\end{array}$$

Die in Klammern gesetzten Übergänge können durch einfache Variation aus dem jeweils gezeigten Fall hergeleitet werden. Die Korrektheit für die Übergänge $q_2 \mapsto q_{lbeta}$, $q_2 \mapsto q_{nd}$, $q_2 \mapsto q_{lapp}$, $q_3 \mapsto q_{VW}$, $q_4 \mapsto q_W$ ist offensichtlich. $\square$

Die im letzten Lemma bewiesenen Zerlegungs- und damit Kontextaussagen lassen die folgenden Schlußfolgerungen zu:

Korollar 2.3.1. *Folgende Aussagen sind korrekt:*

- *Ein no-Redex vom Typ lbeta, nd, lapp befindet sich immer in einem W-Kontext. Ein no-Redex vom Typ llet oder cp befindet sich immer in einem L_R^*-Kontext. Das Kopierziel eines (no, cp)-Redex befindet sich stets in einem W-Kontext.*

- *Jeder Ausdruck in WHNF hat die Form $L_R^*[r]$ mit r eine Abstraktion oder die Konstante* `choice` *. Jeder Ausdruck in VWHNF hat die Form $W[x]$ mit einer freien Variable x.*
- *Ein no-Redex kann nur Subausdruck eines anderen Redex vom Typ cp oder llet sein, aber nicht Teil des zu kopierenden Ausdrucks bei einem Redex vom Typ cp. Ein no-Redex kann ferner kein Subausdruck eines Redex vom Typ lbeta, nd oder lapp sein.*

Der folgende Satz wird in vielen späteren Beweisen benutzt, da er einen direkten Bezug zwischen der Struktur eines Ausdrucks und dem Typ des no-Redex bzw. einer WHNF oder einer VWHNF formuliert.

Satz 2.3.1. *Gegeben sei ein Ausdruck t. r sei bei den unten aufgeführten gdw.-Beziehungen stets eine Abstraktion oder die Konstante* `choice` *. Folgende Aussagen sind korrekt:*

$$\begin{aligned}
& t \equiv L_R^*[r] && \iff && \textit{t befindet sich in WHNF} \\
& t \equiv W[x] \textit{ und } x \in V_{free}(t) && \iff && \textit{t befindet sich in VWHNF} \\
& t \equiv L_R^*[\texttt{let } x = \texttt{let } y = t_y \texttt{ in } t_x \texttt{ in } W[x]] && \iff && \textit{t besitzt einen } (no, llet)\textit{-Redex} \\
& t \equiv W[(\texttt{let } x = t_x \texttt{ in } s)t] && \iff && \textit{t besitzt einen } (no, lapp)\textit{-Redex} \\
& t \equiv W[(\lambda x.s)t] && \iff && \textit{t besitzt einen } (no, lbeta)\textit{-Redex} \\
& t \equiv L_R^*[\texttt{let } x = r \texttt{ in } W[x]] && \iff && \textit{t besitzt einen } (no, cp)\textit{-Redex} \\
& t \equiv W[\texttt{choice } s] && \iff && \textit{t besitzt einen } (no, nd)\textit{-Redex}
\end{aligned}$$

Beweis. Alle "$\Leftarrow$"-Richtungen werden bereits durch Lemma 2.3.3 bzw. Korollar 2.3.1 bewiesen. Die andere Richtung kann in allen Fällen durch eine einfache Anwendung des no-Markierungsalgorithmus gezeigt werden:

Der no-Markierungsalgorithmus angewendet auf einen Ausdruck der Form $L_R^*[r]$ mit r eine Abstraktion oder die Konstante `choice` führt immer zu Regel *i)* , was eine einfache Induktion über die Tiefe des L_R^*-Kontext sofort zeigt.

Der no-Markierungsalgorithmus angewendet auf einen beliebigen Ausdruck der Form $W[s]$ mit s einer **Variablen oder Applikation** führt stets zu $W[s^E]$ ohne zuvor einen no-Redex markieren zu können, wie sich durch strukturelle Induktion über den Aufbau eines W-Kontext schnell zeigen läßt:

- $W \equiv (L_R^*[A_L^*[s]])^E$

Der no-Markierungsalgorithmus durchläuft zunächst den L_R^*-Kontext mittels Regel *vi)* und anschließend den A_L^*-Kontext mittels Regel *iv)*, somit wird $(L_R^*[A_L^*[s^E]])$ erhalten.

- $W \equiv (L_R^*[\mathtt{let}\ x = A_L^*[s]\ \mathtt{in}\ W[x]])^E$
  Der no-Markierungsalgorithmus durchläuft zunächst mittels Regel *vi)* den L_R^*-Kontext zu $L_R^*[\mathtt{let}\ x = A_L^*[s]\ \mathtt{in}\ (W[x])^E]$. Gemäß Induktionsvoraussetzung wird im weiteren $L_R^*[\mathtt{let}\ x = A_L^*[s]\ \mathtt{in}\ (W[x^E])]$ erhalten. Da s eine Variable oder eine Applikation ist, gilt dies auch für $A_L^*[s]$ und es wird mittels Regel *ix)* zu $L_R^*[\mathtt{let}\ x = (A_L^*[s])^E\ \mathtt{in}\ (W[x])]$ übergegangen. Final wird mittels Regel *iv)* der A_L^*-Kontext durchlaufen und $L_R^*[\mathtt{let}\ x = A_L^*[s^E]\ \mathtt{in}\ (W[x])]$ erhalten.

Aus dem oben gezeigten folgt die Hypothese direkt für einen $(no, lbeta)$-Redex (Regel *ii)*), einen (no, nd)-Redex (Regel *iii)*), sowie einen $(no, lapp)$-Redex (Regel *v)*). Ferner folgt wegen Regel *x)* die Aussage bezüglich einer VWHNF. (Es ist zu beachten, daß obiges Teillemma nicht für s eine Abstraktion, die Konstante `choice` sowie einen let-Ausdruck gilt.)

Bleibt der Fall eines $(no, llet)$- sowie eines (no, cp)-Redex:
Der no- Markierungsalgorithmus angewendet auf $(L_R^*[\mathtt{let}\ x = \mathtt{let}\ y = t_y\ \mathtt{in}\ t_x\ \mathtt{in}\ W[x]])^E$ durchläuft mittels Regel *vi)* zunächst den L_R^*-Kontext und es wird $L_R^*[\mathtt{let}\ x = \mathtt{let}\ y = t_y\ \mathtt{in}\ t_x\ \mathtt{in}\ W[x]^E]$ erhalten. Gemäß des obigen Teillemma wird bei weiterer Anwendung des Algorithmus $L_R^*[\mathtt{let}\ x = \mathtt{let}\ y = t_y\ \mathtt{in}\ t_x\ \mathtt{in}\ W[x^E]]$ erreicht. Im Anschluß erfolgt mittels Regel *vii)* die Markierung eines $(no, llet)$-Redex. Genau dasselbe Schema gilt bei $L_R^*[\mathtt{let}\ x = r\ \mathtt{in}\ W[x]]$, nur mit dem Unterschied, daß in diesem Fall mittels Regel *viii)* ein (no, cp)-Redex markiert wird. □

Das Beispiel $L_R^*[A_L^n[\lambda x.s]] \equiv L_R^*[A_L^{n-1}[(\lambda x.s)t]]$ zeigt, daß sich bei einem gegebenen Ausdruck durchaus mehrere Subausdrücke in einem W-Kontext befinden können, ohne daß dies jedoch im Widerspruch zu dem zuletzt gezeigten Lemma steht.

Lemma 2.3.4. *Sei t ein Ausdruck in WHNF. Wenn $t \xrightarrow{a} t'$ mit $a \in \{llet, lapp, lbeta, nd, cp\}$, dann ist t' in WHNF.*

Beweis. Aus t in WHNF folgt, daß t die Form $L_R^*[r]$ besitzt, mit r eine Abstraktion oder die Konstante `choice` . Durch die Reduktion eines beliebigen Redex vom Typ *llet*, *lapp*, *lbeta*, *nd* oder *cp* in t entsteht ein Ausdruck t', der wie t die Form $L_R^*[r']$, mit r' eine Abstraktion oder die Konstante `choice` , besitzt, was sich durch Überprüfen der zwei möglichen Fälle eines Redex in L_R^* oder r leicht zeigen läßt. Gemäß Lemma 2.3.1 ist t' damit auch in WHNF. □

Lemma 2.3.5. *t sei ein geschlossener Ausdruck , der sich nicht WHNF befindet. Wenn $t \xrightarrow{i,a} t'$ mit $a \in \{llet, lapp, lbeta, nd, cp\}$, dann ist auch t' nicht in WHNF.*

Beweis. Aus Lemma 2.3.3 ist für die verschiedenen Redexarten bekannt, in welchen Kontextarten sie als no-Redex vorkommen können. Es ist für alle Redextypen *llet*, *lapp*, *lbeta*, *cp*, *nd* zu zeigen, daß die Reduzierung eines beliebigen internen Redex, einen Ausdruck nicht in die Form $L_R^*[r]$ mit r eine Abstraktion oder die Konstante `choice` , überführen kann, da er dann gemäß Lemma 2.3.1 in WHNF wäre.

Wir analysieren stellvertretend den Fall, daß ein Ausdruck t einen no-Redex vom Typ *lapp* besitzt.
In diesem Fall existiert eine Zerlegung $t \equiv W[((\texttt{let}\ x = t_x\ \texttt{in}\ t_s)s)^E]$. Wir betrachten alle Positionen, an denen die Reduktion eines Redex, der nicht no-Redex ist, möglich ist:

- Eine Reduktion innerhalb der Subausdrücke des *lapp*-Redex kann diesen nicht auflösen, strukturell bleibt die Applikation erhalten.
- Eine Reduktion innerhalb des W-Kontext kann nicht zu einem Ausdruck der Form $L_R^*[r]$ mit r eine Abstraktion oder die Konstante `choice` führen, wie eine einfache Betrachtung der Struktur eines W-Kontexts zeigt.

□

Das folgende Korollar folgt mittels einer einfachen Widerspruchsüberlegung aus dem obigen Lemma.

Korollar 2.3.2. *t sei ein geschlossener Ausdruck. Wenn $t \xrightarrow{i,a} t'$ mit $a \in \{llet, lapp, lbeta, nd, cp\}$ dann, folgt aus t' in WHNF, daß sich auch t in WHNF befindet.*

Lemma 2.3.6. *t sei ein geschlossener Ausdruck, der sich nicht in WHNF befindet. Wenn $t \xrightarrow{no,a} t'$ mit $a \in \{llet, lapp, nd\}$, dann befindet sich auch t' nicht in WHNF.*

Beweis. Dies zeigt eine einfache Betrachtung der durch die no-Reduktion erhaltenen Ausdrücke bei den drei Redexarten:

(*lapp*) $W[\texttt{let}\ x = t_x\ \texttt{in}\ (st)]$
(*nd*) $W[(\lambda x.\lambda y.x)s]$, $W[(\lambda x.\lambda y.y)s]$
(*llet*) $L_R^*[\texttt{let}\ x = t_x\ \texttt{in}\ \texttt{let}\ y = t_y\ \texttt{in}\ W[x]] \equiv W[x]$

Keiner der obigen Ausdrücke kann gleichzeitig die Form $L_R^*[r]$ mit r eine Abstraktion oder die Konstante `choice` besitzen. □

2.4 no-Reduktionen und WHNF

Eine Fragestellung, die sich nun ergibt lautet: "Haben no-Reduktionen und WHNF mehr miteinander zu tun, als ihr gemeinsames Vorkommen im no-Markierungsalgorithmus". Eine allgemein mit dem Begriff no-Reduktion assoziierte Eigenschaft ist, daß die no-Reduktion zu einem Ausdruck in WHNF führt, wenn irgendeine Reduktion zu einem Ausdruck in WHNF existiert. Wir werden nun zeigen, daß dieses im Falle der no-Reduktion des Λ_{let}-Kalküls auch der Fall ist. Für den klassischen Lambda-Kalkül ist vergleichbares bekannt, wenn als no-Redex stets der am weitesten links stehende Redex benutzt wird.
Nun ist die no-Reduktion des Λ_{let}-Kalküls wegen des Nichtdeterminismus nicht eindeutig, was ein gewisse Präzisierung der Fragestellung erforderlich macht. Wir werden hier zeigen, daß ausgehend von einem geschlossenen Ausdruck t eine no-Reduktion zu einem Ausdruck t' in WHNF existiert, wenn ausgehend von t irgendeine Reduktion zu einem Ausdruck in WHNF existiert.

2.4.1 Reduktionsdiagramme, Gabeldiagramme und Vertauschungsdiagramme

Ein bekanntes Beweishilfsmittel bei der Untersuchung von Reduktionen besteht in der Verwendung von Diagrammen, die Reduktionszusammenhänge beschreiben. Beispiele für Publikationen, die einen Eindruck von der Mächtigkeit dieses Beweiswerkzeuges geben, sind [BvOK98], [DC96] und [vO94]. Für die diagrammartige Darstellung von Reduktionszusammenhängen benutzen wir hier den Begriff Reduktionsdiagramm, den wie folgt formal definieren:

Definition 2.4.1. *Ein Reduktionsdiagramm ist ein gerichteter Graph, dessen Knoten Ausdrücke sind und dessen Kanten Reduktionen entsprechen. Dabei werden die zwei folgenden Kantenformen unterschieden:*

$\longrightarrow$	*Ein durchgängiger Pfeil bezeichnet eine gegebene Reduktion.*
$\dashrightarrow$	*Ein gestrichelter Pfeil bezeichnet eine Existenz-quantifizierte Reduktion.*

An eine Kante können zusätzlich Informationen über die Art der Reduktion angeheftet sein.

Der Begriff des Reduktionsdiagramms soll nun anhand eines Beispiels vertiefend betrachtet werden.

Beispiel 2.4.1. Gegeben seien die drei Ausdrücke:

$$\begin{tikzcd}
t_1 \arrow[r, "{i,llet}"] \arrow[dd, "{no,lapp}"'] & t_2 \arrow[d, dashed, "{no,lapp}"] \\
 & t_4 \arrow[d, dashed, "{no,lapp}"] \\
t_3 \arrow[r, dashed, "{i,llet}"] & t_5
\end{tikzcd}$$

Ausdrücke explizit spezifiziert

$$\begin{tikzcd}
\circ \arrow[r, "{i,llet}"] \arrow[dd, "{no,lapp}"'] & \circ \arrow[d, dashed, "{no,lapp}"] \\
 & \circ \arrow[d, dashed, "{no,lapp}"] \\
\circ \arrow[r, dashed, "{i,llet}"] & \circ
\end{tikzcd}$$

Ausdrücke durch Platzhalter ersetzt

Abbildung 2.2: Reduktionsdiagramme zu Beispiel 2.4.1

$$\begin{aligned}
t_1 &\equiv (\texttt{let}\ x = (\texttt{let}\ y = s_1\ \texttt{in}\ s_2)\ \texttt{in}\ s_3)s_4 \\
t_2 &\equiv (\texttt{let}\ y = s_1\ \texttt{in}\ (\texttt{let}\ x = s_2\ \texttt{in}\ s_3))s_4 \\
t_3 &\equiv \texttt{let}\ x = (\texttt{let}\ y = s_1\ \texttt{in}\ s_2)\ \texttt{in}\ (s_3 s_4)
\end{aligned}$$

Der Ausdruck t_1 verfügt über einen *llet*- sowie *lapp*-Redex. Durch Reduktion des *llet*-Redex wird zu t_2 gelangt, durch Reduktion des *lapp*-Redex zu t_3. Gedanklich bilden die drei Ausdrücke t_1, t_2, t_3 zusammen mit den zwei Reduktionen $t_1 \xrightarrow{llet} t_2$ und $t_1 \xrightarrow{lapp} t_3$ eine "Gabel", bestehend aus zwei gegebenen Reduktionen.
Unsere Fragestellung sei nun, wie wir ausgehend von den Gabelenden t_2 und t_3 einen gemeinsamen Nachfahren erreichen können. Wenn wir ausgehend von t_2 eine *lapp*-Reduktion ausführen, können wir einen Ausdruck

$$t_4 \equiv \texttt{let}\ y = s_1\ \texttt{in}\ ((\texttt{let}\ x = s_2\ \texttt{in}\ s_3)s_4)$$

erreichen. Von t_4 wiederum erreichen wir mittels einer weiteren *lapp*-Reduktion einen Ausdruck

$$t_5 \equiv \texttt{let}\ y = s_1\ \texttt{in}\ (\texttt{let}\ x = s_2\ \texttt{in}\ (s_3 s_4)).$$

Denselben Ausdruck können wir ausgehend von t_3 mittels einer einzelnen *llet*-Reduktion erreichen. Das linke Reduktionsdiagramm aus Abbildung 2.2 stellt die beschriebenen Reduktionszusammenhänge grafisch dar. Dabei beinhaltet die Beschriftung der Kanten die zusätzliche Information, ob es sich bei der jeweiligen Reduktion um eine interne Reduktion oder no-Reduktion handelt.

Reduktionszusammenhänge, wie sie das obige Beispiel beschreibt, werden im Kontext später erscheinender Beweise von zentralem Interesse sein. Dabei werden uns jedoch nicht weiter spezifische Ausdrücke interessieren, wie dies bei obigem Beispiel der Fall ist, sondern nur das "Muster" der Reduktionszusammenhänge an sich. Aus diesem Grund führen wir folgende Notation ein:

$$\begin{tikzcd}
\circ \arrow[rr, "a"] \arrow[d, "b"] & & \circ \arrow[d, dashed, "r_1''"] \\
\circ \arrow[d, dashed, "r_1"] & & \vdots \\
\vdots \arrow[d, dashed, "r_n"] & & \vdots \arrow[d, dashed, "r_p''"] \\
\circ \arrow[r, dashed, "r_1'"] & \cdots \arrow[r, dashed, "r_m'"] & \circ
\end{tikzcd}$$

Abbildung 2.3:

Notation 2.4.1. *Das Symbol '$\circ$' entspricht, sofern es in einem Reduktionsdiagramm oder einer Reduktion vorkommt, einem Platzhalter für einen Ausdruck.*

Bei dem rechten Reduktionsdiagramm aus Abbildung 2.2 sind alle Ausdrücke aus Beispiel 2.4.1 durch Platzhalter ersetzt. Es ist offensichtlich, daß viele weitere 5-Tupel von Ausdrücken existieren, die genauso die Reduktionszusammenhänge des rechten Reduktionsdiagramms aus Abbildung 2.2 erfüllen, das rechte Diagramm ist somit eine verallgemeinerte Darstellung des linken Diagramms.

Wir werden nun eine spezielle Notation kennenlernen, die dazu geeignet ist, in kompakter Form eine bestimmte Familie von Reduktionsdiagrammen zu beschreiben, bei denen der Platzhalter '$\circ$' für Ausdrücke verwendet wird. Dies dient zur Vereinfachung späterer Beweise, um nicht ständig auf eine grafische Darstellung für Reduktionsdiagramme zurückgreifen zu müssen.

Definition 2.4.2. *Ein Gabeldiagramm G ist eine Metaregel der Gestalt*

$$\xleftarrow{r_n} \circ \cdots \xleftarrow{r_1} \circ \xleftarrow{b} \circ \xrightarrow{a} \;\rightsquigarrow\; \xrightarrow{r_1'} \circ \cdots \xrightarrow{r_m'} \circ \xleftarrow{r_p''} \cdots \circ \xleftarrow{r_1''}$$
$$\text{mit } m, n, p \geq 0$$

und entspricht einer syntaktischen Repräsentation der Reduktionszusammenhänge aus Abbildung 2.3.
Als Sprechweise vereinbaren wir, daß ein Gabeldiagramm ein Reduktionsdiagramm beschreibt. *Die Reduktionen* $\xleftarrow{b} \circ \xrightarrow{a}$ *bezeichnen wir dabei als die* Gabel *des durch das Gabeldiagramm beschriebenen Reduktionsdiagramms.*

In Erweiterung werden wir Gabeldiagramme nicht nur einzeln betrachten, sondern eine Definition treffen, die eine all-quantifizierte Aussage bezüglich einer

Menge von Gabeln mit speziellen Eigenschaften erlaubt.

Definition 2.4.3. *Gegeben eine Reduktionsvorschrift a, z.B. llet, lbeta. Ein* vollständiger Satz von Gabeldiagrammen *für die a-Reduktion ist ein Satz von Gabeldiagrammen $\mathcal{G}$, der folgende Eigenschaft erfüllt:*
Für alle Tripel von Ausdrücken (t, t', t'') mit

1. *t befindet sich nicht WHNF*

2. *$t \xrightarrow{a} t'$ und $t \xrightarrow{no} t''$*

existiert ein Gabeldiagramm $G \in \mathcal{G}$ sowie eine Ersetzung aller Platzhalter in dem durch G beschriebenen Reduktionsdiagramm R, so daß die Gabel des Reduktionsdiagramms R nach Ersetzung aller Platzhalter $t'' \xleftarrow{no} t \xrightarrow{a} t'$ entspricht.

Um die obige Definition zu verdeutlichen kehren wir zu unserem obigen Beispiel zurück. Das Reduktionsdiagramm aus Abbildung 2.2 beinhaltet die Gabel $t_3 \xleftarrow{no,lapp} t_1 \xrightarrow{i,llet} t_2$; Lemma 3.3.1 wiederum beinhaltet einen vollständigen Satz von Gabeldiagrammen für eine $(i, llet, R[\cdot])$-Reduktion[4]. Somit muß Lemma 3.3.1 ein Gabeldiagramm beinhalten, das den Fall aus Beispiel 2.4.1 abdeckt. Ein Abgleich der Gabeldiagramme aus Lemma 3.3.1 zeigt, daß dies das mit *v)* gekennzeichnete Diagramm leistet, wobei für ω der Wert 1 einzusetzen ist.
Es sei noch angemerkt, daß bei einem vollständigen Satz von Gabeldiagrammen zu einer spezifischen Gabel, bestehend aus drei Ausdrücken, durchaus mehrere Gabeldiagramme ein passendes Reduktionsdiagramm beschreiben können. Dies ist jedoch nicht weiter störend.
Bleibt die Frage nach dem Zweck von Gabeldiagrammen zu beantworten. Gabeldiagramme werden später verwendet, um Aussagen bezüglich der Übertragbarkeit von Reduktionen treffen zu können. In späteren Beweisen wird für verschiedene Ausdrucksumformungen, z.B. interne Reduktionen, zu zeigen sein, daß eine Reduktion zu WHNF bzw. eine unendliche no-Reduktion nach Anwendung der Umformung erhalten bleibt. Um dies zu zeigen können, werden wir Gabeldiagramme in Kombination mit einer induktiven Argumentation benutzen. Weitere Informationen zu diesem Thema enthält Abschnitt 3.1.1.

Neben den Gabeldiagrammen werden wir nun eine weitere Diagrammform kennenlernen. In späteren Beweisen werden wir ein Hilfsmittel benötigen, um Reduktionen "umordnen" zu können, genauer ausgedrückt bei Erhaltung von Start- und Zielausdruck nicht-no-Reduktionen und no-Reduktionen einer Reduktion anders anordnen zu können. Zu diesem Zweck führen wir Vertauschungsdiagramme ein, die wie folgt formal definiert werden:

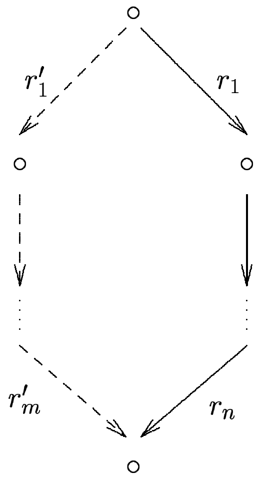


Abbildung 2.4:

Definition 2.4.4. *Ein Vertauschungsdiagramm ist eine Metaregel der Gestalt*

$$r_1 \circ \cdots \circ r_n \rightsquigarrow r'_1 \circ \cdots \circ r'_m \text{ mit } m \geq 0 \text{ und } n \geq 2$$

und entspricht einer syntaktischen Repräsentation der Reduktionszusammenhänge aus Abbildung 2.4.
Als Sprechweise vereinbaren wir wie bei den Gabeldiagrammen, daß ein Vertauschungsdiagramm ein Reduktionsdiagramm beschreibt.

Wir werden im Kontext späterer Beweise, wie auch bei den Gabeldiagrammen, nicht an einem einzelnen Diagramm interessiert sein, sondern an einem Satz von Diagrammen, der in seiner Gesamtheit eine all-quantifizierte Aussage beinhaltet.

Definition 2.4.5. *Gegeben eine Reduktionsvorschrift a, z.B. llet, lbeta. Ein* vollständiger Satz von Vertauschungsdiagrammen *für die a-Reduktion ist ein Satz von Vertauschungsdiagrammen V, der folgende Eigenschaft erfüllt:*

Für alle Tupel von Ausdrücken (t, t') mit $t \xrightarrow{a} t'$ existiert ein Vertauschungsdiagramm V sowie ein Ausdruck t'', so daß V ein Reduktionsdiagramm R beschreibt, bei dem eine Ersetzung aller Platzhalter derart möglich ist, daß die gegebene Reduktion des Reduktionsdiagramms R die Form $t \xrightarrow{a} t' \xrightarrow{no,+} t''$ hat.

Wie bereits erwähnt, werden wir Vertauschungsdiagramme dazu benutzen, um durch schrittweise Umordnung von nicht-no-Reduktionen und no-Reduktionen aus bestehenden Reduktionen neue herzuleiten, die spezielle Eigenschaften aufweisen. Der Zusatz "vollständig" gewährleistet uns dabei, daß wir keinen Fall

---

[4] Der Zusatz $R[\cdot]$ schränkt die Position des $(i, llet)$-Redex kontextuell ein. Für eine genaue Erklärung siehe Notation 3.2.1.

auslassen und unsere Betrachtungen vollständig sind. Weitere Informationen zu diesem Thema enthält Abschnitt 3.1.1.

Ein vollständiger Satz von Gabel- oder Vertauschungsdiagrammen für eine Reduktion a, ist somit ein Satz von Schablonen für Reduktionsdiagramme, die in ihrer Gesamtheit jeweils eine genaue Aussage darüber treffen, welche lokalen Konfluenzeigenschaften (Gabeldiagramme) die Reduktion a aufweist, bzw. wie a-Reduktionen über eine oder mehrere direkt anschließende no-Reduktionen (Vertauschungsdiagramme) bei Erhaltung von Start und Ziel hinwegbewegt werden können.
Abschließend treffen wir noch folgende Vereinbarung, die das Vorkommen von *nd*-Reduktionen in Gabel- bzw. Vertauschungsdiagrammen betrifft und zum Zwecke der einfacheren Spezifikation eingeführt wird.

Konvention 2.4.1. *Wird beim Vorkommen von nd-Reduktionen auf die zusätzliche Spezifikation von left oder right verzichtet, so bedeutet dies, daß ein Vertauschungs- bzw. Gabeldiagramm als zwei Einzeldiagramme aufzufassen ist, bei denen alle nd-Reduktionen jeweils die gleiche Richtung (left oder right) besitzen.*

2.4.2 Beziehung zwischen Vertauschungs- und Gabeldiagrammen

Zwischen Vertauschungsdiagrammen und Gabeldiagrammen besteht ein hohes Maß an Verwandtschaft. Das unten stehenden Reduktionsdiagramm soll diese Verwandtschaft verdeutlichen:

$$\begin{tikzcd}
\circ \arrow[r, shift left, "a"] \arrow[r, shift right] \arrow[d, dashed, shift right, "b"'] \arrow[d, shift left] & \circ \arrow[d, dashed, shift right] \arrow[d, shift left, "b"] \\
\circ \arrow[r, dashed, shift left] \arrow[r, dashed, shift right, "a"'] & \circ
\end{tikzcd}$$

Auf der Ebene der äußeren Pfeile korrespondiert obiges Reduktionsdiagramm mit einem Vertauschungsdiagramm, auf der Ebene der inneren Pfeile hingegen mit einem Gabeldiagramm. Durch die Sichtweise als Reduktionsdiagramm kann somit offensichtlich, abgesehen von den unten aufgeführten Ausnahmen, auf einfachem Wege aus einem Vertauschungsdiagramm ein Gabeldiagramm hergeleitet werden und umgekehrt.

In folgenden Ausnahmefällen ist eine wechselseitige Herleitung jedoch ausgeschlossen:

- Bei “degenerierten” Diagrammen kann eine wechselseitige Interpretation durch das Fehlen von Reduktionen an erforderlichen Stellen ausgeschlossen sein. Einen solchen Sonderfall beinhalteten beispielsweise die Gabeldiagramme der *ldel*-Reduktion, die in Lemma 3.4.1 vorgestellt werden. Bei den mit *iii)* und *iv)* gekennzeichneten Regel fehlt auf der rechten Seite eine no-Reduktion, was eine Interpretation des Gabeldiagramms als Vertauschungsdiagramm ausschließt.

- Beim Vorkommen von *nd*-Reduktionen ist bei der Herleitung von Gabeldiagrammen aus Vertauschungsdiagrammen Vorsicht angebracht. Die Ursache hierfür ist, daß Vertauschungsdiagramme einen sequentiellen Zusammenhang beschreiben, Gabeldiagramme hingegen einen parallelen Zusammenhang. Bei einem Gabeldiagramm können die beiden Zweige der Gabel die *left*- und *right*-Reduktion eines *nd*-Redex sein:

$$\begin{tikzcd}
C[\texttt{choice}\ s] \arrow[r, "nd{,}left"] \arrow[d, "nd{,}right"] & C[(\lambda x.\lambda y.x)s] \\
C[(\lambda x.\lambda y.y)s] &
\end{tikzcd}$$

  Die beiden Ausdrücke $C[(\lambda x.\lambda y.y)s]$ und $C[(\lambda x.\lambda y.x)s]$ können im allgemeinen wegen der fehlenden Konfluenz auf keinen gemeinsamen Nachfahren reduziert werden. Bei Vertauschungsdiagrammen ist diese “Anomalie” jedoch nicht möglich, da die Reduktionen hintereinander angeordnet sind. Aus diesem Grund lassen sich später für (i, nd)-Reduktionen Vertauschungdiagramme angeben jedoch wäre eine Herleitung von Gabeldiagrammen aus diesen Vertauschungsdiagrammen unzulässig.

2.4.3 Struktur des WHNF-Beweises

Wie bereits aufgeführt ist das Ziel dieses Abschnitts zu zeigen, daß eine no-Reduktion zu einer WHNF führt, sofern diese existiert. Gegenüber deterministischen Kalkülen wie dem klassischen Lambda-Kalkül besteht beim Λ_{let}-Kalkül die zusätzliche Schwierigkeit, daß die no-Reduktion nicht eindeutig sein muß, d.h. es können mehrere no-Reduktionen von einem gegebenen Ausdruck ausgehen. Es ist daher zunächst nicht offensichtlich welche Beweismethodik geeignet ist, die Hypothese zu zeigen. Folgende Überlegungen weisen den Weg zu der hier benutzen Methodik:

- Voraussetzung für die Existenz einer no-Reduktion zu WHNF ist, daß irgendeine beliebige Reduktion zu WHNF existiert. Existiert eine solche Reduktion nicht, kann auch keine no-Reduktion zu WHNF existieren.

- Wird eine Reduktion zu WHNF als Folge einzelner Reduktionsschritte betrachtet, so ist jeder dieser Einzelschritte entweder eine interne oder no-Reduktion. Durch Vertauschen der internen und no-Reduktionen kann aus einer gegebenen Reduktion eine variierte Reduktion hergeleitet werden.

Auf Basis dieser Überlegung reicht es zu zeigen, daß sich jede Reduktion derart modifizieren läßt, daß alle no-Reduktionen am Anfang und alle internen Reduktionen am Ende vorkommen. Da interne Reduktionen wie bereits gezeigt einen Ausdruck aus WHNF nicht herausführen können ist damit gezeigt, daß eine no-Reduktion zu WHNF existiert.

Wir werden dies mittels Vertauschungsdiagrammen zeigen, die wir für alle Reduktionstypen des Λ_{let}-Kalküls formulieren. Eine dabei auftretende Schwierigkeit ist zu beweisen, daß dieser Vertauschungsprozess stets terminiert. Eine Ursache hierfür besteht darin, daß die Bewegung einer (no, cp)-Reduktion an einer internen-Reduktion vorbei zu einer Vervielfachung der internen Reduktion führen kann. Jedoch überschneiden sich die einzelnen Redizes der dabei hinzukommenden internen Reduktionen nicht, was zur Definition der *cppar*-Reduktion geführt hat, die wir im nun folgenden Abschnitt einführen und untersuchen.

2.4.4 Die *cppar*-Reduktion

Definition 2.4.6. *t sei ein geschlossener Ausdruck. Die Reduktion $t \xrightarrow{cppar} t'$ wird wie folgt definiert:*
Für $x, y \in V_{let}(t)$ gelte $x <\cdot\ y$, gdw. (`let` $y = t_y$ `in` s) ein Subausdruck von t ist und x in t_y vorkommt. $<$ bezeichne den transitiven Abschluß der Relation $<\cdot$. Es werde eine Antikette $A \subseteq V_{let}(t)$ ausgewählt, d.h. kein Variablenpaar in A steht in der Beziehung $<$. Die Definition der Relation $\xrightarrow{cppar,A}$ für Subausdrücke s von t erfolgt dann wie folgt:

- $x \xrightarrow{cppar,A} x$
- $x \xrightarrow{cppar,A} t'_x$ *wenn $x \in A$, x in t eine Abstraktion, die t_x mittels eines Subausdrucks (`let` $x = t_x$ `in` t_s) bindet, und t'_x eine umbenannte Kopie von t_x ist.*
- `choice` $\xrightarrow{cppar,A}$ `choice`
- *Wenn $(s_1 s_2)$ ein Subausdruck von t ist und $s_1 \xrightarrow{cppar,A} s'_1$ sowie $s_2 \xrightarrow{cppar,A} s'_2$ gilt, dann $(s_1 s_2) \xrightarrow{cppar,A} (s'_1 s'_2)$.*
- *Wenn $(\lambda z.s_1)$ ein Subausdruck von t ist und $s_1 \xrightarrow{cppar,A} s'_1$ gilt, dann $(\lambda z.s_1) \xrightarrow{cppar,A} (\lambda z.s'_1)$.*

- *Wenn* (`let` $z = t_z$ `in` s_z) *ein Subausdruck von* t *ist und* $t_z \xrightarrow{cppar,A} t'_z$ *sowie* $s_z \xrightarrow{cppar,A} s'_z$ *gilt, dann* (`let` $z = t_z$ `in` s_z) $\xrightarrow{cppar,A}$ (`let` $z = t'_z$ `in` s'_z).

Wir schreiben $t \xrightarrow{cppar} t'$ *gdw.* $t \xrightarrow{cppar,A} t'$ *für irgendeine Antikette* A.

Wir werden nun zeigen, daß Folgen einzelner *cp*-Reduktionen, die bestimmten Anforderungen genügen, einer einzelnen *cppar*-Reduktion entsprechen, und im darauffolgenden Lemma umgekehrt, daß jede *cppar*-Reduktion einer Folge einzelner *cp*-Reduktionen entspricht. Zuvor definieren noch *cp*-Reduktion, die auf einer bestimmten Antikette basieren.

Definition 2.4.7. *Gegeben eine Antikette* A. *Eine* (cp, A)*-Reduktion ist eine cp-Reduktion*

$$C[\texttt{let } x = r \texttt{ in } D[x]] \xrightarrow{cp} C[\texttt{let } x = r \texttt{ in } D[r']]$$

bei der $x \in A$ *gilt.*

Lemma 2.4.1. t *und* t' *seien zwei geschlossene Ausdrücke. Wenn* $t \xrightarrow{cp,*} t'$ *und die Menge aller in den einzelnen cp-Reduktionen ersetzten Variablen bildet eine Antikette in* t*, dann gilt* $t \xrightarrow{cppar} t'$.

Beweis. Wir führen eine Induktion über die Menge aller der Länge nach geordneten Präfixreduktionen von $t \xrightarrow{cp,*} t'$. Die Induktionshypothese lautet, daß eine Folge von *cp*-Reduktionen, bei der die Menge aller ersetzten Variablen eine Antikette bildet, einer einzelnen *cppar*-Reduktion entspricht.

Induktionsbeginn:
Eine einzelne *cp*-Reduktion entspricht immer einer *cppar*-Reduktion.

Induktionsschritt ($t \xrightarrow{cp,*} s \xrightarrow{cp} s'$)*:*
Für die Reduktionsfolge $t \xrightarrow{cp,*} s$ gilt gemäß Induktionsvoraussetzung $t \xrightarrow{cppar} s$. Die Reduktion $s \xrightarrow{cp} s'$ entspreche $C[x] \xrightarrow{cp} C[t_x]$. Wir leiten $t \xrightarrow{cppar} s'$ aus Definition 2.4.6 her, indem an allen Stellen, wo bei der Bildung von $t \xrightarrow{cppar} s$ eine Ersetzung erfolgte, wiederum eine Ersetzung vorgenommen wird und zusätzlich das Vorkommen der Variablen x bei Besuch der Stelle $C[x]$ durch t_x ersetzt wird. Letzteres ist stets zulässig, da die Menge aller zu ersetzenden Variablen gemäß Voraussetzung eine Antikette in t bildet. □

Lemma 2.4.2. t *und* t' *seien zwei geschlossene Ausdrücke. Wenn* $t \xrightarrow{cppar} t'$*, dann* $t \xrightarrow{cp,A,*} t'$ *für eine Antikette* A.

Beweis. Aus $t \xrightarrow{cppar} t'$ folgt die Existenz einer Antikette A, so daß $t \xrightarrow{cppar,A} t'$. Bei der Bildung von $t \xrightarrow{cppar,A} t'$ wird beim Besuch jeder Variable, die in der Menge A enthalten ist und in einem übergeordneten Let-Kontext eine Abstraktion bindet, eine Auswahl dahingehend getroffen, ob die Variable beibehalten oder durch die gebundene Abstraktion ersetzt wird; weitere Wahlmöglichkeiten bestehen nicht. Jede dieser Ersetzungen für sich genommen entspricht einer einzelnen *cp*-Reduktion. Die Antikette A gewährleistet, daß kein Ausdruck, der bei diesem Ersetzungsprozess anstelle einer Variablen eingesetzt wird, zuvor selbst Ziel einer Ersetzung war. Somit läßt sich das Ergebnis des Ersetzungsprozesses der *cppar*-Reduktion auch durch eine Folge beliebig permutierbarer einzelner *cp*-Reduktionen erzielen. □

In den nun folgenden vier Lemmata wird ein vollständiger Satz von Vertauschungsdiagramme für die *cppar*-Reduktion entwickelt.

Lemma 2.4.3. *Gegeben eine Antikette A. Jede* (i, cp, A)*-Reduktion kann gemäß eines der beiden folgenden Vertauschungsdiagramme mit einer anschließenden* (no, cp)*-Reduktion vertauscht werden:*

1) $(i, cp, A) \circ (no, cp) \rightsquigarrow (no, cp) \circ (a, cp, A)$ *wobei* $a \in \{no, i\}$

2) $(i, cp, A) \circ (no, cp) \rightsquigarrow (no, cp) \circ (a, cp, A') \circ (i, cp, A')$ *wobei* $A \subseteq A'$ *und* A' *eine Antikette sowie* $a \in \{no, i\}$

Beweis. Es ist erforderlich, die verschiedenen Überlappungsmöglichkeiten des (i, cp, A)- und (no, cp)-Redex zu verifizieren. Eine Anmerkung in Lemma 2.3.5 besagt, daß bei einem gegebenen Ausdruck, der nicht in WHNF ist, der Typ des no-Redex durch die Reduktion eines internen Redex nicht verändert werden kann. Dies gewährleistet, daß der am Beginn der Reduktionsfolge stehende Ausdruck stets einen no-Redex vom Typ *cp* besitzt. Die (no, cp)-Reduktion ersetze eine Variable z durch eine Kopie eines Subausdrucks t_z. Die (i, cp, A)-Reduktion ersetze eine Variable x durch eine Kopie eines Subausdrucks t_x. Folgende Fälle können in Abhängigkeit von A unterschieden werden:

$A \cup \{z\}$ *ist eine Antikette und* $z \notin A$
Beide Reduktionen sind komplett unabhängig und können direkt getauscht werden. Die interne Reduktion kann sich durch den Tausch jedoch in eine no-Reduktion wandeln.

$A \cup \{z\}$ *ist eine Antikette und* $z \in A$
Es existieren zwei Möglichkeiten:
Gilt für die zu ersetzenden Variablen $x \not\equiv y$, sind beide Reduktionen vollständig unabhängig voneinander. Ansonsten besteht folgende Situation:
(no, cp)-Reduktion: $C[\texttt{let}\ x = t_x\ \texttt{in}\ D[x]]$
(i, cp, A)-Reduktion: $C[\texttt{let}\ x = t_x\ \texttt{in}\ D'[x]]$

In beiden Fällen können die zwei Reduktionen bei Ergebnisgleichheit direkt getauscht werden, jedoch kann sich die interne Reduktion dabei in eine no-Reduktion wandeln.

$A \cup \{z\}$ ist keine Antikette, da $x > z$
Wir zeigen, daß dieser Fall nicht möglich bzw. die Antikette nicht sinnvoll gewählt wurde. Ohne Beschränkung der Allgemeinheit gehen wir davon aus, daß in A nur solche Variablen vorkommen, die tatsächlich mit einem *cp*-Redex assoziiert werden können. Da gemäß Korollar 2.3.1 ein (no, cp)-Redex nicht als Teil des kopierbaren Subausdruck eines anderen *cp*-Redex vorkommen kann, existiert, obige Beschränkung vorausgesetzt, keine Variable $z' \in A$ mit $z' > z$.

$A \cup \{z\}$ ist keine Antikette, da $x < z$
Folgende drei Überlappungssituationen sind möglich:

i) Das Kopierziel der (i, cp)-Reduktion befindet sich nicht im Ausdruck t_z, d.h. es besteht eine Situation $(\texttt{let}\ x = t_x\ \texttt{in}\ C[\texttt{let}\ z = t_z\ \texttt{in}\ s_z])$ und s_z entspricht zum einen $D[x]$ und zum anderen $D'[z]$. Die Reduktionen sind hier entsprechend der zuletzt gezeigten Fälle direkt vertauschbar, da beide Kopierziele in s_z liegen.

ii) Das Kopierziel der (i, cp)-Reduktion befindet sich im Ausdruck t_z, jedoch erfolgt die Bindung von t_x außerhalb von t_z. Die zugehörige Reduktionsfolge

$$\begin{array}{rl}
 & (\texttt{let}\ x = t_x\ \texttt{in}\ C[\texttt{let}\ z = t_z[x]\ \texttt{in}\ D[z]]) \\
\xrightarrow{i,cp} & (\texttt{let}\ x = t_x\ \texttt{in}\ C[\texttt{let}\ z = t_z[t'_x]\ \texttt{in}\ D[z]]) \\
\xrightarrow{no,cp} & (\texttt{let}\ x = t_x\ \texttt{in}\ C[\texttt{let}\ z = t_z[t'_x]\ \texttt{in}\ D[t'_z[t''_x]]])
\end{array}$$

kann alternativ durch

$$\begin{array}{rl}
 & (\texttt{let}\ x = t_x\ \texttt{in}\ C[\texttt{let}\ z = t_z[x]\ \texttt{in}\ D[z]]) \\
\xrightarrow{no,cp} & (\texttt{let}\ x = t_x\ \texttt{in}\ C[\texttt{let}\ z = t_z[x]\ \texttt{in}\ D[t'_z[x]]]) \\
\xrightarrow{\{no,i\},cp} & (\texttt{let}\ x = t_x\ \texttt{in}\ C[\texttt{let}\ z = t_z[x]\ \texttt{in}\ D[t'_z[t''_x]]]) \\
\xrightarrow{i,cp} & (\texttt{let}\ x = t_x\ \texttt{in}\ C[\texttt{let}\ z = t_z[t'_x]\ \texttt{in}\ D[t'_z[t''_x]]])
\end{array}$$

dargestellt werden. Wesentlich ist die Reihenfolge der beiden letzten Reduktionen, die direkt austauschbar sind. Die vorletzte Reduktion kann eine no-Reduktion sein und muß aus später vorkommenden Ordnungsgesichtspunkten immer zuerst erfolgen.

iii) Der gesamte Redex der (i, cp)-Reduktion ist ein Subausdruck von t_z. Die zugehörige Reduktionsfolge

$$\begin{array}{rl}
 & (\texttt{let}\ z = t_z[(\texttt{let}\ x = t_x\ \texttt{in}\ C[x])]\ \texttt{in}\ D[z]]) \\
\xrightarrow{i,cp,A} & (\texttt{let}\ z = t_z[(\texttt{let}\ x = t_x\ \texttt{in}\ C[t'_x])]\ \texttt{in}\ D[z]]) \\
\xrightarrow{no,cp} & (\texttt{let}\ z = t_z[(\texttt{let}\ x = t_x\ \texttt{in}\ C[t'_x])]\ \texttt{in}\ D[t'_z[(\texttt{let}\ x' = t''_x\ \texttt{in}\ C'[t'''_x])]]])
\end{array}$$

kann alternativ durch

$$\begin{array}{ll}
 & (\texttt{let}\ z = t_z[(\texttt{let}\ x = t_x\ \texttt{in}\ C[x])]\ \texttt{in}\ D[z]]) \\
\xrightarrow{no,cp} & (\texttt{let}\ z = t_z[(\texttt{let}\ x = t_x\ \texttt{in}\ C[x])]\ \texttt{in}\ D[t'_z[(\texttt{let}\ x' = t''_x\ \texttt{in}\ C'[x'])]]]) \\
\xrightarrow{\{no,i\},cp,A'} & (\texttt{let}\ z = t_z[(\texttt{let}\ x = t_x\ \texttt{in}\ C[x])]\ \texttt{in}\ D[t'_z[(\texttt{let}\ x' = t''_x\ \texttt{in}\ C'[t'''_x])]]]) \\
\xrightarrow{i,cp,A'} & (\texttt{let}\ z = t_z[(\texttt{let}\ x = t_x\ \texttt{in}\ C[t'_x])]\ \texttt{in}\ D[t'_z[(\texttt{let}\ x' = t''_x\ \texttt{in}\ C'[t'''_x])]]])
\end{array}$$

dargestellt werden, wobei $A' = A \cup \{x'\}$. A' ist wie A eine Antikette, da t'_x eine umbenannte Kopie von t_x ist und somit alle Aussagen bezüglich freier Variablen von t_x auf t'_x übertragbar sind. Ferner besitzt x' dieselbe Ordnungsposition in A' wie x, d.h. für alle $y \in A'$ mit $y \not\equiv x$ und $y \not\equiv x'$ gilt $y > x$ gdw. $y > x'$. Bezüglich der Anordnung der letzten und vorletzten Reduktion gelten die Aussagen von *ii)*. □

Lemma 2.4.4. *Jede $(i, cppar)$-Reduktion kann gemäß des folgenden Vertauschungsdiagramms mit einer anschließenden (no, cp)-Reduktion vertauscht werden:*

$$(i, cppar) \circ (no, cp) \rightsquigarrow (no, cp, +) \circ (i, cppar, 0/1)$$

Beweis. Gemäß Lemma 2.4.2 existiert eine Antikette A, so daß die $(i, cppar)$-Reduktion einer Folge von (i, cp, A)-Reduktionen entspricht. Wir zeigen daher zunächst:

$(i, cp, A, *) \circ (no, cp) \rightsquigarrow (no, cp, +) \circ (i, cp, A', *)$, wobei $A \subseteq A'$ und A' eine Antikette.

Dazu benutzen wir folgende Verfahrensweise:
Solange eine (i, cp)-Reduktion existiert, die direkt links vor einer (no, cp)-Reduktion steht, bewegen wir wir diese gemäß der Regeln aus Lemma 2.4.3 nach rechts.
Wir zeigen nun, daß dieser Prozess stets terminiert. Für eine gegebene Folge von (no, cp)- und (i, cp)-Reduktionen definieren wir folgendes Maß: Für jede (i, cp)-Reduktion zählen wir die Anzahl der (no, cp)-Reduktionen, die rechts von ihr stehen. Wir ordnen die erhaltene Menge von Zahlwerten nach der Ordnung für Multimengen[5]. Offensichtlich erniedrigt jede Bewegung einer (no, cp)-Reduktion gemäß der Regeln aus Lemma 2.4.3 dieses Maß.

Bei dem obigen Prozess können (i, cp)-Reduktionen entstehen, die auf einer um eine Variable gegenüber A erweiterten Antikette basieren. Da alle so neu hinzukommenden Variablen aber immer ordnungsidentisch mit einer bereits in der Antikette A vorkommenden Variable sind, bildet die Vereinigung der hinzukommenden Variablen mit A wiederum eine Antikette; diese sei A'. Ferner entstehen alle in A' gegenüber A hinzugekommenen Variablen im Rahmen der nach links bewegten (no, cp)-Reduktionen, d.h derjenige Ausdruck, der nach Abschluß des

---

[5] Bezüglich Ordnungen auf Multimengen und ihre Eigenschaften sei wie schon in Abschnitt 2.2.1 auf [DM79] verwiesen.

Bewegungsprozesses zwischen der letzten (no, cp)- und ersten (i, cp)-Reduktion steht, beinhaltet bereits alle Variablen der Antikette A'. Die als Ergebnis des Bewegungsprozesses entstandene Folge von (i, cp)-Reduktionen kann somit gemäß Lemma 2.4.1 zu einer einzelnen $(i, cppar)$-Reduktion zusammengefaßt werden. Da bei dem Bewegungsprozeß beliebig viele (i, cp)-Reduktionen in (no, cp)-Reduktionen gewandelt werden können, besteht die Möglichkeit, daß alle (i, cp)-Reduktionen und damit auch die $(i, cppar)$-Reduktion verschwinden. Die Umkehrung ist jedoch nicht möglich, daher hat die als Ergebnis des Bewegungsprozesses entstandene Reduktionsfolge mindestens eine (no, cp)-Reduktion. □

Das folgende Lemma benötigen wir als Vorbereitung von Lemma 2.4.6.

Lemma 2.4.5. *Jede (i, cp)-Reduktion kann gemäß des folgenden Vertauschungsdiagramms mit einer anschließenden (no, a)-Reduktion, wobei $a \in \{llet, lapp, lbeta, nd\}$, vertauscht werden:*

$$(i, cp) \circ (no, a) \rightsquigarrow (no, a) \circ (b, cp), \text{ wobei } b \in \{i, no\}$$

Beweis. Gemäß Korollar 2.3.1 ist eine no-Redex niemals Teil des kopierbaren Subausdrucks eines *cp*-Redex. Somit kann ein no-Redex durch eine *cp*-Reduktion weder bewegt, noch verdoppelt werden. Da die no-Reduktion nicht vom Typ *cp* sein darf, kann das Ziel der (i, cp)-Reduktion durch die (no, cp)-Reduktion zwar bewegt jedoch nicht vervielfacht werden. Beide können somit stets direkt getauscht werden. Jedoch kann sich die (i, cp)-Reduktion bei der Vertauschung in eine (no, cp)-Reduktion wandeln. □

Lemma 2.4.6. *Die $(i, cppar)$-Reduktion verfügt über den folgenden vollständigen Satz von Vertauschungsdiagrammen:*

$$\begin{gathered}(i, cppar) \circ (no, a) \rightsquigarrow (no, a) \circ (no, cp, *) \circ (i, cppar, 0/1) \\ \text{für } a \in \{llet, lapp, lbeta, cp, nd\}\end{gathered}$$

Beweis. Gemäß Lemma 2.4.2 läßt sich die einzelne *cppar*-Reduktion in eine Folge $\mathcal{F}_A$ beliebig permutierbarer *cp*-Reduktionen auf Basis einer Antikette A auflösen. Durch wiederholtes Anwenden von Lemma 2.4.5 wird die einzelne (no, a)-Reduktion schrittweise durch die Folge $\mathcal{F}_A$ hindurch an die ganz linke Position bewegt. Dabei können einzelne (i, cp)-Reduktionen in eine (no, cp)-Reduktion gewandelt werden, jedoch bleibt die beliebige Vertauschbarkeit aller *cp*-Reduktionen erhalten, obgleich die mit der (no, a)-Reduktion gekoppelten Bewegungen bei einer oder mehreren (i, cp)-Reduktionen die Bewegung des Kopierziels bewirken können. In einem zweiten Schritt bewegen wir alle zuletzt entstandenen (no, cp)-Reduktionen an die Position direkt rechts neben der (no, a)-Reduktion. Die verbleibende Folge von (i, cp, A)-Reduktionen fassen wir gemäß Lemma 2.4.1 zu einer einzelnen $(i, cppar)$-Reduktion zusammen. □

2.4.5 Vertauschungsdiagramme für interne Reduktionen

Lemma 2.4.7. *Die Menge aller (i, a)-Reduktionen mit $a \in \{llet, lapp, lbeta, cppar, nd\}$ verfügt über den folgenden vollständigen Satz von Vertauschungsdiagrammen:*

i) $(i, a) \circ (no, b) \leadsto (no, b) \circ (c, a)$
für $a \in \{llet, lapp, lbeta, (nd, d)\}$, $b \in \{llet, lapp, lbeta, cp, (nd, e)\}$,
$c \in \{i, no\}$, $d, e \in \{left, right\}$

ii) $(i, a) \circ (no, cp) \leadsto (no, cp) \circ (c, a) \circ (i, a)$
für $a \in \{llet, lapp, lbeta\}$, $c \in \{i, no\}$

iii) $(i, cppar) \circ (no, a) \leadsto (no, a) \circ (no, cp, *) \circ (i, cppar, 0/1)$
für $a \in \{llet, lapp, lbeta, cp, nd\}$

iv) $(i, llet) \circ (no, lapp, m) \circ (no, llet) \circ (no, lapp, m) \circ (no, llet)$
$\leadsto (no, lapp, m) \circ (no, llet) \circ (c, llet)$ *für* $c \in \{i, no\}$ *und* $m \geq 0$

v) $(i, llet) \circ (no, lapp, 2*m) \leadsto (no, lapp, m) \circ (c, llet)$ *für* $c \in \{i, no\}$ *und* $m \geq 1$

Beweis. Es sind alle Möglichkeiten eines internen gefolgt von einem no-Redex zu verifizieren. Wir benutzen dabei folgende Methodik:
Aus Korollar 2.3.1 kennen wir die Kontextformen, in denen no-Redizes der verschiedenen Redex-Typen vorkommen können. Darauf basierend zeigen wir für allen Typen von no-Redizes, daß bei einem beliebig positionierten internen Redex stets eine der obigen Vertauschungsregeln anwendbar ist. Damit gewährleisten wir die Betrachtung aller Ausdrücke, die nicht in WHNF sind.

Wir betrachten nun alle Typen die ein no-Redex annehmen kann. Dabei gilt die Einschränkung, daß bei der (i, a)-Reduktion kein Redex vom Typ *cp* reduziert wird. Die Vertauschungsregeln für diesen Fall werden in Lemma 2.4.6 gezeigt.

$(no, lbeta)$
Ein no-Redex vom Typ *lbeta* hat stets die Form $W[(\lambda x.s)t]$. Nach Anwendung einer beliebigen internen Reduktion entsteht wiederum ein Ausdruck mit einem no-Redex vom Typ *lbeta*. Es ist offensichtlich, daß alle möglichen internen Reduktionen, die nicht vom Typ *cp* sind, direkt mit einer $(no, lbeta)$-Reduktion getauscht werden können. Jedoch kann sich die interne Reduktion dabei in eine no-Reduktion wandeln.

(no, cp)
Es sind die zwei Fälle zu unterscheiden, ob sich der (i, a)-Redex innerhalb des kopierbaren Subausdrucks einer (no, cp)-Reduktion befindet oder nicht. Befindet sich der (i, a)-Redex nicht im kopierbaren Subausdruck einer (no, cp)-Reduktion,

können beide Reduktionen direkt getauscht werden, wobei sich die interne Reduktion in eine no-Reduktion wandeln kann. Ansonsten stellt sich die Situation wie folgt dar:

$$\begin{array}{ll}
 & L_R^*[\mathtt{let}\ x = t_x\ \mathtt{in}\ W[x]] \\
\xrightarrow{i,a} & L_R^*[\mathtt{let}\ x = t'_x\ \mathtt{in}\ W[x]] \\
\xrightarrow{no,cp} & L_R^*[\mathtt{let}\ x = t'_x\ \mathtt{in}\ W[t'_x]]
\end{array}$$

entspricht der Reduktionsfolge:

$$\begin{array}{ll}
 & L_R^*[\mathtt{let}\ x = t_x\ \mathtt{in}\ W[x]] \\
\xrightarrow{no,cp} & L_R^*[\mathtt{let}\ x = t_x\ \mathtt{in}\ W[t_x]] \\
\xrightarrow{\{no,i\},cp} & L_R^*[\mathtt{let}\ x = t_x\ \mathtt{in}\ W[t'_x]] \\
\xrightarrow{i,cp} & L_R^*[\mathtt{let}\ x = t_{x'}\ \mathtt{in}\ W[t'_x]]
\end{array}$$

Wegen des W-Kontexts existiert keine interne Reduktion, die das Kopierziel der (no, cp)-Reduktion bewegen könnte. Der Fall eines internen nd-Redex bei t_x fällt weg, da t_x eine Abstraktion sein muß und die Reduktion von nd-Redizes innerhalb einer Abstraktion nicht zulässig ist.

(no, nd)
Wir betrachten beispielhaft den Fall, daß der (i, a)-Redex innerhalb eines (no, nd)-Redex vorkommt und der (i, a)-Redex selbst nicht vom Typ nd ist:

$$\begin{array}{ll}
 & W[\mathtt{choice}\ s] \\
\xrightarrow{i,a} & W[\mathtt{choice}\ s'] \\
\xrightarrow{no,nd} & W[(\lambda x.(\lambda y.y))s'] \text{ oder } W[(\lambda x.(\lambda y.x))s']
\end{array}$$

entspricht der Reduktionsfolge:

$$\begin{array}{llcll}
 & W[\mathtt{choice}\ s] & & & \\
\xrightarrow{no,nd,right} & W[(\lambda x.(\lambda y.y))s] & \text{oder} & \xrightarrow{no,nd,left} & W[(\lambda x.(\lambda y.x))s] \\
\xrightarrow{i,a} & W[(\lambda x.(\lambda y.y))s'] & & \xrightarrow{i,a} & W[(\lambda x.(\lambda y.x))s']
\end{array}$$

Auch in allen übrigen Fällen bestehen keine strukturellen Abhängigkeiten und die (i, a)-Reduktion kann mit der (no, nd)-Reduktion gemäß obigem Schema getauscht werden.

$(no, llet)$
Ein $(no, llet)$-Redex kann strukturell mit einem $(i, llet)$-Redex überlappen. Es besteht dann folgende Situation:

$$\begin{array}{ll}
 & L_R^*[\mathtt{let}\ z = (\mathtt{let}\ x = (\mathtt{let}\ y = t_y\ \mathtt{in}\ t_x)\ \mathtt{in}\ t_z)\ \mathtt{in}\ t] \\
\xrightarrow{i,llet} & L_R^*[\mathtt{let}\ z = (\mathtt{let}\ y = t_y\ \mathtt{in}\ (\mathtt{let}\ x = t_x\ \mathtt{in}\ t_z))\ \mathtt{in}\ t] \\
\xrightarrow{no,llet} & L_R^*[\mathtt{let}\ y = t_y\ \mathtt{in}\ (\mathtt{let}\ z = (\mathtt{let}\ x = t_x\ \mathtt{in}\ t_z)\ \mathtt{in}\ t)] \\
\xrightarrow{no,llet} & L_R^*[\mathtt{let}\ y = t_y\ \mathtt{in}\ (\mathtt{let}\ x = t_x\ \mathtt{in}\ (\mathtt{let}\ z =\ \ \mathtt{in}\ t_z\ \mathtt{in}\ t))]
\end{array}$$

entspricht der Reduktionsfolge:

$$\begin{array}{ll}
 & L^*_R[\texttt{let } z = (\texttt{let } x = (\texttt{let } y = t_y \texttt{ in } t_x) \texttt{ in } t_z) \texttt{ in } t] \\
\xrightarrow{no,llet} & L^*_R[\texttt{let } x = (\texttt{let } y = t_y \texttt{ in } t_x) \texttt{ in } (\texttt{let } z = t_z \texttt{ in } t)] \\
\xrightarrow{\{i,no\},llet} & L^*_R[\texttt{let } y = t_y \texttt{ in } (\texttt{let } x = t_x \texttt{ in } (\texttt{let } z = \texttt{ in } t_z \texttt{ in } t))]
\end{array}$$

In allen übrigen Fällen bestehen keine strukturellen Abhängigkeiten und die (i, a)-Reduktion kann direkt mit der $(no, llet)$-Reduktion getauscht werden, jedoch kann sich die interne Reduktion dabei in eine no-Reduktion wandeln.

$(no, lapp)$
Ein $(no, lapp)$-Redex kann strukturell mit einem $(i, llet)$-Redex überlappen. In Abhängigkeit davon, ob der $(no, lapp)$-Redex in einem Kontext mit einem L_L-Subkontext vorkommt oder nicht, bestehen die beiden folgenden Möglichkeiten:

1) $(no, lapp)$ und $(i, llet)$, wobei kein L_L-Kontext vorhanden:

$$\begin{array}{ll}
 & L^*_R[A^m_L[(\texttt{let } x = (\texttt{let } y = t_y \texttt{ in } t_x) \texttt{ in } s)t]] \\
\xrightarrow{i,llet} & L^*_R[A^m_L[(\texttt{let } y = t_y \texttt{ in } (\texttt{let } x = t_x \texttt{ in } s))t]] \\
\xrightarrow{no,lapp} & L^*_R[A^m_L[\texttt{let } y = t_y \texttt{ in } (\texttt{let } x = t_x \texttt{ in } s)t]] \\
\xrightarrow{no,lapp,m} & L^*_R[\texttt{let } y = t_y \texttt{ in } A^m_L[(\texttt{let } x = t_x \texttt{ in } s)t]] \\
\xrightarrow{no,lapp} & L^*_R[\texttt{let } y = t_y \texttt{ in } A^m_L[\texttt{let } x = t_x \texttt{ in } (st)]] \\
\xrightarrow{no,lapp,m} & L^*_R[\texttt{let } y = t_y \texttt{ in } (\texttt{let } x = t_x \texttt{ in } A^m_L[st])]
\end{array}$$

entspricht der Reduktionsfolge:

$$\begin{array}{ll}
 & L^*_R[A^m_L[(\texttt{let } x = (\texttt{let } y = t_y \texttt{ in } t_x) \texttt{ in } s)t]] \\
\xrightarrow{no,lapp} & L^*_R[A^m_L[\texttt{let } x = (\texttt{let } y = t_y \texttt{ in } t_x) \texttt{ in } (st)]] \\
\xrightarrow{no,lapp,m} & L^*_R[\texttt{let } x = (\texttt{let } y = t_y \texttt{ in } t_x) \texttt{ in } A^m_L[st]] \\
\xrightarrow{\{i,no\},llet} & L^*_R[\texttt{let } y = t_y \texttt{ in } (\texttt{let } x = t_x \texttt{ in } A^m_L[st])]
\end{array}$$

2) $(no, lapp)$ und $(i, llet)$, wobei L_L-Kontext vorhanden:

$$\begin{array}{ll}
 & L^*_R[L_L[A^m_L[(\texttt{let } x = (\texttt{let } y = t_y \texttt{ in } t_x) \texttt{ in } s)t]]] \\
\xrightarrow{i,llet} & L^*_R[L_L[A^m_L[(\texttt{let } y = t_y \texttt{ in } (\texttt{let } x = t_x \texttt{ in } s))t]]] \\
\xrightarrow{no,lapp} & L^*_R[L_L[A^m_L[\texttt{let } y = t_y \texttt{ in } (\texttt{let } x = t_x \texttt{ in } s)t]]] \\
\xrightarrow{no,lapp,m} & L^*_R[L_L[\texttt{let } y = t_y \texttt{ in } A^m_L[(\texttt{let } x = t_x \texttt{ in } s)t]]] \\
\xrightarrow{no,llet} & L^*_R[\texttt{let } y = t_y \texttt{ in } L_L[A^m_L[(\texttt{let } x = t_x \texttt{ in } s)t]]] \\
\xrightarrow{no,lapp,m+1} & L^*_R[\texttt{let } y = t_y \texttt{ in } L_L[\texttt{let } x = t_x \texttt{ in } A^m_L[st]]] \\
\xrightarrow{no,llet} & L^*_R[\texttt{let } y = t_y \texttt{ in } (\texttt{let } x = t_x \texttt{ in } L_L[A^m_L[st]])]
\end{array}$$

entspricht der Reduktionsfolge:

$$\begin{array}{ll}
 & L^*_R[L_L[A^m_L[(\texttt{let } x = (\texttt{let } y = t_y \texttt{ in } t_x) \texttt{ in } s)t]]] \\
\xrightarrow{no,lapp} & L^*_R[L_L[A^m_L[\texttt{let } x = (\texttt{let } y = t_y \texttt{ in } t_x) \texttt{ in } (st)]]]
\end{array}$$

$$
\begin{aligned}
&\xrightarrow{no,lapp,m} L_R^*[L_L[\texttt{let}\ x = (\texttt{let}\ y = t_y\ \texttt{in}\ t_x)\ \texttt{in}\ A_L^m[st]]] \\
&\xrightarrow{no,llet} L_R^*[\texttt{let}\ x = (\texttt{let}\ y = t_y\ \texttt{in}\ t_x)\ \texttt{in}\ L_L[A_L^m[st]]] \\
&\xrightarrow{\{i,no\},llet} L_R^*[\texttt{let}\ y = t_y\ \texttt{in}\ (\texttt{let}\ x = t_x\ \texttt{in}\ L_L[A_L^m[st]])]
\end{aligned}
$$

In allen übrigen Fällen können die (i, a)-Reduktion und die $(no, lapp)$-Reduktion direkt getauscht werden, da keine strukturellen Abhängigkeiten bestehen können. Die interne Reduktion kann sich dabei jedoch in eine no-Reduktion wandeln.

(i, nd)
Abschließend soll noch der Fall einer (i, nd)-Reduktion gesondert betrachtet werden. Die Vertauschungsdiagramme können aus folgendem Reduktionszusammenhang abgeleitet werden:

$$\begin{tikzcd}
W[\texttt{choice}\ s] \arrow[r, "{i,nd,left}"] \arrow[rrd, "{i,nd,right}"] \arrow[dd, "{no,a}"] & W[(\lambda x.\lambda y.x)s] \arrow[dd, "{no,a}"] & \\
 & & W[(\lambda x.\lambda y.y)s] \arrow[dd, "{no,a}"] \\
W'[\texttt{choice}\ s'] \arrow[r, "{b,nd,left}"] \arrow[rrd, "{c,nd,right}"] & W'[(\lambda x.\lambda y.x)s'] & \\
 & & W'[(\lambda x.\lambda y.y)s']
\end{tikzcd}$$

wobei

- die (no, a)-Reduktion und die (i, nd)-Reduktion nicht die beiden Alternativen einer einzelnen nd-Reduktion sind
- und $b, c \in \{no, i\}$ jedoch nicht $b = no$ und $c = no$.

□

Lemma 2.4.8. *Gegeben zwei geschlossene Ausdrücke t und t'. Wenn $t \xrightarrow{i,a} s \xrightarrow{no,*} t'$ und t' in WHNF, dann existiert eine Reduktionsfolge $t \xrightarrow{no,*} t'' \xrightarrow{i,*} t'$ mit t'' in WHNF.*

Beweis. Wir benutzen die Vertauschungsregeln aus Lemma 2.4.7 um die gesuchte Reduktionsfolge zu erhalten. Dabei betrachten wir eine einzelne (i, cp)-Reduktion als eine $(i, cppar)$-Reduktion. Wir benutzen folgende Methodik:
Solange eine interne Reduktions existiert, die links von einer no-Reduktion steht, bewegen wir diese mittels der Vertauschungsregeln aus Lemma 2.4.7 nach rechts. Das folgende Maß zeigt, daß dieser Prozess stets terminiert:
Das Maß besteht aus drei lexikographisch geordneten Komponenten.

- Die erste Komponente besteht aus der Multimenge folgender Zahlwerte, geordnet nach der Ordnung für Multimengen: Für jede interne Reduktion die Anzahl von (no, cp)-Reduktionen, die rechts von ihr stehen.

	1.	2.	3.
i) und $c = i$	$<$oder $=$	$=$	$<$
i) und $c = no$	$=$	$<$	$>$
ii) und $c = i$	$<$	$>$	$<$
ii) und $c = no$	$<$	$=$	$>$
iii) und $(i, cppar)^0$	$=$	$<$	$<$
iii) und $(i, cppar)^1$	$=$	$=$	$<$
iv) und $c = i$	$=$	$=$	$<$
iv) und $c = no$	$=$	$<$	$<$ oder $=$
v) und $c = i$	$=$	$=$	$<$
v) und $c = no$	$=$	$<$	$<$ oder $=$

Tabelle 2.13:

- Die zweite Komponente ist die Gesamtzahl aller internen Reduktionen.
- Die dritte Komponente besteht aus der Multimenge folgender Zahlwerte geordnet nach der Ordnung für Multimengen: Für jede interne Reduktion die Anzahl no-Reduktionen, die rechts von ihr stehen.

Tabelle 2.13 zeigt, daß das obige Maß durch jede Anwendung einer Vertauschungsregel strikt erniedrigt wird. Dabei ist zu beachten, daß bei der Rechtsbewegung einer einzelnen $(i, cppar)$-Reduktion immer nur eine interne Reduktion bestehen bleibt und damit keine Vergrößerung der ersten Komponente möglich ist.

Somit wird nach endlich vielen Vertauschungen eine Reduktionsfolge $t \xrightarrow{no,*} t'' \xrightarrow{i,*} t'$ erreicht, wobei gemäß Korollar 2.3.2 t'' in WHNF, da t' in WHNF. □

Satz 2.4.1. *Gegeben zwei geschlossene Ausdrücke t und t'. Wenn $t \xrightarrow{*} t'$ und t' ist in WHNF, dann existiert ein Ausdruck t'' in WHNF, so daß $t \xrightarrow{no,*} t''$ und $t'' \xrightarrow{i,*} t'$.*

Beweis. Solange eine Zerlegung der Reduktionsfolge $t \xrightarrow{*} t'$ in $t \xrightarrow{*} t_a \xrightarrow{i,a} t_b \xrightarrow{no,*} t_c \xrightarrow{i,*} t'$ existiert, bewegen wir mittels Lemma 2.4.8 die (i, a)-Reduktion derart nach rechts, daß wir eine Reduktionsfolge $t \xrightarrow{*} t_a \xrightarrow{no,*} t'_b \xrightarrow{i,*} t_c \xrightarrow{i,*} t'$ erhalten. Dies ist zulässig, da gemäß Korollar 2.3.2 aus t' in WHNF folgt, daß t_c in WHNF ist. Existiert nach dem Abschluß einer Rechtsbewegung erneut eine Zerlegung gemäß dem geforderten Schema, dann besitzt die Reduktionsfolge zwischen t und dem dann neu gewählten t_c eine interne Reduktion weniger als zuvor. Da gemäß Voraussetzung nur endlich viele interne Reduktionen vorhanden sind, muß der Prozess somit stets terminieren, wobei die zuletzt erreichte Reduktionsfolge die

Form $t \xrightarrow{no,*} t''_c \xrightarrow{i,*} t'$ hat. Wir wählen die final erreichte Reduktionsfolge mit $t'' \equiv t''_c$; gemäß Korollar 2.3.2 ist t'' in WHNF, da t' in WHNF. □

Aus obigem Satz folgt als Korollar direkt die in diesem Abschnitt zu zeigende Aussage, daß ausgehend von einem Ausdruck t eine no-Reduktion zu einem Ausdruck in WHNF existiert, wenn ausgehend von t eine beliebige Reduktion zu einem Ausdruck in WHNF existiert.

Korollar 2.4.1. *Ein geschlossener Ausdruck t besitzt eine Reduktion zu einem Ausdruck in WHNF, gdw. ausgehend von t eine endlich beschränkte no-Reduktion existiert, d.h.* $t \xrightarrow{no,*} t'$ *für einen Ausdruck* t' *in WHNF.*

Satz 2.4.1 läßt einige weitere Schlußfolgerungen zu, die in den nun folgenden Korollaren festgehalten werden.

Korollar 2.4.2. *Wenn ein Ausdruck bis zu einer WHNF reduziert werden kann, dann verfügt die no-Reduktion bis zu einer WHNF über die kleinste Anzahl von nd-Reduktionen.*

Beweis. Keine Vertauschungsregel aus Lemma 2.4.7 hat das Vervielfachen einer nd-Reduktion zur Folge. Jedoch können (i, nd)-Reduktionen an den rechten Rand geschoben werden und damit für das Erreichen eines Ausdrucks in WHNF irrelevant sein. □

Korollar 2.4.3. *Jede unendliche no-Reduktion enthält eine unendliche Anzahl von* (no, cp)*-Reduktionen und eine unendliche Zahl von* $(no, lbeta)$*-Reduktionen.*

Beweis. Alleinig die Reduktion eines $lbeta$-Redex kann das in Definition 2.2.6 eingeführte Maß ψ erhöhen. Auf der anderen Seite kann alleinig die Reduktion eines cp-Redex das in Definition 2.2.4 eingeführte Maß φ erhöhen. □

Ein Beispiel für einen Ausdruck, der eine unendliche no-Reduktion besitzt, die alleinig aus $lbeta$- und cp-Reduktionen besteht, bildet der Ausdruck $(\lambda x.xx)\,(\lambda x.xx)$. Dies kann durch einfaches Reduzieren leicht nachgeprüft werden.

2.5 Abschließende Bemerkungen

Wir haben in diesem Abschnitt den Λ_{let}-Kalkül kennengelernt und gezeigt, daß dieser bei einer Ausdehnung der Sichtweise auf Mengen von Ausdrücken konfluent ist. Im Anschluß daran wurde eine normal-order Reduktion definiert und verschiedene elementare Eigenschaften von dieser gezeigt. Die hier definierte Form der no-Reduktion unterscheidet sich leicht von der in [AFM$^+$95] vorgestellten. Dieser Unterschied wird in Kapitel 7 noch einmal aufgegriffen und näher betrachtet werden. Eine Motivation für die Entwicklung des Λ_{let}-Kalküls war, wie in der Einführung erwähnt, die Schaffung einer formalen Grundlage für verzögernd auswertende funktionale Sprachen mit einem Seiteneffekt behafteten I/O. Als alternativer Ausgangspunkt hätte sich eine abstrakte Maschine angeboten, wie sie in [Ses97] vorgestellt wird. In Kapitel 6 wird daher eine zum Λ_{let}-Kalkül passende abstrakte Maschine vorgestellt und die Unterschiede zwischen beiden Modellen dargelegt. Wie bereits erwähnt, ist die no-Reduktion in der weiteren Arbeit von zentraler Bedeutung, insbesondere ihr Terminierungsverhalten. Sie wird in dem nun folgenden Kapitel die Grundlage einer wesentlichen Definition bilden, an der das Korrektheitsverständnis von Transformationen auf Ausdrucksebene verankert wird.

Kapitel 3

kontextuelle Äquivalenz

Im letzten Kapitel wurde der Begriff der Mengenkonfluenz eingeführt und für den Λ_{let}-Kalkül gezeigt, daß letzterer diese Art von Konfluenz erfüllt. Mengenkonfluenz erlaubt eine qualitative Einordnung des Λ_{let}-Kalküls im Kontext anderer Kalküle, da dem Konfluenzbegriff bei der Bewertung von Kalkülen große Bedeutung zugemessen wird. Außerhalb der 5 Reduktionen des Λ_{let}-Kalküls bietet dieser Begriff jedoch keine Möglichkeit, Ausdrücke in Beziehung zueinander zu setzen. Die beiden Ausdrücke $\mathtt{let}\ x = \lambda y.y\ \mathtt{in}\ \lambda z.z$ und $\lambda z.z$ sind mittels Reduktion nicht zu einem gemeinsamen Nachfahren überführbar und müssen somit als verschieden angesehen werden. Es wäre aber durchaus wünschenswert beide Ausdrücke als gleich anzusehen, denn wegen der fehlenden Bindung des x beim ersten Ausdruck verfügen beide Ausdrücke über denselben Kern. Eine Möglichkeit bestände darin, eine explizite Eliminationsregel für "Let-gebundene Ausdrücke ohne Vorkommen der bindenden Variable" in den Λ_{let}-Kalkül aufzunehmen, um beide Ausdrücke als gleich ansehen zu können. Eine Erweiterung des Λ_{let}-Kalküls durch die Aufnahme immer neuer Regeln führt jedoch zu einem immer komplizierteren Kalkül und damit einem immer komplizierteren Beweis der Mengenkonfluenz. Ein weiterer Schwachpunkt des Λ_{let}-Kalküls, der auf diese Weise nicht überwunden werden würde, ist seine fehlende Kompatibilität als Folge der kontextuellen Einschränkung bei der nd-Reduktion. Da die nd-Reduktion erhalten bleiben müßte, würden auch alle erweiterten Formen des Λ_{let}-Kalküls diese Schwachstelle aufweisen.

In diesem Kapitel werden wir eine Möglichkeit kennenlernen, die Zulässigkeit von Reduktionen zu zeigen, ohne diese Reduktionen in den Λ_{let}-Kalkül aufzunehmen oder den Λ_{let}-Kalkül irgendwie zu verändern oder zu erweitern. Letzterer Satz wirft die Frage auf, was unter einer zulässigen Reduktion zu verstehen ist. Die Zulässigkeit einer Reduktion erhalten wir über den Begriff der no-Reduktion, die wir im zweiten Teil des letzten Kapitels definiert und untersucht haben. Mittels der no-Reduktion können wir für einen Ausdruck t ein Profil seines Verhaltens beschreiben, indem wir für alle Kontexte C danach differenzieren, ob der Ausdruck

t eingesetzt in den Kontext C eine no-Reduktion zu WHNF oder eine nichtterminierende no-Reduktion besitzt. Dabei ist zu beachten, daß die no-Reduktion beim Λ_{let}-Kalkül wegen potentiell vorkommender (no, nd)-Reduktionen nicht eindeutig ist, weshalb ausgehend von einem gegebenen Ausdruck gleichzeitig eine no-Reduktion zu WHNF und eine nichtterminierende no-Reduktion existieren können.

Dieser Verhaltensgedanke ist vollständig von dem Konfluenzbegriff losgelöst. In [MS99] beispielsweise wird daher soweit gegangen, mittels einer abstrakten Maschine einen Kalkül alleinig auf seine no-Reduktion zu reduzieren und den Konfluenzbegriff vollständig auszublenden.
Mittels der Differenzierung nach dem no-Reduktionsverhalten bei allen umgebenden Kontexten ist es möglich, zwei Ausdrücke in Beziehung zueinander zu setzen. So kann ein Ausdruck t kleiner gleich einem Ausdruck s angesehen werden, wenn für jeden Kontext C gilt:

- wenn ausgehend von $C[t]$ eine no-Reduktion zu WHNF existiert, dann existiert auch ausgehend von $C[s]$ eine no-Reduktion zu WHNF und
- wenn ausgehend von $C[s]$ eine unendliche no-Reduktion existiert, dann existiert auch ausgehend von $C[t]$ eine unendliche no-Reduktion.

Dies ermöglicht es uns, eine Präordnung auf der Menge aller Λ_{let}-Ausdrücke anzugeben. Die in der Präordnung enthaltene Gleichheit können wir wiederum zur Formulierung eines Äquivalenzbegriffes benutzen, indem wir zwei Ausdrücke t, s als gleich ansehen, wenn t kleiner gleich s ist und umgekehrt. Genau dieser Äquivalenz entspricht die später definierte kontextuelle Äquivalenz. Zwei Ausdrücke werden somit als gleichwertig angesehen, wenn ihr Terminierungsverhalten in allen umgebenden Kontexten identisch ist. Eine Reduktion wird als zulässig angesehen, wenn bei ihrer Anwendung auf einen Ausdruck t stets ein zu t verhaltensgleicher Ausdruck entsteht. Es ist nicht offensichtlich, welche der 5 Reduktionen des Λ_{let}-Kalküls diese Eigenschaft erfüllen. In einem späteren Abschnitt dieses Kapitels werden wir zeigen, daß dies bei allen Reduktionen außer der nd-Reduktion der Fall ist.

Die Ordnung der Ausdrücke gemäß ihres kontextuellen Verhaltens beinhaltet ein beachtliches semantisches Leistungspotential. Als Beleg dafür sei auf Kapitel 5 verwiesen, wo Fixpunktaussagen getroffen werden, ohne daß auf eine denotationale Semantik zurückgegriffen wird.
Ein zunächst unerwartetes Charakteristikum der kontextuellen Äquivalenz ist, daß sie von ihren Eigenschaften her einer Reduktionsrelation entspricht, d.h sie ist reflexiv, transitiv und kompatibel. Dies ist um so interessanter, da im Zusammenhang mit der kontextuellen Äquivalenz keinerlei Redexbegriff formulierbar

ist. Zudem werden alle oben im Zusammenhang mit dem Konfluenzbegriff aufgeführten Kritikpunkte durch die kontextuelle Äquivalenz überwunden. Die beiden Ausdrücke `choice` $(\lambda x.x)(\lambda y.y)$ und $\lambda z.z$ sind kontextuell äquivalent; es ist uninteressant, wie oft derselbe Ausdruck erreicht werden kann, weil dieser immer dasselbe kontextuelle Verhalten aufweist.

Zu Beginn dieses Kapitels werden wir die kontextuelle Äquivalenz definieren und einige grundlegende Eigenschaften der letzteren zeigen. Im Anschluß daran wird ein Kontextlemma gezeigt, dessen Wert darin besteht, spätere Beweise zu vereinfachen. Wie oben erwähnt, ist es nicht offensichtlich welche Reduktionstypen des Λ_{let}-Kalküls die kontextuelle Äquivalenz erhalten. Dieser Thematik ist der dritte Abschnitt gewidmet. Im Anschluß daran erscheint ein Abschnitt, in dem vier weitere Reduktionsregeln eingeführt werden und gezeigt wird, daß diese die kontextuelle Äquivalenz erhalten. Dieser Teil verfolgt das alleinige Ziel, die kontextuelle Äquivalenz für die vier neuen Reduktionen zu zeigen und verfügt in der Folge nur über wenig begleitenden Text. Den aufwendigsten Beweis besitzt dabei die *ucp*-Reduktion, die vor allem in Kapitel 6 benötigt wird. Zum Abschluß zeigen wir noch die Zulässigkeit von Lambda-Lifting beim Λ_{let}-Kalkül unter Rückgriff auf einige der im vorhergehenden Abschnitt eingeführten Reduktionen.

3.1 Definition der kontextuellen Äquivalenz

Ausgehend von einem gegebenen Ausdruck können mehrere no-Reduktionen, potentiell sogar unendlich viele, existieren. Manche von diesen können bei einer WHNF oder VWHNF terminieren, andere können über unendliche viele einzelne Reduktionen verfügen. Im Kontext der nun folgenden Beweise wird die Einführung einer Notation erforderlich, die es erlaubt, solche Charakteristika der no-Reduktionen eines Ausdrucks zu spezifizieren.

Notation 3.1.1. *Gegeben ein Ausdruck t. Wir schreiben*

$t{\Downarrow}_{WHNF}$	*gdw. ausgehend von t existiert eine no-Reduktion zu einem Ausdruck in WHNF.*
$t{\Downarrow}_{VWHNF}$	*gdw. ausgehend von t existiert eine no-Reduktion zu einem Ausdruck in VWHNF.*
$t{\Downarrow}$	*gdw. $t{\Downarrow}_{WHNF}$ oder $t{\Downarrow}_{VWHNF}$*
$t{\Uparrow}$	*gdw. ausgehend von t existiert eine nichtterminierende no-Reduktion.*
$t{\not\Uparrow}$	*gdw. ausgehend von t existiert keine nichtterminierende no-Reduktion, d.h. alle von t ausgehenden no-Reduktionen terminieren bei einem Ausdruck in WHNF oder VWHNF.*

$t\not\Downarrow$ *gdw. ausgehend von t existiert keine no-Reduktion zu einem Ausdruck in WHNF, d.h. alle von t ausgehenden no-Reduktionen terminieren nicht.*

Die oben eingeführte Notation soll nun an einigen Beispielen vorgestellt werden:

Beispiel 3.1.1. Die folgende Tabelle enthält in jeder Zeile zuerst einen Ausdruck und anschließend die Eigenschaften der no-Reduktion dieses Ausdrucks:

Ausdruck t	Eigenschaften der no-Reduktion
$\lambda x.x$	$t\Downarrow_{WHNF}$, $t\Downarrow$, $t\not\Uparrow$
$(\lambda x.xx)(\lambda x.xx)$	$t\Uparrow$, $t\not\Downarrow$
$\texttt{choice}\ (\lambda x.x)((\lambda x.xx)(\lambda x.xx))$	$t\Downarrow_{WHNF}$, $t\Downarrow$, $t\Uparrow$
$\texttt{let}\ y = x\ \texttt{in}\ y \quad (x \in V_{free}(t))$	$t\Downarrow_{VWHNF}$, $t\Downarrow$, $t\not\Uparrow$
$\texttt{choice}\ (\texttt{let}\ y = x\ \texttt{in}\ y)(\lambda x.x)$	$t\Downarrow_{WHNF}$, $t\Downarrow_{VWHNF}$, $t\Downarrow$, $t\not\Uparrow$
$\texttt{choice}\ (\texttt{let}\ y = x\ \texttt{in}\ y)((\lambda x.xx)(\lambda x.xx))$	$t\Downarrow_{VWHNF}$, $t\Downarrow$, $t\Uparrow$

Alle Angaben lassen sich durch einfaches Reduzieren leicht überprüfen.

Wir treffen nun die in der Einleitung erwähnte Definition, die zwei Ausdrücke über ihr no-Reduktionsverhalten in allen möglichen umgebenden Kontexten in Beziehung setzt.

Definition 3.1.1. *s, t seien zwei beliebige Ausdrücke. Dann*

- *$s \leq_c t$ gdw. für alle Kontexte C mit $C[s]$ und $C[t]$ geschlossen :*
$$C[s]\Downarrow \Rightarrow C[t]\Downarrow \text{ und } C[t]\Uparrow \Rightarrow C[s]\Uparrow$$
- *$s \sim_c t$ gdw. $s \leq_c t$ und $t \leq_c s$*

Notation 3.1.2. *t, s seien zwei Ausdrücke.*

- *Wenn $t \leq_c s$, dann sprechen wir davon, daß t* kontextuell kleiner *als s ist.*
- *Wenn $t \sim_c s$, dann sprechen wir davon, daß t und s* kontextuell äquivalent *sind.*

Die folgenden Propositionen und Korollare zeigen einige grundlegende Eigenschaften der zuletzt definierten Relationen auf Λ_{let}.

Proposition 3.1.1. *$\leq_c$ ist eine Präkongruenz-Relation bzw. Reduktions-Relation.*

Beweis. Zu zeigen sind Reflexivität, Transitivität, sowie Kompatibilität mit allen Kontexten:

(Reflexivität)
$C[t]\Downarrow \Rightarrow C[t]\Downarrow \wedge C[t]\Uparrow \Rightarrow C[t]\Uparrow$

(Transitivität)
s, s', t seien Ausdrücke. Für alle Kontexte C mit $C[s], C[s'], C[t]$ geschlossen gilt

$$\begin{aligned}
(C[s]\Downarrow \Rightarrow C[s']\Downarrow) \wedge (C[s']\Downarrow \Rightarrow C[t]\Downarrow) \quad &\Rightarrow \quad (C[s]\Downarrow \Rightarrow C[t]\Downarrow) \text{ und} \\
(C[t]\Uparrow \Rightarrow C[s']\Uparrow) \wedge (C[s']\Uparrow \Rightarrow C[s]\Uparrow) \quad &\Rightarrow \quad (C[t]\Uparrow \Rightarrow C[s']\Uparrow)
\end{aligned}$$

(Kompatibilität)
C sei ein beliebiger Kontext. Für alle Kontexte D existiert ein Kontext D', so daß $D[C[\cdot]] \equiv D'[\cdot]$ gilt. Für jeden Kontext D' gilt gemäß Voraussetzung $(D'[s]\Downarrow \Rightarrow D'[t]\Downarrow)$ und $(D'[t]\Uparrow \Rightarrow D'[s]\Uparrow)$. Daraus folgt, daß für alle Kontexte D gilt: $(D[C[s]]\Downarrow \Rightarrow D[C[t]]\Downarrow)$ und $(D[C[t]]\Uparrow \Rightarrow D[C[s]]\Uparrow)$. □

Korollar 3.1.1. $\sim_c$ *ist eine Kongruenz-Relation.*

Die untenstehende Proposition erlaubt eine wesentliche Schlußfolgerung in vielen der später vorkommenden Beweise der kontextuellen Äquivalenz.

Proposition 3.1.2. $\xrightarrow{r}$ *sei eine kompatible Reduktion auf* Λ_{let}*. Wenn für alle* $s, t \in \Lambda_{let}$ *mit* $s \xrightarrow{r} t$

$$s\Downarrow \Leftrightarrow t\Downarrow \text{ und } s\Uparrow \Leftrightarrow t\Uparrow$$

dann gilt für alle Kontexte C:

$$C[s]\Downarrow \Leftrightarrow C[t]\Downarrow \text{ und } C[s] \;\; \Leftrightarrow C[t]\Uparrow$$

Beweis. Offensichtlich, da aus der Kompatibilität direkt $C[s] \xrightarrow{r} C[t]$ folgt. □

Bevor wir zum Kontextlemma übergehen, soll in dem folgenden Abschnitt eine später wiederholt benutzte Beweismethodik kurz skizziert werden.

3.1.1 Zum Beweis der kontextuellen Äquivalenz von Reduktionen verwendete Methodik

Wir werden im weiteren Verlauf dieses Kapitels für eine Vielzahl von Reduktionen zeigen, daß ihre Anwendung zu zwei kontextuell äquivalenten Ausdrücken führt. Die Beweise gehorchen dabei alle einem ähnlichem Schema, das daher in dieser Stelle kurz vorgestellt werden soll.
Sei *red* eine kompatible Reduktion, für die wir die kontextuelle Äquivalenz zeigen wollen. Wir werden im allgemeinen folgenden Beweisablauf verwenden:

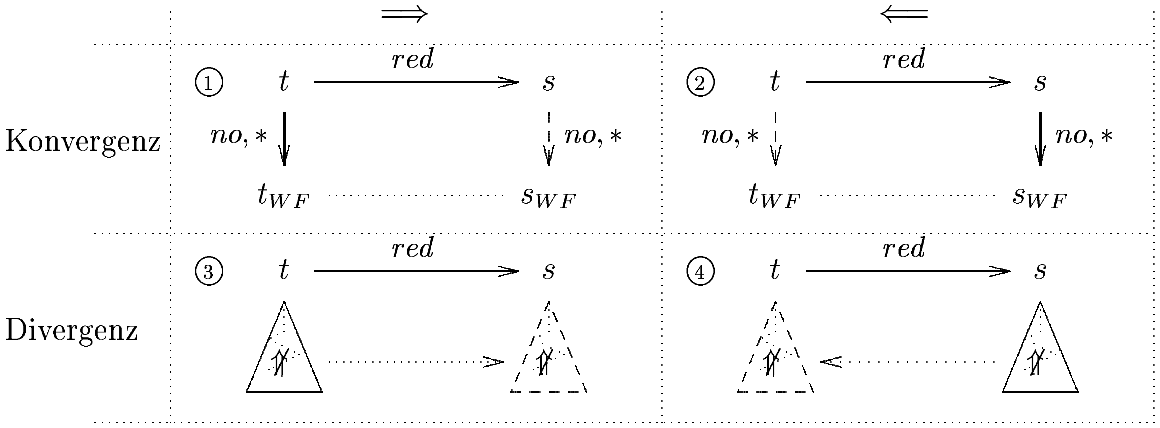


Abbildung 3.1:

1. Ein vollständiger Satz von Gabel- und Vertauschungsdiagrammen für die Reduktion *red* wird entwickelt.

2. Die wechselseitige Erhaltung einer WHNF wird gezeigt, d.h. wenn $s \xrightarrow{red} t$ dann gilt s in WHNF gdw. t in WHNF.

3. Mittels einer induktiven Argumentation unter Zuhilfenahme der zuvor gezeigten Gabel- und Vertauschungsdiagramme sowie der wechselseitigen Erhaltung einer WHNF wird die wechselseitige Übertragbarkeit einer no-Reduktion zu WHNF sowie die wechselseitige Übertragbarkeit der Endlichkeit aller no-Reduktionen gezeigt. Formal entspricht dies den folgenden 4 Aussagen:
   Gegeben zwei Ausdrücke s, t mit $s \xrightarrow{red} t$

   ① $t{\Downarrow} \Longrightarrow s{\Downarrow}$
   ② $s{\Downarrow} \Longrightarrow t{\Downarrow}$
   ③ $t{\not\Uparrow} \Longrightarrow s{\not\Uparrow}$
   ④ $s{\not\Uparrow} \Longrightarrow t{\not\Uparrow}$

   Abbildung 3.1 zeigt alle 4 Aussagen nocheinmal in Form eines Reduktionsdiagramms. Durch logische Umkehrung bzw. Widerspruch folgt aus ③ und ④ sofort

$$s{\Uparrow} \Longleftrightarrow t{\Uparrow}$$

4. Aus den unter 3. bewiesenen Aussagen folgt gemäß Proposition 3.1.2 die zu zeigende Erhaltung der kontextuellen Äquivalenz durch die Reduktion *red*.

Von dem oben vorgestellten Schema wird meistens leicht abgewichen, so ermöglicht beispielsweise die Verwendung des im folgenden Abschnitt vorgestellten Kontextlemmas bei manchen Reduktionen eine signifikante Vereinfachung der bei 3. erforderlichen induktiven Argumentation. Ferner ist es teilweise sinnvoller die Divergenz unter Zuhilfenahme von Königs Lemma zu zeigen, ein Beispiel hierfür gibt die später in diesem Kapitel vorgestellte $ldel$-Reduktion.

3.2 Das Kontextlemma

Eine Analyse des no-Markierungsalgorithmus zeigt, daß sich der no-Redex sowie das Kopierziel einer (no, cp)-Reduktion niemals innerhalb einer Abstraktion befinden. Diese Sachverhalte erlauben den Beweis des in diesem Abschnitt vorgestellten Kontextlemmas. Das Kontextlemma bildet ein leistungsfähiges Beweiswerkzeug:

- Bei Beweisen der kontextuellen Äquivalenz läßt sich durch die Verwendung des Kontextlemmas die lästige Vervielfachungseigenschaft anderer Reduktionen durch eine (no, cp)-Reduktionen ausblenden, was vereinfachte Induktionsbeweise ermöglicht. Eine solche Verwendung des Kontextlemmas erfolgt in Abschnitt 3.3.
- Durch das Kontextlemma lassen sich Terminierungsaussagen, die nur für W-Kontexte bewiesen wurden, auf beliebige Kontexte ausdehnen. Diese Technik wird in Abschnitt 5.1 verwendet, wo für Ausdrücke ohne Reduktion zu WHNF gezeigt wird, daß sie die kleinste Ausdrucksklasse bezüglich $\leq_c$ bilden.

Zunächst werden wir eine erweiterte Notation für Reduktionen kennenlernen, die wir im weiteren Verlauf benötigen werden.

Notation 3.2.1. *S sei ein Symbol wie R oder W, das zur Spezifikation eines Kontexts mit festgelegter Struktur benutzt wird, und s, t seien zwei beliebige Ausdrücke. Wir schreiben $s \xrightarrow{S[\cdot]} t$ um zu spezifizieren, daß sich der reduzierte Redex in einem S-Kontext befindet.*

In Übereinstimmung mit der obigen Notation treffen wir folgende Definition:

Definition 3.2.1. *S sei ein Symbol wie R oder W, das zur Spezifikation eines Kontexts mit festgelegter Struktur benutzt wird. a bezeichne eine Reduktionsvorschrift wie $llet$ oder $lbeta$. Wir treffen folgende Definition:*

Die $(a, S[\cdot])$-Reduktion umfaßt alle a-Reduktionen der Form $t \xrightarrow{a} s$, bei denen der reduzierte Redex in t in einem S-Kontext vorkommt.

Obige Definition wird es uns ermöglichen, Ausschnitte von bestehenden Reduktionen zu untersuchen, die durch die Kontexteinschränkung über besondere Eigenschaften verfügen. So möchten wir beispielsweise verhindern, daß sich der Redex innerhalb einer Abstraktion befinden kann, da dann eine Verdoppelung des Redex durch eine *cp*-Reduktion vermieden wird.
Zum Beweis des Kontextlemmas werden wir das nun folgende Lemma benötigen.

Lemma 3.2.1. *s, t seien zwei Λ_{let}-Ausdrücke. Wenn für alle W-Kontexte*

$$W[s]\Downarrow \;\Rightarrow\; W[t]\Downarrow \quad\wedge\quad W[t]\Uparrow \;\Rightarrow\; W[s]\Uparrow$$

dann gilt für alle Multikontexte C

$$C[s_1,\ldots,s_n]\Downarrow \;\Rightarrow\; C[t_1,\ldots,t_n]\Downarrow \quad\wedge\quad C[t_1,\ldots,t_n]\Uparrow \;\Rightarrow\; C[s_1,\ldots,s_n]\Uparrow$$

wobei für alle $\forall i \in \{1,\ldots,n\} s_i$ und t_i umbenannte Kopien von s bzw. t sind und $C[s_1,\ldots,s_n]$ wie $C[t_1,\ldots,t_n]$ geschlossene Ausdrücke sind.

Beweis. Konvergenz und Divergenz werden getrennt gezeigt.

Konvergenz
Wir zeigen $C[s_1,\ldots,s_n]\Downarrow \;\Rightarrow\; C[t_1,\ldots,t_n]\Downarrow$ durch Induktion über die Menge aller Ausdrücke $C[s_1,\ldots,s_n]$ mit einer Reduktion zu WHNF, die wir mittels des folgenden aus zwei lexikographisch geordneten Komponenten bestehenden Bewertungsmaßes ordnen:

1. Der erste Komponente ist die Anzahl einzelner no-Reduktionen, über die die kürzeste no-Reduktion ausgehend von $C[s_1,\ldots,s_n]$ zu WHNF verfügt.
2. Die zweite Komponente ist die Anzahl der Löcher im Kontext C.

Eines der Löcher von C entspreche einem W-Kontext, o.b.d.A. sei dies das erste Loch von C. Verfügt C nur über ein einzelnes Loch, dann folgt $C[t_1]\Downarrow$ direkt aus den Voraussetzungen. Besitzt C mehrere Löcher dann ist $C[\cdot, t_2,\ldots,t_n]$ ein W-Kontext und es gilt gemäß Voraussetzung $C[s_1,t_2,\ldots,t_n]\Downarrow \Rightarrow C[t_1,t_2,\ldots,t_n]\Downarrow$. Wir wählen $C' \equiv C[s_1,\cdot,\ldots,\cdot]$. $C[s_1,\ldots,s_n]$ und $C'[s_2,\ldots,s_n]$ sind syntaktisch äquivalent jedoch ist C' kleiner als C, da C' über ein Loch weniger als C verfügt. Somit folgt gemäß Induktionsvoraussetzung $C'[s_2,\ldots,s_n]\Downarrow \Rightarrow C'[t_2,\ldots,t_n]\Downarrow$. Nun sind $C'[t_2,\ldots,t_n]$ und $C[s_1,t_2,\ldots,t_n]$ syntaktisch äquivalent, woraus direkt $C[t_1,t_2,\ldots,t_n]\Downarrow$ folgt.

Befindet sich keines der Löcher in einem W-Kontext und ist der Ausdruck $C[s_1,\ldots,s_n]$ nicht in WHNF, so wählen wir die kürzeste von $C[s_1,\ldots,s_n]$ ausgehende no-Reduktion. Da sich gemäß Satz 2.3.1 alle direkten Subausdrücke eines no-Redex sowie der no-Redex selbst stets in einem W-Kontext befinden und somit kein Ausdruck s_i direkt an dem no-Redex beteiligt sein kann, muß auf dem ausgewählten no-Pfad $C[s_1,\dots,s_n] \xrightarrow{no} C'[s'_1,\dots,s'_m]$ und $C[t_1,\dots,t_n] \xrightarrow{no} C'[t'_1,\dots,t'_m]$ gelten. Die Anzahl der Löcher kann in C' durch den Kopiervorgang einer (no, cp)-Reduktion vergrößert werden. Die no-Reduktion ausgehend von $C'[s'_1,\dots,s'_m]$ zu WHNF ist jedoch um einen Schritt verkürzt. Daher liefert die Induktionsvoraussetzung $C'[s'_1,\dots,s'_m]\Downarrow \;\Rightarrow\; C'[t'_1,\dots,t'_m]\Downarrow$ und in Kombination mit der zuvor aufgeführten no-Reduktion ergibt sich $C[t_1,\dots,t_n]\Downarrow$.

Zuletzt betrachten wir den Fall, daß sich keines der Löcher in einem W-Kontext befindet und $C[s_1,\dots,s_n]$ in WHNF ist. Gemäß Satz 2.3.1 hat jeder Ausdruck in WHNF die Form $L_R^*[r]$ mit r eine Abstraktion oder die Konstante `choice`. Kein Loch in C kann mit einem Teil des L_R^*-Kontext zusammenfallen, da jeder L_R-Kontext ein W-Kontext ist. So kann in $C[t_1,\dots,t_n]$ zwar der L_R^*-Kontext und die Abstraktion r variiert sein, jedoch ist $C[t_1,\dots,t_n]$ damit weiterhin in WHNF.

Divergenz
Wir zeigen $C[t_1,\dots,t_n]\Uparrow \;\Rightarrow\; C[s_1,\dots,s_n]\Uparrow$ nicht direkt sondern über den Umweg der logischen Umkehrung und beweisen:

> Wenn alle no-Reduktionen ausgehend von $C[s_1,\dots,s_n]$ zu einer WHNF führen, dann führen auch alle no-Reduktionen ausgehend von $C[t_1,\dots,t_n]$ zu einer WHNF.

Obige Hypothese zeigen wir durch Induktion über die Menge aller Ausdrücke $C[s_1,\dots,s_n]$, bei denen alle no-Reduktionen terminieren, die wir mittels des folgenden aus zwei lexikographisch geordneten Komponenten bestehenden Bewertungsmaßes ordnen:

1. Die erste Komponente ist die Anzahl einzelner no-Reduktionen, über die die längste no-Reduktion ausgehend von $C[s_1,\dots,s_n]$ verfügt.
2. Die zweite Komponente ist die Anzahl der Löcher im Kontext C.

Eines der Löcher von C entspreche einem W-Kontext. Besitzt C nur ein einzelnes Loch dann folgt die Terminierung aller no-Reduktionen ausgehend von $W[t_1]$ direkt aus einer logischen Umkehrung der Voraussetzung $W[t_1]\Uparrow \;\Rightarrow\; W[s_1]\Uparrow$. Besitzt C mehrere Löcher, so entspreche o.b.d.A. das erste Loch einem W-Kontext. Dann ist $C[\cdot, t_2,\dots,t_n]$ ein W-Kontext und es gilt gemäß Voraussetzung $C[t_1,t_2,\dots,t_n]\Uparrow \;\Rightarrow\; C[s_1,t_2,\dots,t_n]\Uparrow$. Nun zeigen wir, daß die Annahme $C[t_1,t_2,\dots,t_n]\Uparrow$ in Kombination mit der Voraussetzung, daß alle no-Reduktionen ausgehend von $C[s_1,\dots,s_n]$ terminieren einen Widerspruch bildet. Wir wählen $C' \equiv C[s_1,\cdot,\dots,\cdot]$. Gemäß Induktionsvoraussetzung gilt, daß alle no-Reduktionen ausgehend von $C'[t_2,\dots,t_n]$ terminieren, da $C'[s_2,\dots,s_n]$ und

$C[s_1, s_2, \ldots, s_n]$ syntaktisch äquivalent sind. Nun gilt jedoch $C'[t_2, \ldots, t_n] \equiv C[s_1, t_2, \ldots, t_n]$ und die Annahme $C[t_1, t_2, \ldots, t_n]\Uparrow$ ergibt in Kombination mit der zuerst aufgeführten Implikation einen Widerspruch.

Wir betrachten nun den Fall, daß sich keines der Löcher in einem W-Kontext befindet und $C[s_1, \ldots, s_n]$ nicht in WHNF ist. Von $C[s_1, \ldots, s_n]$ können, je nachdem ob der no-Redex ein (no, nd)-Redex ist oder nicht, mehrere no-Reduktionen ausgehen. Kein Ausdruck s_i kann direkt an dem no-Redex beteiligt sein, da sich gemäß Satz 2.3.1 alle direkten Subausdrücke eines no-Redex sowie der no-Redex selbst stets in einem W-Kontext befinden. Somit gilt für alle Alternativen $C[s_1, \ldots, s_n] \xrightarrow{no} C'[s'_1, \ldots, s'_m] \;\Rightarrow\; C[t_1, \ldots, t_n] \xrightarrow{no} C'[t'_1, \ldots, t'_m]$, wobei die Anzahl der Löcher in C' größer sein kein als in C. Da die Anzahl der no-Schritte der längsten no-Reduktion zu WHNF stets um einen Schritt vermindert wird, läßt sich bei allen Alternativen die Induktionsvoraussetzung anwenden, die besagt, daß alle no-Reduktionen ausgehend von $C'[t'_1, \ldots, t'_m]$ zu einer WHNF führen. Daraus folgt insgesamt, daß alle no-Reduktionen ausgehend von $C[t_1, \ldots, t_n]$ zu einer WHNF führen.

Bleibt als letzter Fall, daß sich keines der Löcher in einem Reduktionskontext befindet und $C[s_1, \ldots, s_n]$ in WHNF ist. Hier kann direkt die Argumentation desselben Falls beim Beweis der Konvergenz übernommen werden, die besagt, daß dann auch $C[t_1, \ldots, t_n]$ in WHNF ist. □

Bei der logischen Umkehr beim Beweis der Divergenz ist wesentlich, daß die Ausdrücke $C[s_1, \ldots, s_n]$ und $C[t_1, \ldots, t_n]$ geschlossen sein müssen und in der Folge eine no-Reduktion zu VWHNF nicht möglich ist. Wir sind nun in der Lage ein Kontextlemma zu formulieren:

Lemma 3.2.2. *(Kontextlemma) s, t seien zwei Λ_{let}-Ausdrücke.*

1. $s \leq_W t \iff s \leq_c t, \qquad s \sim_W t \iff s \sim_c t$
2. $s \leq_R t \iff s \leq_c t, \qquad s \sim_R t \iff s \sim_c t$

Beweis. Alle gdw.-Beziehungen werden aus dem zuletzt bewiesenen Lemma in Kombination mit den folgenden einfachen Überlegungen erhalten:

1. Die Klasse der Kontexte mit nur einem Loch ist eine Teilmenge der Kontexte mit beliebig vielen Löcher.
2. $s \leq_W t$ und $t \leq_W s \;\Rightarrow s \leq_c t$ und $t \leq_c s$. (Gemäß Lemma 3.2.1)
3. Jeder W-Kontext ist zugleich ein R-Kontext.
4. $s \leq_c t \;\Rightarrow\; s \leq_{C'} t$ für jede beliebige Kontextklasse C'. □

Das nun folgende Lemma bildet eine später im Zusammenhang mit Beweisen der Erhaltung der kontextuellen Äquivalenz häufig benötigte Ergänzung des zuvor vorgestellten Kontextlemmas.

Lemma 3.2.3. *Gegeben eine kompatible Reduktion a, bei der der Begriff des Redex anwendbar ist und bei der für alle Tupel geschlossener Ausdrücke* (t, s) *mit* $t \xrightarrow{a,R[\cdot]} s$ *die Gültigkeit von*

$$t\Downarrow \iff s\Downarrow \quad \textit{und} \quad t\Uparrow \iff s\Uparrow$$

gezeigt wurde.
Dann gilt für beliebige Ausdrücke t', s' *mit* $t' \xrightarrow{a} s'$ *die Eigenschaft* $t' \sim_c s'$.

Beweis. Gegeben zwei geschlossene Ausdrücke t, s mit $t \xrightarrow{a,R[\cdot]} s$. Die Reduktion a ist per Voraussetzung kompatibel. Daraus folgt $R[t] \xrightarrow{a} R[s]$ für jeden R-Kontext. Da die Schachtelung $R[R'[\cdot]]$ zweier R-Kontexte wiederum einen R-Kontext liefert, gilt in Erweiterung auch $R[t] \xrightarrow{a,R[\cdot]} R[s]$ für jeden R-Kontext. In Kombination mit den Voraussetzungen über das Konvergenz- und Divergenzverhalten der $(a, R[\cdot])$-Reduktion folgt daraus, daß für alle Kontexte R, bei denen $R[t]$ und $R[s]$ geschlossen ist, die beiden gdw.-Beziehungen

$$R[t]\Downarrow \iff R[s]\Downarrow \quad \text{und} \quad R[t]\Uparrow \iff R[s]\Uparrow$$

gelten. Letzteres ist gleichbedeutend mit $s \sim_R t$. Gemäß dem zuvor gezeigten Kontextlemma 3.2.2 folgt daraus $s \sim_c t$.
Damit haben wir die Erhaltung der kontextuellen Äquivalenz durch alle $(a, R[\cdot])$-Reduktionen gezeigt. Wir gehen nun zu einer beliebigen a-Reduktion über.

Gegeben zwei Ausdrücke t, s mit $t \xrightarrow{a} s$. Wir bilden die Zerlegungen $t \equiv C[t']$ und $s \equiv C[s']$, so daß t' bei der gegebenen a-Reduktion von t zu s den Redex bildet. (Das Loch $[\cdot]$ des Kontexts C kann sich dabei durch aus innerhalb einer Abstraktion befinden.) Da der leere Kontext $[\cdot]$ ein R-Kontext ist, gilt $t' \xrightarrow{a,R[\cdot]} s'$. Im ersten Teil dieses Lemmas haben wir gezeigt, daß daraus in Kombination mit den Voraussetzungen die Eigenschaft $t' \sim_c s'$ folgt. Da $\sim_c$ eine Kongruenzrelation ist folgt aus letzterem wiederum $C[t'] \sim_c C[s']$, was $t \sim_c s$ entspricht. □

3.3 kontextuelle Äquivalenz von *llet*, *lapp*, *lbeta*, *cp*

Wir werden nun die Erhaltung der kontextuellen Äquivalenz durch die vier Basisreduktionen *llet*, *lapp*, *lbeta* und *cp* des Λ_{let}-Kalküls zeigen. Dazu wird zunächst die Erhaltung der kontextuellen Äquivalenz für alle internen Reduktionen der 4 Reduktionsarten gezeigt. Wir werden in diesem Abschnitt das Kontextlemma verwenden, daher werden wir unsere Betrachtungen zunächst auf die Klasse der R-Kontexte beschränken.

Lemma 3.3.1. *Der vollständige Satz von Gabeldiagrammen der $(i, a, R[\cdot])$-Reduktion mit $a \in \{llet, lapp, lbeta, cp\}$ lautet:*

i): $\xleftarrow{no,a,0/1} \circ \xleftarrow{no,b} \circ \xrightarrow{i,a,R[\cdot]} \rightsquigarrow \xrightarrow{i,a,R[\cdot],0/1} \circ \xleftarrow{no,b}$

für $a \in \{llet, lapp, lbeta\}$, $b \in \{llet, lapp, lbeta, cp, nd\}$

iii): $\xleftarrow{no,a} \circ \xleftarrow{no,cp,*} \circ \xrightarrow{i,cppar,R[\cdot]} \rightsquigarrow \xrightarrow{i,cppar,R[\cdot],0/1} \circ \xleftarrow{no,a}$

für $a \in \{llet, lapp, lbeta, cp, nd\}$.

iv): $\xleftarrow{no,llet,1/2} \circ \xleftarrow{no,lapp,\omega} \circ \xrightarrow{i,llet,R[\cdot]}$

$\rightsquigarrow \xrightarrow{i,llet,R[\cdot],0/1} \circ \xleftarrow{no,llet} \circ \xleftarrow{no,lapp,\omega} \circ \xleftarrow{no,llet} \circ \xleftarrow{no,lapp,\omega}$ *für* $\omega \geq 0$

v): $\xleftarrow{no,llet,0/1} \circ \xleftarrow{no,lapp,\omega} \circ \xrightarrow{i,R[\cdot],llet} \rightsquigarrow \xrightarrow{i,llet,R[\cdot],0/1} \circ \xleftarrow{no,lapp,2*\omega}$ *für* $\omega \geq 1$

Beweis. Alle in Lemma 2.4.7 aufgeführten Vertauschungsdiagramme für $(i,llet)$, $(i, lapp)$, $(i, lbeta)$, (i, cp) können wie in Abschnitt 2.4.2 beschrieben in Gabeldiagramme überführt werden.
Alle Vertauschungsdiagramme, bei denen eine (no, cp)-Reduktion zur Verdoppelung des internen Redex führt können dabei fortgelassen werden, da sich dazu der interne Redex unterhalb einer Abstraktion befinden müßte, was durch die Einschränkung auf die Kontextklasse R ausgeschlossen wird. □

Die Einschränkung auf die Menge aller R-Kontexte ist eine Optimierung, die durch die spätere Verwendung des Kontextlemmas 3.2.2 ermöglicht wird. Das Ausklammern des Falles der Duplizierung interner Redizes durch eine (no, cp)-Reduktion vereinfacht den nun folgenden Beweis.

Lemma 3.3.2. *Gegeben zwei Ausdrücke t, s mit $t \xrightarrow{i,a,R[\cdot]} s$, wobei $a \in \{llet, lapp, lbeta, cp\}$, sowie ein Ausdruck t' mit $t \xrightarrow{no,*} t'$. Dann existieren zwei Ausdrücke t'' und s', so daß die Reduktionszusammenhänge des unten stehenden Diagramms erfüllt werden und zusätzlich $\#lbeta(s \xrightarrow{no,*} s') \geq \#lbeta(t \xrightarrow{no,*} t') - 1$ gilt, wobei $\#lbeta$ die Funktion sei, die die Anzahl einzelner lbeta-Reduktionen der Argumentreduktion wiedergibt:*

$$\begin{tikzcd}
t \arrow[r, "{i, a, R[\cdot]}"] \arrow[d, "{no, *}"] & s \arrow[dd, dashed, "{no, +}"] \\
t' \arrow[d, dashed, "{no, *}"] & \\
t'' \arrow[r, dashed, "{i, a, R[\cdot], 0/1}"] & s'
\end{tikzcd}$$

Diagramm 1

$$\begin{tikzcd}
t \arrow[r, two heads, "{i,a,R[\cdot]}"] \arrow[d, two heads, "{no,+}"'] \arrow[dr, phantom, "\text{Lemma 3.3.1}"] & s \arrow[d, dashed, two heads, "{no,+}"] \\
t_d \arrow[r, dashed, two heads, "{i,a,R[\cdot]}"] \arrow[r, two heads, shift right] \arrow[d, two heads, "{no,*}"'] & s_d \arrow[dd, dashed, two heads] \\
t' \arrow[r, phantom, "\text{Ind. Voraus.}"] \arrow[d, dashed, two heads, "{no,*}"'] & \\
t'' \arrow[r, dashed, two heads, "{i,a,R[\cdot],0/1}"'] & s'
\end{tikzcd}$$

Diagramm 2

$$\begin{tikzcd}
t \arrow[r, two heads, "{i,a,R[\cdot]}"] \arrow[d, two heads, "{no,+}"] \arrow[r, phantom, "\text{Lemma 3.3.1}"', shift right=2] & s \arrow[dl, dashed, two heads, "{no,+}"] \\
t_d \arrow[d, two heads, "{no,*}"] & \\
t' &
\end{tikzcd}$$

Diagramm 3

$$\begin{tikzcd}
t \arrow[r, two heads, "{i,a,R[\cdot]}"] \arrow[d, two heads, "{no,+}"'] & s \arrow[dd, dashed, two heads, "{no,*}"] \\
t' \arrow[r, phantom, "\text{Lemma 3.3.1}"] \arrow[d, dashed, two heads, "{no,+}"'] & \\
t'' \arrow[r, dashed, two heads, "{i,a,R[\cdot]}"'] & s'
\end{tikzcd}$$

Diagramm 4

$$\begin{tikzcd}
t \arrow[r, two heads, "{i,a,R[\cdot]}"] \arrow[d, two heads, "{no,+}"'] \arrow[r, phantom, "\text{Lemma 3.3.1}"', shift right=2] & s \arrow[ddl, dashed, two heads, "{no,*}"] \\
t' \arrow[d, dashed, two heads, "{no,+}"'] & \\
t'' \equiv s' &
\end{tikzcd}$$

Abbildung 3.2:

Beweis. Dies zeigt ein einfacher Induktionsbeweis über die einzelner no-Reduktionen von t zu t':

Induktionsbeginn:
Die no-Reduktion von t zu t' hat die Länge 0, d.h es gilt $t \equiv t'$, Wir wählen $s' \equiv s$ und haben die Hypothese für diesen Fall gezeigt.

Induktionsschritt:
Passend zu der von t ausgehenden no-Reduktion wählen wir eines der Gabeldiagramme aus Lemma 3.3.1, das wir schließen können. Nun sind zwei Fälle zu unterscheiden:

1. Fall: Die no-Reduktion von t zu t' ist verfügt über keine ausreichende Länge, um das ausgewählte Gabeldiagramm schließen zu können.
In diesem Fall wählen wir den Ausdruck t'' derart, daß das Gabeldiagramm geschlossen werden kann. Die Existenz von s' ergibt sich dann aus dem verwendeten Gabeldiagramm. Die zwei möglichen entstehenden Reduktionszusammenhänge zeigen die Diagramme3 und 4 aus Abbildung 3.2.

2. Fall: Die no-Reduktion von t zu t' ist verfügt über ausreichend viele no-Reduktionen, um das ausgewählte Gabeldiagramm schließen zu können.
Wir zerlegen die von t ausgehende no-Reduktion entsprechend des benutzen Gabeldiagramms in $t \xrightarrow{no,+} t_d \xrightarrow{no,*} t'$ und differenzieren in einer zweiten Ebene danach, ob das Gabeldiagramm auf der rechten Seite eine $(i, a, R[\cdot])$-Reduktion aufweist oder nicht:
Weist das Gabeldiagramm eine $(i, a, R[\cdot])$-Reduktion auf, zeigt Diagramm 1 aus Abbildung 3.2 die entstehenden Reduktionszusammenhänge. Die von t_d ausgehende no-Reduktion ist kürzer als die von t ausgehende, somit können wir bei t_d die Induktionsvoraussetzung anwenden und haben die Hypothese gezeigt. Im

anderen Fall zeigt Diagramm 2 aus Abbildung 3.2 die entstehenden Reduktionszusammenhänge. Die Gültigkeit der Hypothese ist in diesem Fall offensichtlich.

Bleibt das Erhaltungsgesetz für die $(no, lbeta)$-Reduktionen zu zeigen. Eine Betrachtung der Gabeldiagramme aus Lemma 3.3.1 zeigt:

- Kein Gabeldiagramm für die Reduktionen $(i, llet)$, $(i, lapp)$, (i, cp) verändert die Anzahl der $(no, lbeta)$-Reduktionen. Daraus folgt in all diesen Fällen direkt die Erhaltung der $(no, lbeta)$-Reduktionen.
- Eine $(i, lbeta)$-Reduktion kann zum Wegfall einer $(no, lbeta)$-Reduktion führen. Dies kann jedoch nur einmal vorkommen, da dann die interne Reduktion verschwindet; dies ist der in Diagramm 2 aus Abbildung 3.2 gezeigte Fall. Ansonsten bleibt eine $(no, lbeta)$-Reduktion auch hier stets erhalten, woraus direkt die Gültigkeit des Erhaltungsgesetzes auch bei diesem Fall folgt.

Die Ungleichung faßt die beiden somit möglichen Fälle $\#lbeta(s \longrightarrow s') = \#lbeta(t \longrightarrow t')$ und $\#lbeta(s \longrightarrow s') = \#lbeta(t \longrightarrow t') - 1$ zusammen. □

Mit dem nun folgenden Lemma schaffen wir die Grundlage, um in einem dann folgenden Schritt die Erhaltung der kontextuellen Äquivalenz durch eine interne Reduktion vom Typ *llet*, *lapp*, *lbeta* oder *cp* zeigen zu können.

Lemma 3.3.3. *t, s seien zwei geschlossene Ausdrücke. Wenn $t \xrightarrow{i,a,R[\cdot]} s$ mit $a \in \{llet, lapp, lbeta, cp\}$, dann*

$$t\Downarrow \Leftrightarrow s\Downarrow \text{ und } t\Uparrow \Leftrightarrow s\Uparrow$$

Beweis. Wir zeigen $t\Downarrow \Leftrightarrow s\Downarrow$.:
Die Richtung "⇒" folgt direkt aus der Kombination der in Lemma 3.3.2 gezeigten Übertragbarkeit einer no-Reduktion und der in 2.3.4 gezeigten Erhaltung einer WHNF durch interne Reduktionen. Die Umkehrung "⇐" wird bereits in Lemma 2.4.8 gezeigt, dort sogar für Reduktionen innerhalb beliebiger Kontexte.

Wir zeigen $t\Uparrow \Leftrightarrow s\Uparrow$:
Wir werden stellvertretend nur die Richtung ⇒ zeigen. Die Umkehrung kann nach derselben Methodik auf der Grundlage von Lemma 2.4.8 gezeigt werden.

Gemäß Korollar 2.4.3 enthält jede unendliche Reduktion unendlich viele *lbeta*-Reduktionen. Existiert ausgehend von t eine unendliche no-Reduktion, so folgt daraus in Kombination mit Lemma 3.3.2, daß wir für jede Zahl $k \geq 0$ eine no-Reduktion ausgehend von s angeben können, die über mindestens k $(no, lbeta)$-Reduktionen verfügt. Jeder Ausdruck kann maximal zwei Nachfahren besitzen, die in einem no-Reduktionsschritt erreichbar sind. Aus der Anwendung des Königs-Lemma folgt damit, daß ausgehend von s eine unendliche no-Reduktion existiert. □

Wir zeigen nun für eine beliebige interne Reduktion vom Typ *llet*, *lapp*, *lbeta* oder *cp* die Erhaltung der kontextuellen Äquivalenz.

Satz 3.3.1. *s, t seien zwei geschlossene Ausdrücke. Wenn $t \xrightarrow{i,a} s$ mit $a \in \{$llet, lapp, lbeta, cp$\}$, dann $t \sim_c s$.*

Beweis. Die betrachtete (i, a)-Reduktion ist kompatibel und der Redex-Begriff anwendbar. Damit folgt aus Lemma 3.3.3 in Kombination mit Lemma 3.2.3 direkt die Hypothese. ☐

Für (i, nd) lassen sich aus den in Abschnitt 2.4.1 aufgeführten Gründen keine Gabeldiagramme angeben. Auf der Basis der dort vorgestellten Überlegungen läßt sich schnell ein Beispiel angeben, bei dem eine (i, nd)-Reduktion die kontextuelle Äquivalenz nicht erhält:

Beispiel 3.3.1. Gegeben seien zwei Ausdrücke s, t mit $s\not\Uparrow$ und $t\not\Downarrow$ sowie der Ausdruck $t_e \equiv \texttt{choice}\ \ s\,t$. Es ist offensichtlich, daß ausgehend von t_e eine terminierende wie eine nichtterminierende no-Reduktion existiert, je nachdem ob $(no, nd, left)$- eine oder $(no, nd, right)$-Reduktion vorkommt. O.b.d.A. sei die $(nd, right)$-Reduktion intern und wir erhalten

$$t_e \begin{cases} \xrightarrow{(no,nd,left)} & t_{left} \equiv (\lambda x.\lambda y.x)\, s\, t \\ \xrightarrow{(i,nd,right)} & t_{right} \equiv (\lambda x.\lambda y.y)\, s\, t \end{cases}$$

Ausgehend von t_{right} existiert keine terminierende no-Reduktion, da eine Projektion auf t erfolgt. Somit besitzen die beiden Ausdrücke t_e und t_{right} für den leeren Kontext $[\cdot]$ ein unterschiedliches Terminierungsverhalten, woraus folgt, daß sie nicht kontextuell äquivalent sind.

Zum Abschluß zeigen wir die Erhaltung der kontextuellen Äquivalenz durch eine beliebige *llet*-, *lapp*-, *lbeta*- oder *cp*-Reduktion, egal ob eine interne oder no-Reduktion.

Satz 3.3.2. *s, t seien geschlossene zwei Ausdrücke. Wenn $t \xrightarrow{a} s$ mit $a \in \{$llet, lapp, lbeta, cp$\}$, dann $t \sim_c s$.*

Beweis. Die Reduktion von t nach s ist entweder eine no-Reduktion oder eine interne Reduktion. Für $t \xrightarrow{i,a} s$ haben wir die kontextuelle Äquivalenz in Satz 3.3.1 gezeigt. Bei $t \xrightarrow{no,a} s$ ist die kontextuelle Äquivalenz von t und s offensichtlich, da die no-Reduktion bei allen vier Reduktionstypen eindeutig ist. ☐

3.4 Vier weitere Reduktionsregeln

Bis jetzt haben wir in diesem Kapitel den Begriff der kontextuellen Äquivalenz kennengelernt, ein Kontextlemma formuliert und für vier Reduktionen des Λ_{let}-Kalküls gezeigt, daß sie die kontextuelle Äquivalenz erhalten. Wir werden nun als nächsten Schritt 4 neue Reduktionen einführen und für diese die Erhaltung der kontextuellen Äquivalenz zeigen. Die Einführung der neuen Reduktionsregeln geschieht vor dem Hintergrund, daß wir für komplexere Transformationen auf Ausdrucksebene wie beispielsweise Lambda-Lifting zeigen wollen, daß diese bezüglich der kontextuellen Äquivalenz korrekt sind.

3.4.1 Vorstellung der neuen Reduktionsregeln

Die vier neu hinzukommenden Regeln tragen die Bezeichnung $ldel, lcv, lcom$ sowie *ucp* und ihre operationale Semantik und verfügen über die folgende Definition:

Definition 3.4.1. *Wir treffen folgende Definitionen für Reduktionen:*

$(ldel)$ $C[(\texttt{let}\ x = s\ \texttt{in}\ t)] \xrightarrow{ldel} C[t]$ *falls x nicht in t vorkommt*

(lcv) $C[(\texttt{let}\ x = y\ \texttt{in}\ D[x])] \xrightarrow{lcv} C[(\texttt{let}\ x = y\ \texttt{in}\ D[y])]$
wobei x und y Variablen

$(lcom)$ $C[\texttt{let}\ x = t_x\ \texttt{in}\ (\texttt{let}\ y = t_y\ \texttt{in}\ s)] \xrightarrow{lcom} C[\texttt{let}\ y = t_y\ \texttt{in}\ (\texttt{let}\ x = t_x\ \texttt{in}\ s)]$
falls x nicht in t_y vorkommt

(ucp) $C[\texttt{let}\ x = t_x\ \texttt{in}\ R[x]] \xrightarrow{ucp} C[\texttt{let}\ x = t_x\ \texttt{in}\ R[t_x]]$
falls die Variable x im Ausdruck $R[x]$ genau einmal vorkommt und

Alle Regeln sollen nun kurz vorgestellt werden:
Eine *ldel*-Reduktion ermöglicht das Löschen bindungsloser Subausdrücke bei Let-Ausdrücken. Bindungslose Subausdrücke können beispielsweise durch wiederholtes Kopieren entstehen, da mit jedem Kopiervorgang der Verlust eines Kopierziels verbunden ist. Bezogen auf eine maschinelle Sichtweise von Reduktion entspricht ihr Effekt einer garbage-collection.
Mittels einer *lcv*-Reduktion ist das Kopieren einer einzelnen Variable möglich. Wird jede Variablen mit einem Zeiger assoziiert, so entspricht *lcv* der Elimination von Indirektionen. Mittels *lcom* können zwei aufeinanderfolgende Let-Bindungen getauscht werden, sofern dies die Abhängigkeiten der Variablen zulassen. *lcom*-Reduktionen werden zum Beweis der Korrektheit der *ucp*-Reduktionsregel benötigt. Die *ucp*-Regel besagt, daß die Kopie eines Let-gebundenen Ausdrucks immer möglich ist, wenn genau ein Kopierziel existiert und dieses nicht unterhalb einer

Abstraktion vorkommt. Die *ucp*-Reduktionsregel verfügt über starke Nebenbedingungen und besitzt den schwierigsten Korrektheitsbeweis aller vier Regeln. Dies ist in der Komplexität der Gabel- und Vertauschungsdiagramme dieser Reduktion begründet. Zugleich ist *ucp* aber eine sehr leistungsfähige Reduktion, was ihre Bedeutung und Verwendung im Kapitel über die maschinellen Nachahmung der no-Reduktion unterstreicht. Die *ucp*-Reduktion kann sinnvoll mit einer *ldel*-Reduktion kombiniert werden, um einen durch eine *ucp*-Reduktion entstehenden bindungslosen Ausdruck zu löschen.

3.4.2 kontextuelle Äquivalenz der *ldel*-Reduktion

In diesem Abschnitt werden wir die Erhaltung der kontextuellen Äquivalenz durch eine *ldel*-Reduktion zeigen. Zu Beginn werden wir einen vollständigen Satz von Gabel- sowie Vertauschungsdiagrammen für die *ldel*-Reduktion vorstellen, wobei wir dabei die Verdoppelung eines *ldel*-Redex durch eine (no, cp)-Reduktion durch die Einschränkung auf die Klasse der R-Kontexte vermeiden werden. Danach wird die wechselseitige Übertragbarkeit einer no-Reduktion unter Erhaltung der Zahl an (no, cp)-Reduktionen sowie die wechselseitige Erhaltung einer WHNF durch eine *ldel*-Reduktion gezeigt. Unter Zuhilfenahme des Kontextlemmas wird daraus die Erhaltung der kontextuellen Äquivalenz gefolgert werden.

Lemma 3.4.1. *Die $(ldel, R[\cdot])$-Reduktion verfügt über den folgenden vollständigen Satz von Gabeldiagrammen:*

i) $\xleftarrow{no,a} \circ \xrightarrow{ldel,R[\cdot]} \;\rightsquigarrow\; \xrightarrow{ldel,R[\cdot]} \circ \xleftarrow{no,a}$ *für* $a \in \{llet, lapp, lbeta, cp, nd\}$

ii) $\xleftarrow{no,llet} \circ \xrightarrow{ldel,R[\cdot]} \;\rightsquigarrow\; \xrightarrow{ldel,R[\cdot]}$

iii) $\xleftarrow{no,lapp} \circ \xrightarrow{ldel,R[\cdot]} \;\rightsquigarrow\; \xrightarrow{ldel,R[\cdot]}$

Beweis. Wir zeigen, daß bei allen Positionskombinationen von no-Redex und *ldel*-Redex eines der obigen Gabeldiagramme erfüllt wird. Wir betrachten dazu jeden einzelnen Redextyp, den ein no-Redex annehmen kann, und benutzen die aus Satz 2.3.1 bekannten Informationen über die Struktur der Kontexte, in denen die jeweils betrachtete Art von no-Redizes vorkommt.

$(no, llet)$
Ein $(no, llet)$-Redex hat stets die Form $L_R^*[\texttt{let } x = (\texttt{let } y = t_y \texttt{ in } t_x) \texttt{ in } W[x]]$. Wir betrachten alle Positionsmöglichkeiten eines $(ldel, R[\cdot])$-Redex:

- Bei einer $(ldel, R[\cdot])$-Reduktion innerhalb von t_y oder t_x kann offensichtlich immer ein Gabeldiagramm der Art *i)* konstruiert werden.

- Eine $(ldel, R[\cdot])$-Reduktion innerhalb des L_R^*-Kontext, kann zum Wegfallen einer einzelnen let-Bindung innerhalb des L_R^*-Kontext führen, jedoch kann dadurch keine Veränderung bezüglich der Selektion des no-Redex eintreten, da ein Kontext ${L_R^*}'$ entsteht. Die Reduktion des no-Redex wiederum hat keine Auswirkungen auf den umgebenden L_R^*-Kontext, es kann somit immer ein Gabeldiagramm der Form *i)* konstruiert werden.

- Erfolgt die $(ldel, R[\cdot])$ -Reduktion innerhalb des Subausdrucks $W[x]$, so entsteht ein Subausdruck $W'[x]$. Die Variable x kann durch eine $(ldel, R[\cdot])$-Reduktion nicht gelöscht werden, da dem potentiell in W enthaltenen L_L-Subkontext stets eine Variablenvorkommen zugeordnet werden kann, das eine Bindung formuliert. Es kann somit immer ein Gabeldiagramm der Form *i)* konstruiert werden.

- Bleibt die Möglichkeit, daß die Variable y nicht in t_x vorkommt. In diesem Fall kann ein Gabeldiagramm der Form *iii)* konstruiert werden:

$$\begin{array}{ccc}
L_R^*[\texttt{let } x = (\texttt{let } y = t_y \texttt{ in } t_x) \texttt{ in } W[x]] & \xrightarrow{ldel,R[\cdot]} & L_R^*[\texttt{let } x = t_x \texttt{ in } W[x]] \\
\xrightarrow{no,llet} & & \equiv \\
L_R^*[\texttt{let } y = t_y \texttt{ in } (\texttt{let } x = t_x \texttt{ in } W[x])] & \xrightarrow{ldel,R[\cdot]} & L_R^*[\texttt{let } x = t_x \texttt{ in } W[x]]
\end{array}$$

$(no, lapp)$
Ein $(no, lapp)$-Redex hat stets die Form $L_R^*[L_L^{0/1}[A_L^*[(\texttt{let } x = t_x \texttt{ in } s)t]]]$. Kommt ein $(ldel, R[\cdot])$ -Redex innerhalb der Subausdrücke t_x, s, t oder innerhalb der Subkontexte L_R^* und A_L^* vor, ist dieselbe Argumentation wie bei den ersten beiden Fällen bei *llet* anzuwenden. Ein möglicherweise existierender L_L-Subkontext kann selbst kein $(ldel, R[\cdot])$-Redex bilden, da der L_L-Kontext nicht bindungslos sein kann. In allen zuletzt genannten Fällen kann ein Gabeldiagramm der Art *i)* konstruiert werden. Bleibt die Möglichkeit, daß die Variable x nicht in s vorkommt. In diesem Fall ist die Konstruktion eines Gabeldiagramms der Art *iv)* möglich:

$$\begin{array}{ccc}
L_R^*[L_L^{0/1}[A_L^*[(\texttt{let } x = t_x \texttt{ in } s)t]]] & \xrightarrow{ldel,R[\cdot]} & L_R^*[L_L^{0/1}[A_L^*[st]]] \\
\xrightarrow{no,lapp} & & \equiv \\
L_R^*[L_L^{0/1}[A_L^*[\texttt{let } x = t_x \texttt{ in } (st)]]] & \xrightarrow{ldel,R[\cdot]} & L_R^*[L_L^{0/1}[A_L^*[st]]]
\end{array}$$

(no, cp)
Ein (no, cp)-Redex hat stets die Form $L_R^*[(\texttt{let } x = t_x \texttt{ in } W[x]]$. Bei einem Vorkommen des $(ldel, R[\cdot])$-Redex innerhalb des Kontexts L_R^* oder innerhalb des Subausdrucks $W[x]$ kann die Argumentation vom ersten bzw. dritten Fall bei *llet* übernommen werden. Innerhalb des Subausdrucks t_x kann kein $(ldel, R[\cdot])$-Redex vorkommen, da sich dieser außerhalb eines R-Kontexts befinden würde.

$(no, lbeta)$
Ein $(no, lbeta)$-Redex hat stets die Form $L^*_R[L_L^{0/1}[A^*_L[(\lambda x.s)t]]]$. Bei allen drei Positionsmöglichkeiten des $(ldel, R[\cdot])$-Redex (innerhalb der Subausdrücke s bzw. t, innerhalb der Subkontexte L^*_R oder A^*_L, innerhalb des Subkontexts L_L) kann die Argumentation von *lapp* übernommen werden. Es kommen alleinig Gabeldiagramme der Form *i)* vor.

(no, nd)
Ein (no, nd)-Redex hat stets die Form $L^*_R[L_L^{0/1}[A^*_L[\texttt{choice}\ st]]]$. Es kann die Argumentation von *lbeta* übernommen werden, mit der Besonderheit, daß die Existenz eines Gabeldiagramms für beide Alternativen der $(no, \texttt{choice})$-Reduktion zu zeigen ist. □

Aufbauend auf dem zuvor vorgestellten vollständigen Satz von Gabeldiagrammen werden wir nun einen vollständigen Satz Vertauschungsdiagrammen vorstellen.

Lemma 3.4.2. *Die $(ldel, R[\cdot])$-Reduktion verfügt über den folgenden vollständigen Satz von Vertauschungsdiagrammen:*

i) $(ldel, R[\cdot]) \circ (no, a) \rightsquigarrow (no, a) \circ (ldel, R[\cdot])$

ii) $(ldel, R[\cdot]) \circ (no, a) \rightsquigarrow (no, llet) \circ (no, a) \circ (ldel, R[\cdot])$

iii) $(ldel, R[\cdot]) \circ (no, a) \rightsquigarrow (no, lapp)^+ \circ (no, llet)^{0/1} \circ (no, a) \circ (ldel, R[\cdot])$

Dabei gilt bei i) und iii) $a \in \{llet, lapp, lbeta, cp, nd\}$.

Beweis. Das Gabeldiagramm *i)*aus Lemma 3.4.1 kann direkt in ein Vertauschungsdiagramm überführt werden; dies ist das Vertauschungsdiagramme *i)*. Bei den Gabeldiagrammen *ii)* und *iii)* ist eine Überführung nicht möglich, da auf der rechten Seite der Gabeldiagramme kein no-Redex erscheint. Wir benutzen die Kenntnis über Art und Aussehen des no-Redex bei diesen Sonderfällen und entwickeln die zugehörigen Vertauschungsdiagramme:

1. $(no, llet)$ und y kommt nicht in t_x vor
   ($ldel$-Redex in einem Kontext der Form $L^*_R[L_L]$):

$$\begin{array}{ll}
 & L^*_R[\texttt{let}\ x = (\texttt{let}\ y = t_y\ \texttt{in}\ t_x)\ \texttt{in}\ t] \\
\xrightarrow{ldel,R[\cdot]} & L^*_R[\texttt{let}\ x = t_x\ \texttt{in}\ t] \\
\xrightarrow{no,a} & L^{*\prime}_R[t'] \\
 & \rightsquigarrow \\
 & L^*_R[\texttt{let}\ x = (\texttt{let}\ y = t_y\ \texttt{in}\ t_x)\ \texttt{in}\ t] \\
\xrightarrow{no,llet} & L^*_R[\texttt{let}\ y = t_y\ \texttt{in}\ (\texttt{let}\ x = t_x\ \texttt{in}\ t)] \\
\xrightarrow{no,a} & L^{*\prime}_R[\texttt{let}\ y = t_y\ \texttt{in}\ t'] \\
\xrightarrow{ldel,R[\cdot]} & L^{*\prime}_R[t']
\end{array}$$

2. $(no, lapp)$ und x kommt nicht in t_x vor und ein L_L Subkontext ist vorhanden ($ldel$-Redex in einem Kontext der Form $L^*_R[L_L[A^\omega_L[L_L]]]$):

$$\begin{array}{ll}
 & L^*_R[L_L[A^\omega_L[(\mathtt{let}\ x = t_x\ \mathtt{in}\ s)t]]] \\
\xrightarrow{ldel,R[\cdot]} & L^*_R[L_L[A^\omega_L[st]]] \\
\xrightarrow{no,a} & L^{*\prime}_R[L_L[A^\omega_L[t']]] \\
 & \leadsto \\
 & L^*_R[L_L[A^\omega_L[(\mathtt{let}\ x = t_x\ \mathtt{in}\ s)t]]] \\
\xrightarrow{no,lapp} & L^*_R[L_L[A^\omega_L[(\mathtt{let}\ x = t_x\ \mathtt{in}\ (st))]]] \\
\xrightarrow{no,lapp,\omega} & L^*_R[L_L[\mathtt{let}\ x = t_x\ \mathtt{in}\ A^\omega_L[st]]] \\
\xrightarrow{no,llet} & L^*_R[\mathtt{let}\ x = t_x\ \mathtt{in}\ L_L[A^\omega_L[st]]] \\
\xrightarrow{no,a} & L^{*\prime}_R[L_L[A^\omega_L[t']] \\
\xrightarrow{ldel,R[\cdot]} & L^{*\prime}_R[L_L[A^\omega_L[t']]]
\end{array}$$

3. $(no, lapp)$ und x kommt nicht in t_x vor und kein L_L Subkontext ist vorhanden:
($ldel$-Redex in einem Kontext der Form $L^*_R[A^\omega_L[L_L]]$):

$$\begin{array}{ll}
 & L^*_R[A^\omega_L[(\mathtt{let}\ x = t_x\ \mathtt{in}\ s)t]] \\
\xrightarrow{ldel,R[\cdot]} & L^*_R[A^\omega_L[st]] \\
\xrightarrow{no,a} & L^{*\prime}_R[A^\omega_L[t']] \\
 & \leadsto \\
 & L^*_R[A^\omega_L[(\mathtt{let}\ x = t_x\ \mathtt{in}\ s)t]] \\
\xrightarrow{no,lapp} & L^*_R[A^\omega_L[(\mathtt{let}\ x = t_x\ \mathtt{in}\ (st))]] \\
\xrightarrow{no,lapp,\omega} & L^*_R[\mathtt{let}\ x = t_x\ \mathtt{in}\ A^\omega_L[st]] \\
\xrightarrow{no,a} & L^{*\prime}_R[\mathtt{let}\ x = t_x\ \mathtt{in}\ A^\omega_L[t']] \\
\xrightarrow{ldel,R[\cdot]} & L^{*\prime}_R[A^\omega_L[t']]
\end{array}$$

□

Bei den oben betrachteten Sonderfällen ist zusätzlich die Form des Kontext angegeben, in dem der $ldel$-Redex vorkommt. Dies soll die Überprüfung der später bei der Untersuchung der ucp-Reduktion getroffenen Aussagen über die Gabel- und Vertauschungsdiagramme der $(ldel, L^*_R[\cdot])$-Reduktion erleichtern.
Nach den Gabel- und Vertauschungsdiagrammen wird nun die wechselseitige Erhaltung einer WHNF durch eine $ldel$-Reduktion gezeigt. Wir zeigen die Aussage dabei für eine allgemeine $ldel$-Reduktion, d.h. auch für eine $ldel$-Reduktion, bei der sich der Redex außerhalb eines R-Kontext befindet.

Lemma 3.4.3. *Gegeben zwei geschlossene Ausdrücke t und s mit $t \xrightarrow{ldel} s$. Dann gilt t in WHNF gdw. s in WHNF.*

Beweis.
"$\Rightarrow$"
Gemäß Lemma 2.3.1 ist ein Ausdruck in WHNF gdw. er die Form $L_R^*[r]$ besitzt, mit r eine Abstraktion oder die Konstante `choice` . Durch die Anwendung einer *ldel*-Reduktion kann entweder der Kontext L_R^* in einen Kontext $L_R^{*\prime}$ überführt werden, oder, die Abstraktion r in eine Abstraktion r'. Ein Ausdruck $L_R^{*\prime}[r']$ ist wegen gdw.-Beziehung wiederum in WHNF.

"$\Leftarrow$"
Es reicht die Umkehrung zu zeigen: Falls t einen no-Redex besitzt, dann besitzt auch s einen no-Redex. Die zu beweisende Aussage folgt dann durch eine einfache Widerspruchsüberlegung.

Wir benutzen den vollständigen Satz von Gabeldiagrammen aus Lemma 3.4.1. Diese erfassen zwar nur die $(ldel, R[\cdot])$-Reduktion, der Fall eines *ldel*-Redex unterhalb einer Abstraktion kann jedoch durch eine einfache Überlegung mit eingeschlossen werden.
Bei dem Gabeldiagramm *i)* ist es offensichtlich, daß s genauso wie t über einen no-Redex verfügt. Bei den Gabeldiagrammen *ii)* und *iii)* greifen wir auf die Aussage von Lemma 2.3.6 zurück, daß eine $(no, lapp)$- bzw. $(no, llet)$-Reduktion einen Ausdruck nicht in WHNF überführen kann und daher eine weitere no-Reduktion vom Typ *cp* oder *lbeta* existieren muß. Da bei einer no-Reduktion vom Typ *cp* oder *lbeta* nur Gabeldiagramm *i)* möglich ist, muß s also auch in diesem Fall einen no-Redex aufweisen. Bleibt der Fall eines *ldel*-Redex unterhalb einer Abstraktion. Eine einfache Überlegung zeigt, daß der *ldel*-Redex in diesem Fall zu keinem verschwinden des no-Redex führen kann, woraus folgt, daß s auch in diesem Fall über einen no-Redex verfügt. □

Lemma 3.4.4. *Gegeben zwei Ausdrücke t und s mit $t \xrightarrow{ldel,R[\cdot]} s$. Es gelten folgende beide Aussagen:*

1. *Gegeben ein weiterer Ausdruck t' mit $t \xrightarrow{no,*} t'$. Dann existiert ein Ausdruck s' mit $t' \xrightarrow{ldel,R[\cdot]} s'$ und $s \xrightarrow{no,*} s'$, so daß die Nebenbedingung $\#lbeta(t \rightarrow t') = \#lbeta(s \rightarrow s')$ erfüllt wird.*

2. *Gegeben ein weiterer Ausdruck s' mit $s \xrightarrow{no,*} s'$. Dann existiert ein Ausdruck t' mit $t' \xrightarrow{ldel,R[\cdot]} s'$ und $t \xrightarrow{no,*} t'$, so daß die Nebenbedingung $\#lbeta(t \rightarrow t') = \#lbeta(s \rightarrow s')$ erfüllt wird.*

Dabei sei #lbeta die Funktion, die die Anzahl einzelner lbeta-Reduktionen der Argumentreduktion liefert.

Diagramm 1

$$\begin{tikzcd}
t \arrow[r, "{ldel, R[\cdot]}"] \arrow[d, "no"'] \arrow[dr, phantom, "\begin{array}{c}\text{Lemma 3.4.1}\\ \text{①}\end{array}"] & s \arrow[d, dashed, "{no,*}"] \\
t_d \arrow[r, dashed, shift left, "{ldel, R[\cdot]}"] \arrow[r, shift right] \arrow[d, "{no,*}"'] \arrow[dr, phantom, "\begin{array}{c}\text{Ind. Voraus.}\\ \text{②}\end{array}"] & s_d \arrow[d, dashed, "{no,*}"] \\
t' \arrow[r, dashed, "{ldel, R[\cdot]}"'] & s'
\end{tikzcd}$$

Diagramm 2

$$\begin{tikzcd}
t \arrow[r, "{ldel, R[\cdot]}"] \arrow[d, dashed, "{no,+}"'] \arrow[dr, phantom, "\begin{array}{c}\text{Lemma 3.4.2}\\ \text{①}\end{array}"] & s \arrow[d, "no"] \\
t_d \arrow[r, dashed, shift left, "{ldel, R[\cdot]}"] \arrow[r, shift right] \arrow[d, dashed, "{no,*}"'] \arrow[dr, phantom, "\begin{array}{c}\text{Ind. Voraus.}\\ \text{②}\end{array}"] & s_d \arrow[d, "{no,*}"] \\
t' \arrow[r, dashed, "{ldel, R[\cdot]}"'] & s'
\end{tikzcd}$$

Abbildung 3.3:

Beweis. Es wird zunächst die erste Aussage gezeigt:
Wir führen eine Induktion über die Länge der no-Reduktion von t zu t' und benutzen als Induktionshypothese die Hypothese des Lemmas.

Induktionsbeginn:
Die no-Reduktion von t zu t' hat die Länge 0, d.h. $t \equiv t'$. In diesem Fall wählen wir s als s' und haben die Hypothese gezeigt.

Induktionsschritt:
Wir betrachten dazu die in Diagramm 1 aus Abbildung 3.3 gezeigten Reduktionszusammenhänge. Zunächst zerlegen wir die no-Reduktion von t zu t' in $t \xrightarrow{no} t_d \xrightarrow{no,*} t'$. Eines der Gabeldiagramme aus 3.4.1 muß wegen der Vollständigkeit des Diagrammsatzes zu der entstandenen Gabel passen und wir erhalten die Existenz des Ausdrucks s_d. Die no-Reduktion von t_d zu t' ist kürzer als die von t ausgehende, somit können wir auf t_d die Induktionsvoraussetzung anwenden und erhalten die Existenz von s'.

Bleibt zu zeigen, daß die Nebenbedingung bezüglich der Erhaltung der Anzahl an $(no, lbeta)$-Reduktion erfüllt wird. Dies zeigt ein Blick auf die Gabeldiagramme aus Lemma 3.4.1. Eine $(no, lbeta)$-Reduktion wird durch jedes Diagramm erhalten. Daraus folgt in Verbindung mit der oben vorgenommenen induktiven Argumentation direkt $\#lbeta(t \rightarrow t') = \#lbeta(s \rightarrow s')$.

Zum Beweis der zweiten Aussage wird ausgehend von dem Ausdruck s genauso vorgegangen wie bei der ersten Aussage ausgehend von t. Es wird eine Induktion über die Länge der no-Reduktion von s zu s' geführt und als Induktionshypothese die Hypothese des Lemmas benutzt. Statt der Gabeldiagramme werden die Vertauschungsdiagramme aus Lemma 3.4.2 benutzt. Der Beweis kann aus den in Diagramm 2 aus Abbildung 3.3 gezeigten Reduktionszusammenhängen abgelesen werden. Für die Erhaltung der Zahl an $(no, lbeta)$-Reduktionen kann dieselbe Argumentation wie bei der ersten Aussage angewendet werden. □

Obiges Lemma versetzt uns in die Lage, die wechselseitige Übertragbarkeit einer Reduktion zu WHNF sowie die wechselseitige Übertragbarkeit einer unendlichen zu zeigen.

Lemma 3.4.5. *Gegeben geschlossene zwei Ausdrücke t und s mit $t \xrightarrow{ldel,R[\cdot]} s$. Dann gilt $t\Downarrow \Leftrightarrow s\Downarrow$ und $t\Uparrow \Leftrightarrow s\Uparrow$.*

Beweis.
Konvergenz
Sei t_{WF} ein Ausdruck in WHNF mit $t \xrightarrow{no,*} t_{WF}$. Aus Lemma 3.4.4 folgt dann, daß ein Ausdruck s_{WF} existiert, mit $s \xrightarrow{no,*} s_{WF}$ und $t_{WF} \xrightarrow{ldel,R[\cdot]} s_{WF}$. Da t_{WF} gemäß Voraussetzung in WHNF ist, ist gemäß Lemma 3.4.3 auch s_{WF} WHNF. Die Argumentation kann unter Zuhilfenahme der beiden zuletzt benutzen Lemmata genauso von s aus angewendet werden, womit die Konvergenz gezeigt ist.

Divergenz
Jede unendliche no-Reduktion verfügt gemäß Korollar 2.4.3 über unendlich viele $(no, lbeta)$-Reduktionen. Durch eine schrittweise Erhöhung der involvierten $(no, lbeta)$-Reduktionen kann gemäß Lemma 3.4.4 eine beliebig lange no-Reduktion von t nach s übertragen werden und umgekehrt. Da jeder Ausdruck maximal zwei direkte Nachfahren besitzt, die mittels einer einzelnen no-Reduktion erreicht werden können, folgt unter Anwendung des Königschen Lemmas, daß eine unendliche no-Reduktion ausgehend von t eine unendliche no-Reduktion ausgehend von s impliziert und umgekehrt. □

Wir werden nun unter indirekter Zuhilfenahme des Kontextlemmas zeigen, daß die Anwendung einer *ldel*-Reduktion auf einen gegebenen Ausdruck zu einem kontextuell äquivalenten Ausdruck führt.

Satz 3.4.1. *Gegeben zwei geschlossene Ausdrücke t und s. Wenn $t \xrightarrow{ldel} s$, dann $t \sim_c s$.*

Beweis. Die *ldel*-Reduktion ist eine kompatible Reduktion, bei der der Redex-Begriff anwendbar ist. Die Hypothese folgt damit direkt aus der in Lemma 3.4.5 gezeigten Erhaltung von Konvergenz und Divergenz durch die $(ldel, R[\cdot])$-Reduktion in Kombination mit Lemma 3.2.3. □

3.4.3 kontextuelle Äquivalenz der *lcv*-Reduktion

In diesem Abschnitt werden wir für die *lcv*-Reduktion zeigen, daß sie bei ihrer Anwendung zu einem kontextuell äquivalenten Ausdruck führt. Dazu wird anders vorgegangen werden als bei der zuvor betrachteten *ldel*-Reduktion. Ein

Rückgriff auf das Kontextlemma ist bei der *lcv*-Reduktion nicht möglich bzw. erbringt keinerlei Vorteile. Die Ursache hierfür zeigt der Beweis des vollständigen Satzes von Gabeldiagrammen der *lcv*-Reduktion in Lemma 3.4.6. Eine Betrachtung der (no, cp)-Reduktion zeigt, daß das Kopierziel eines *lcv*-Redex durch eine (no, cp)-Reduktion vervielfältigt werden kann, was zwei *lcv*-Reduktionen erforderlich macht, um das Gabeldiagramm in diesem Fall schließen zu können. Eine Einschränkung auf die R-Kontexte hilft nicht, diese störende Verdoppelung zu beseitigen.

Zunächst werden wir einen vollständigen Satz von Gabel- sowie Vertauschungsdiagrammen entwickeln sowie die wechselseitige Erhaltung der WHNF durch eine *lcv*-Reduktion zeigen. Danach werden wir ein mit ξ bezeichnetes Bewertungsmaß für Ausdrücke einführen und zwei Hilfslemmata beweisen. Diese Bestandteile werden anschließend benötigt, um beim Beweis der wechselseitigen Übertragbarkeit von Konvergenz und Divergenz induktiv argumentieren zu können. Zum Beweis der wechselseitigen Übertragbarkeit der Divergenz wird bei der *lcv*-Reduktion eine andere Methodik wie bei der im Abschnitt zuvor betrachteten *ldel*-Reduktion verwendet werden. Wir werden zeigen, daß die Eigenschaft, daß alle no-Reduktionen endlich sind, wechselseitig übertragen werden kann und damit der in Abschnitt 3.1.1 vorgestellten Methode folgen.

Lemma 3.4.6. *Die lcv-Reduktion verfügt über den folgenden vollständigen Satz von Gabeldiagrammen:*

i) $\xleftarrow{no,a} \circ \xrightarrow{lcv} \;\leadsto\; \xrightarrow{lcv} \circ \xleftarrow{no,a}$ *für* $a \in \{llet, lapp, lbeta, cp, nd\}$

ii) $\xleftarrow{no,cp} \circ \xrightarrow{lcv} \;\leadsto\; \xrightarrow{lcv} \circ \xrightarrow{lcv} \circ \xleftarrow{no,cp}$

iii) $\xleftarrow{no,cp} \circ \xleftarrow{no,cp} \circ \xrightarrow{lcv} \;\leadsto\; \xleftarrow{i,cp} \circ \xleftarrow{no,cp}$

iv) $\xleftarrow{no,cp} \circ \xrightarrow{lcv} \;\leadsto\; \xrightarrow{i,cp} \circ \xleftarrow{i,cp} \circ \xleftarrow{no,cp}$

Beweis. Wir benutzen die aus Satz 2.3.1 bekannte Kenntnis über die Struktur der Kontexte, in denen die verschiedenen Typen von no-Redizes vorkommen.

$(no, llet)$
Der no-Redex hat in diesem Fall die Struktur $L_R^*[$`let` $x = ($`let` $y = t_y$ `in` $t_x)$ `in` $W[x]]$. Befindet sich der *lcv*-Redex innerhalb eines der Subausdrücke t_x, t_y oder innerhalb des L_R^*-Kontext, wobei letzteres den Fall einschließe, daß in den $(no, llet)$-Redex aus dem L_R^*-Kontext heraus hineinkopiert werde, entsteht immer ein Gabeldiagramm der Form *i)*. Bleiben die Fälle eines *lcv*-Redex innerhalb des inneren let-Ausdrucks des $(no, llet)$-Redex sowie innerhalb von $W[x]$. Bei ersterem Fall entsteht ein Gabeldiagramm der Form $i)$, was folgendes Reduktionsdiagramm zeigt:

$$\begin{tikzcd}
L_R^*[\texttt{let}\ x = (\texttt{let}\ y = z\ \texttt{in}\ t_x[y])\ \texttt{in}\ W[x]] \arrow[r, "lcv"] \arrow[d, "{no,llet}"] & L_R^*[\texttt{let}\ x = (\texttt{let}\ y = z\ \texttt{in}\ t_x[z])\ \texttt{in}\ W[x]] \arrow[d, "{no,llet}"] \\
L_R^*[\texttt{let}\ y = z\ \texttt{in}\ (\texttt{let}\ x = t_x[y]\ \texttt{in}\ W[x])] \arrow[r, "lcv"] & L_R^*[\texttt{let}\ y = z\ \texttt{in}\ (\texttt{let}\ x = t_x[z]\ \texttt{in}\ W[y])]
\end{tikzcd}$$

Beim Fall eines *lcv*-Redex innerhalb von $W[x]$ ist zu zeigen, daß die Markierung des no-Redex durch den no-Markierungsalgorithmus nicht gestört wird. Dazu reicht es wegen der gdw.-Beziehungen bei Satz 2.3.1 zu zeigen, daß eine *lcv*-Reduktion innerhalb von $W[x]$ einen Ausdruck $W'[x]$ liefert, wobei W' ein gegenüber W veränderter W-Kontext sei. Letzteres kann mittels einer einfachen strukturellen Induktion über den Aufbau des W-Kontext erfolgen. Der einzig kritische Fall ist die Zerlegung von $W[x]$ zu:

$$L_R^*[\texttt{let}\ y = [x]\ \texttt{in}\ W[y]]$$

Mittels einer *lcv*-Reduktion können wir obigen Ausdruck zu

$$L_R^*[\texttt{let}\ y = [x]\ \texttt{in}\ W[x]]$$

reduzieren, was jedoch dem Erhalt eines Ausdrucks $W'[x]$ entspricht.

$(no, lapp)$, $(no, lbeta)$, (no, nd)
Bei all diesen Fällen wird eine Analyse erhalten, die der des $(no, llet)$-Falls gleicht. Es entstehen stets Gabeldiagramme der Form *i)*. Beim (no, nd)-Fall ist zusätzlich ein Verfolgen beider Alternativen der nd-Reduktion erforderlich.

(no, cp)
Ein (no, cp)-Redex hat stets das Aussehen $L_R^*[\texttt{let}\ x = t_x\ \texttt{in}\ W[x]]$, wobei t_x eine Abstraktion oder die Konstante `choice` ist. In Abhängigkeit von der Position des *lcv*-Redex können folgende Fälle unterschieden werden:

1) Der *lcv*-Redex befindet sich vollständig innerhalb von t_x:

$$\begin{tikzcd}
L_R^*[\texttt{let}\ x = t_x\ \texttt{in}\ W[x]] \arrow[rr, "lcv"] \arrow[d, "{no,cp}"] & & L_R^*[\texttt{let}\ x = t_x'\ \texttt{in}\ W[x]] \arrow[d, "{no,cp}"] \\
L_R^*[\texttt{let}\ x = t_x\ \texttt{in}\ W[t_x]] \arrow[r, "lcv"] & \circ \arrow[r, "lcv"] & L_R^*[\texttt{let}\ x = t_x'\ \texttt{in}\ W[t_x']]
\end{tikzcd}$$

Wir erhalten offensichtlich ein Gabeldiagramm der Form *ii)*.

2) Das Kopierziel des *lcv*-Redex befindet sich innerhalb von t_x, die Bindung erfolgt jedoch innerhalb des Kontexts L_R^*, d.h. es existiert eine Zerlegung $L_R^*[\texttt{let}\ y = z\ \texttt{in}\ L_R^*[\texttt{let}\ x = t_x[y]\ \texttt{in}\ W[x]]]$. Bei der Analyse diesen Falles wird, vergleichbar dem ersten Fall, ein Gabeldiagramm der Form *ii)* erhalten.

3) Der *lcv*-Redex befindet sich vollständig innerhalb des Kontexts W und führt zu einem Bewegen des Kopierziels des (no, cp)-Redex:

$$\begin{tikzcd}
L_R^*[\texttt{let } x = t_x \texttt{ in } L_R^*[\texttt{let } z = x \texttt{ in } C[z]]] \arrow[rr, "lcv"] \arrow[d, "{no,cp}"] & & L_R^*[\texttt{let } x = t_x \texttt{ in } L_R^*[\texttt{let } z = x \texttt{ in } C[x]]] \arrow[d, "{no,cp}"] \\
L_R^*[\texttt{let } x = t_x \texttt{ in } L_R^*[\texttt{let } z = t_x \texttt{ in } C[z]]] \arrow[dr, "{\{no,i\},cp}"] & & L_R^*[\texttt{let } x = t_x \texttt{ in } L_R^*[\texttt{let } z = x \texttt{ in } C[t_x]]] \arrow[dl, "{i,cp}"] \\
& L_R^*[\texttt{let } x = t_x \texttt{ in } L_R^*[\texttt{let } z = t_x \texttt{ in } C[t_x]]] &
\end{tikzcd}$$

Wir erhalten offensichtlich ein Gabeldiagramm der Form *iii)* oder *iv)*.

4) In allen übrigen Fällen,

- der *lcv*-Redex befindet sich vollständig innerhalb des Kontexts L_R^* oder innerhalb des Subausdrucks t_x,
- das Kopierziel des *lcv*-Redex befindet sich innerhalb des Kontexts W, die Bindung erfolgt jedoch innerhalb des Kontexts L_R^*,
- der *lcv*-Redex befindet sich innerhalb des Kontexts W, führt jedoch zu keiner Bewegung des Kopierziels der (no, cp)-Reduktion,

entsteht stets ein Gabeldiagramm der Form *i)*. □

Neben einem vollständigen Satz von Gabeldiagrammen werden wir auch einen vollständigen Satz von Vertauschungsdiagrammen benötigen.

Lemma 3.4.7. *Die lcv-Reduktion verfügt über den folgenden vollständigen Satz an Vertauschungsdiagrammen:*

i) $lcv \circ (no, a) \rightsquigarrow (no, a) \circ lcv$ *für* $a \in \{llet, lapp, lbeta, cp, nd\}$

ii) $lcv \circ (no, cp) \rightsquigarrow (no, cp) \circ lcv \circ lcv$

iii) $lcv \circ (no, cp) \rightsquigarrow (no, cp) \circ (no, cp) \circ \xleftarrow{i,cp}$

iv) $lcv \circ (no, cp) \rightsquigarrow (no, cp) \circ (i, cp) \circ \xleftarrow{i,cp}$

Die Notation $\xleftarrow{i,cp}$ *bezeichnet dabei eine inverse interne cp-Reduktion.*

Beweis. Mittels der in Abschnitt 2.4.2 beschrieben Interpretation kann aus dem in Lemma 3.4.6 vorgestellten vollständigen Satz von Gabeldiagrammen direkt ein vollständiger Satz von Vertauschungsdiagrammen für die *lcv*-Reduktion hergeleitet werden. Als Besonderheit kann dabei eine inverse (i, cp)-Reduktion entstehen; dies ist eine (i, cp)-Reduktion, bei der die Ausdrücke vor und nach Anwendung der Reduktion getauscht sind. □

Wir zeigen nun die wechselseitige Erhaltung einer WHNF bei Anwendung einer *lcv*-Reduktion.

Lemma 3.4.8. *Gegeben zwei geschlossene Ausdrücke t und s mit $t \xrightarrow{lcv} s$. Dann gilt: t in WHNF gdw. s in WHNF.*

Beweis. Der Beweis erfolgt simultan zu dem Beweis von Lemma 3.4.3. Beide Seiten werden getrennt gezeigt und bei der Richtung "$\Leftarrow$" wird eine Widerspruchsüberlegung benutzt. Wie eine *ldel*-Reduktion, so kann auch eine *lcv*-Reduktion einen Ausdruck der Form $L_R^*[r]$ mit r eine Abstraktion oder die Konstante `choice` in keine andere Form überführen. Aus dem in Lemma 3.4.6 vorgestellten vollständigen Satz von Gabeldiagrammen folgt direkt, daß s einen no-Redex besitzt, wenn t einen no-Redex besitzt. □

In Abschnitt 2.2.1 haben wir die beiden Maße φ und ψ zur Bewertung von Ausdrücken kennengelernt, die dort verwendet wurden, um lokale Konfluenz zu zeigen. Wir werden jetzt ein weiteres Maß ξ einführen, daß wir in diesem Abschnitt benötigen werden, um eine induktive Argumentation leisten zu können. Das Maß ξ wird die Eigenschaften aufweisen, daß es durch Anwendung einer Reduktion der Typen *llet*, *lapp*, *cp*, *nd* stets erniedrigt wird, bei Anwendung einer *lcv*-Reduktion jedoch zu einer unveränderten Bewertung führt.

Definition 3.4.2. *Wir definieren die Interpretation $[\![\ ]\!]_\xi$ unter Zuhilfenahme einer Umgebung ρ, die eine Zuordnung zwischen Variablen und Zahlwerten festhält, wie folgt:*

$$\begin{array}{rcl}
[\![x]\!]_\xi\rho & := & \begin{cases} \rho(x) & \text{falls } \rho(x) \text{ definiert} \\ 1 & \text{falls } \rho(x) \text{ undefiniert} \end{cases} \\
[\![\texttt{choice}]\!]_\xi\rho & := & 2 \\
[\![\lambda x.t]\!]_\xi\rho & := & [\![t]\!]_\xi\rho[x \mapsto 1] \\
[\![st]\!]_\xi\rho & := & 2 * [\![s]\!]_\xi\rho + 2 * [\![t]\!]_\xi\rho \\
[\![\texttt{let } x = t_x \texttt{ in } t]\!]_\xi\rho & := & 2 * [\![t_x]\!]_\xi\rho + [\![t]\!]_\xi\rho' \\
 & & \text{wobei } \rho' := \rho[x \mapsto [\![t_x]\!]_\xi\,\rho + a] \\
 & & \text{mit } a := \begin{cases} 0 & \text{falls } t_x \text{ eine Variable} \\ 1 & \text{sonst} \end{cases}
\end{array}$$

Für einen Ausdruck t definieren wir $\xi(t) := [\![t]\!]_\xi\,\emptyset$.

Wir werden zunächst ein im Anschluß benötigtes Lemma beweisen, das die Erhaltung von ungleichen Bewertungen durch die Interpretation $[\![\ ]\!]_\xi$ beim Einschluß in umgebende Kontexte aufzeigt.

Lemma 3.4.9. *Gegeben zwei Ausdrücke t, s so daß $[\![t]\!]_\xi\rho > [\![s]\!]_\xi\rho$ für alle Umgebungen ρ gilt. Dann gilt $[\![C[t]]\!]_\xi\rho > [\![C[s]]\!]_\xi\rho$ für alle Kontexte C und Umgebungen ρ.*

Beweis. Eine einfache strukturelle Induktion über den Aufbau des Kontext.
Induktionsbeginn:
Der Kontext entspricht dem leeren Kontext. Die Gültigkeit der Hypothese ist in diesem Fall trivial.

Induktionsschritt:
Wir differenzieren nach dem Aufbau des Kontext C. Stellvertretend betrachten wir die Zerlegung $C \equiv C'[\cdot]\,t'$:
Wir wenden die Definition der Interpretation ξ an und erhalten $[\![C'[t]\,t']\!]_\xi\rho = 2 * n_t + c$ sowie $[\![C'[s]\,t']\!]_\xi\rho = 2 * n_s + c$, wobei $n_t = [\![C'[t]]\!]_\xi\rho$, $n_s = [\![C'[s]]\!]_\xi\rho$ und $c = 2 * [\![t']\!]_\xi\rho$. Gemäß Induktionsvoraussetzung gilt $n_t > n_s$. Daraus folgt direkt $2 * n_t + c > 2 * n_s + c$ und wir haben die Hypothese für die Zerlegung gezeigt.

Bei allen anderen Zerlegungen ist eine ähnliche Argumentation möglich. □

Unter Zuhilfenahme des obigen Lemmas kann die Verringerung der Bewertung des Maßes ξ durch Reduktionen der zuvor aufgeführten Typen gezeigt werden.

Lemma 3.4.10. *Gegeben zwei geschlossene Ausdrücke t und s. Wenn $t \xrightarrow{a} s$ für $a \in \{llet, lapp, cp, nd\}$, dann $\xi(t) > \xi(s)$. Wenn $t \xrightarrow{lcv} s$, dann $\xi(t) = \xi(s)$.*

Beweis. Lemma 3.4.9 zeigt, daß eine Verkleinerung der Bewertung eines Ausdrucks sich stets in gleicher Weise auf die Bewertung bei Einschluß in einen umgebenden Kontext auswirkt. Ferner läßt sich leicht nachprüfen, daß die Bewertung eines Ausdrucks durch die Interpretation ξ stets echt größer als 0 ist.
Es bleibt somit zu zeigen, daß die Reduzierung eines Redex vom Typ *llet*, *lapp*, *nd* oder *cp* stets zu einer Reduzierung der Bewertung führt bzw. die Reduzierung eines *lcv*-Redex zu keiner Veränderung der Bewertung führt.

1. (*llet*)

$$\begin{aligned}&[\![\texttt{let}\ x = (\texttt{let}\ y = t_y\ \texttt{in}\ t_x)\ \texttt{in}\ t]\!]_\xi\,\rho\\ &= 2 * [\![\texttt{let}\ y = t_y\ \texttt{in}\ t_x]\!]_\xi\,\rho + [\![t]\!]_\xi\,\rho[x \mapsto [\![\texttt{let}\ y = t_y\ \texttt{in}\ t_x]\!]_\xi\,\rho + a]\\ &= 4*[\![t_y]\!]_\xi\,\rho + 2*[\![t_x]\!]_\xi\,\rho[y \mapsto [\![t_y]\!]_\xi\,\rho + a'] + [\![t]\!]_\xi\,\rho[x \mapsto [\![\texttt{let}\ y = t_y\ \texttt{in}\ t_x]\!]_\xi\,\rho + a]\end{aligned}$$

wobei

$$\begin{aligned}&[\![t]\!]_\xi\,\rho[x \mapsto [\![\texttt{let}\ y = t_y\ \texttt{in}\ t_x]\!]_\xi\,\rho + a]\\ &= [\![t]\!]_\xi\,\rho[x \mapsto [\![t_y]\!]_\xi\,\rho + [\![t_x]\!]_\xi\,\rho[y \mapsto [\![t_y]\!]_\xi\,\rho + a'] + a]\end{aligned}$$

$$\begin{aligned}&[\![\texttt{let}\ y = t_y\ \texttt{in}\ (\texttt{let}\ x = t_x\ \texttt{in}\ t)]\!]_\xi\,\rho\\ &= 2 * [\![t_y]\!]_\xi\,\rho + [\![\texttt{let}\ x = t_x\ \texttt{in}\ t]\!]_\xi\,\rho[y \mapsto [\![t_y]\!]_\xi\,\rho + a']\\ &= 2*[\![t_y]\!]_\xi\,\rho + 2*[\![t_x]\!]_\xi\,\rho[y \mapsto [\![t_y]\!]_\xi\,\rho + a'] + [\![t]\!]_\xi\,\rho[x \mapsto [\![t_x]\!]_\xi\,\rho[y \mapsto [\![t_y]\!]_\xi\,\rho + a'] + a]\end{aligned}$$

Es gilt $4 * [\![t_y]\!]_\xi\,\rho > 2 * [\![t_y]\!]_\xi\,\rho$, da $[\![t_y]\!]_\xi\,\rho > 0$.

2. $(lapp)$

   $[\![(\mathtt{let}\ x = t_x\ \mathtt{in}\ t)s]\!]_\xi\, \rho$
   $= 2 * [\![\mathtt{let}\ x = t_x\ \mathtt{in}\ t]\!]_\xi\, \rho + 2 * [\![s]\!]_\xi\, \rho$
   $= 4 * [\![t_x]\!]_\xi\, \rho + 2 * [\![t]\!]_\xi\, \rho[x \mapsto [\![t_x]\!]_\xi\, \rho + a] + 2 * [\![s]\!]_\xi\, \rho$

   $[\![\mathtt{let}\ x = t_x\ \mathtt{in}\ (ts)]\!]_\xi\, \rho$
   $= 2 * [\![t_x]\!]_\xi\, \rho + [\![ts]\!]_\xi\, \rho[x \mapsto [\![t_x]\!]_\xi\, \rho + a]$
   $= 2 * [\![t_x]\!]_\xi\, \rho + 2 * [\![t]\!]_\xi\, \rho[x \mapsto [\![t_x]\!]_\xi\, \rho + a] + 2 * [\![s]\!]_\xi\, \rho[x \mapsto [\![t_x]\!]_\xi\, \rho + a]$

   Es gilt

   a) $4 * [\![t_x]\!]_\xi\, \rho > 2 * [\![t_x]\!]_\xi\, \rho$, da $[\![t_x]\!]_\xi\, \rho > 0$
   b) $[\![s]\!]_\xi\, \rho[x \mapsto [\![t_x]\!]_\xi\, \rho + a] = [\![s]\!]_\xi\, \rho$, da x nicht in s vorkommen darf.

3. (nd)

   $[\![\mathtt{choice}\ t]\!]_\xi\, \rho$
   $= 2 * 2 + 2 * [\![t]\!]_\xi\, \rho$

   $[\![(\lambda x.\lambda y.x)t]\!]_\xi\, \rho$
   $= 2 * 1 + 2 * [\![t]\!]_\xi\, \rho$

   Das gleiche gilt für die andere Alternative der `choice` -Reduktion.

4. (cp, lcv)

   $[\![\mathtt{let}\ x = t_x\ \mathtt{in}\ C[x]]\!]_\xi\, \rho$
   $= 2 * [\![t_x]\!]_\xi\, \rho + [\![C[x]]\!]_\xi\, \rho[x \mapsto [\![t_x]\!]_\xi\, \rho + a]$

   $[\![\mathtt{let}\ x = t_x\ \mathtt{in}\ C[t_x]]\!]_\xi\, \rho$
   $= 2 * [\![t_x]\!]_\xi\, \rho + [\![C[t_x]]\!]_\xi\, \rho[x \mapsto [\![t_x]\!]_\xi\, \rho + a]$

   $\Rightarrow$ falls $a > 0$ gilt $[\![C[x]]\!]_\xi\, \rho[x \mapsto [\![t_x]\!]_\xi\, \rho + a] > [\![C[t_x]]\!]_\xi\, \rho[x \mapsto [\![t_x]\!]_\xi\, \rho + a]$, d.h falls t_x eine Abstraktion oder die Konstante `choice` wird das Maß erniedrigt, falls t_x eine Variable bleibt das Maß gleich. □

Neben dem oben vorgestellten Maß ξ werden wir noch weitere Hilfsmittel benötigen, um in diesem Abschnitt später induktiv argumentieren zu können. In den Gabel- bzw. Vertauschungsdiagrammen der *lcv*-Reduktion erscheinen (i, cp)-Reduktionen. Um mit diesen umgehen zu können, werden wir die nun folgenden zwei Lemmata benötigen.

Lemma 3.4.11. *Gegeben zwei Ausdrücke* t, s *mit* $t \xrightarrow{i,cp} s$ *sowie ein Ausdruck* t_{WF} *in WHNF mit* $t \xrightarrow{no,*} t_{WF}$*. Dann existiert ein Ausdruck* s_{WF} *in WHNF, so daß* $s \xrightarrow{no,*} s_{WF}$ *und* $\#lbeta(t \xrightarrow{no,*} t_{WF}) = \#lbeta(s \xrightarrow{no,*} s_{WF})$ *gilt.*

Beweis. Die Existenz von s_{WF} in WHNF folgt direkt aus der in Satz 3.3.1 bewiesenen kontextuellen Äquivalenz der *cp*-Reduktion. Bleibt die Erhaltung der

Anzahl an *lbeta*-Reduktionen zu beweisen. Dies zeigt eine einfache Betrachtung der Gabeldiagramme der $(i, cppar)$-Reduktion aus Lemma 3.3.1, auf deren transitiver Anwendung der Beweis der Erhaltung der kontextuellen Äquivalenz durch eine *cp*-Reduktion aufbaut. Diese können eine $(no, lbeta)$-Reduktion weder vervielfältigen noch löschen, somit muß die Zahl gleichbleiben. □

Das folgende Lemma beinhaltet dieselbe Aussage wie das obige, nur wird nun von s ausgegangen.

Lemma 3.4.12. *Gegeben zwei Ausdrücke t, s mit $t \xrightarrow{i,cp} s$ sowie ein Ausdruck s_{WF} in WHNF mit $s \xrightarrow{no,*} s_{WF}$. Dann existiert ein Ausdruck t_{WF} in WHNF, so daß $t \xrightarrow{no,*} t_{WF}$ und $\#lbeta(s \xrightarrow{no,*} s_{WF}) = \#lbeta(t \xrightarrow{no,*} t_{WF})$ gilt.*

Beweis. Die Argumentation kann von Lemma 3.4.11 übernommen werden. Satt der Gabeldiagramme werden jedoch die Vertauschungsdiagramme der $(i, cppar)$-Reduktion aus Lemma 2.4.7 benutzt. □

Wir werden nun die bis jetzt in diesem Abschnitt vorgestellten Lemmata dazu benutzen, um vier Lemmata zu entwickeln, die in ihrer Gesamtheit dazu geeignet sind, die Erhaltung der kontextuellen Äquivalenz durch die *lcv*-Reduktion zu folgern. Wir werden zunächst zwei Lemmata vorstellen, die jeweils in einer Richtung die Übertragbarkeit einer Reduktion zu WHNF bei Anwendung einer *lcv*-Reduktion zeigen. Danach wird, wiederum für beide Richtungen getrennt, die Übertragbarkeit der Endlichkeit allen no-Reduktionen gezeigt werden.

Lemma 3.4.13. *Gegeben zwei Ausdrücke t, s mit $t \xrightarrow{lcv} s$ sowie ein Ausdruck t_{WF} in WHNF mit $t \xrightarrow{no,*} t_{WF}$. Dann existiert ein Ausdruck s_{WF} in WHNF, so daß $s \xrightarrow{no,*} s_{WF}$ und $\#lbeta(t \xrightarrow{no,*} t_{WF}) = \#lbeta(s \xrightarrow{no,*} s_{WF})$ gilt.*

Beweis. Wir führen eine Induktion über die Menge aller Reduktionen $t \xrightarrow{no,*} t_{WF}$ mit t in WHNF, die wir wie folgt ordnen:

$$(t \rightarrow t_{WF}) > (s \rightarrow s_{WF}) \Leftrightarrow (\#lbeta(t \rightarrow t_{WF}), \xi(t)) > (\#lbeta(s \rightarrow s_{WF}), \xi(s))$$

Die Tupelkomponenten sind dabei lexikographisch geordnet. Als Induktionshypothese benutzen wir die Hypothese des Lemmas.

Induktionsbeginn:
Die Bedingung $\#lbeta = 0$ impliziert, daß die Reduktion $t \xrightarrow{no,*} t_{WF}$ keine *lbeta*-Reduktionen beinhaltet. Aus $\xi(t)$ minimal ergibt sich, daß auf t keine Reduktion außer einer *lbeta*-Reduktion angewendet werden kann. Somit muß sich t in WHNF befinden. Aus Lemma 3.4.8 folgt daraus, daß sich auch s in WHNF befindet. Wir wählen $s_{WF} \equiv s$ und haben die Hypothese für diesen Fall gezeigt.

Diagramm 1

$$\begin{tikzcd}
t \arrow[r, "lcv"] \arrow[d, "no"'] \arrow[dr, phantom, "\text{Lemma 3.4.6, } i) \ \text{①}"] & s \arrow[d, dashed, "no"] \\
t' \arrow[r, dashed, "lcv"'] \arrow[d, "{no,*}"'] \arrow[dr, phantom, "\text{Ind. Voraus. } \text{②}"] & \circ \arrow[d, dashed, "{no,*}"] \\
t_{WF} \arrow[r, dotted, no head] & s_{WF}
\end{tikzcd}$$

Diagramm 2

$$\begin{tikzcd}
t \arrow[r, "lcv"] \arrow[d, "{no,cp,2}"'] \arrow[dr, phantom, "\text{Lemma 3.4.6, } iii) \ \text{①}"] & s \arrow[d, dashed, "{no,cp}"] \\
t' \arrow[d, "{no,*}"'] \arrow[dr, phantom, "\text{Lemma 3.4.11 } \text{②}"] & \circ \arrow[l, dashed, "{i,cp}"] \arrow[d, dashed, "{no,*}"] \\
t_{WF} \arrow[r, dotted, no head] & s_{WF}
\end{tikzcd}$$

Diagramm 4

$$\begin{tikzcd}
t \arrow[rr, "lcv"] \arrow[d, "{no,cp}"'] \arrow[drr, phantom, "\text{Lemma 3.4.6, } ii) \ \text{①}"] & & s \arrow[d, dashed, "{no,cp}"] \\
t' \arrow[r, dashed, "lcv"] \arrow[d, "{no,*}"'] \arrow[dr, phantom, "\text{②}"] & t'' \arrow[r, dashed, "lcv"] \arrow[d, dashed, "{no,*}"'] \arrow[d, shift left] \arrow[dr, phantom, "\text{③}"] & \circ \arrow[d, dashed, "{no,*}"] \\
t_{WF} \arrow[r, dotted, no head] & t'_{WF} \arrow[r, dotted, no head] & s_{WF}
\end{tikzcd}$$

Induktionsvoraussetzung

Diagramm 3

$$\begin{tikzcd}
t \arrow[rr, "lcv"] \arrow[d, "{no,cp}"'] \arrow[drr, phantom, "\text{Lemma 3.4.6, } iv) \ \text{①}"] & & s \arrow[d, dashed, "{no,cp}"] \\
t' \arrow[r, dashed, "{i,cp}"] \arrow[d, "{no,*}"'] \arrow[dr, phantom, "\text{②}"] & \circ \arrow[d, dashed, "{no,*}"'] \arrow[d, shift left] \arrow[dr, phantom, "\text{③}"] & \circ \arrow[l, dashed, "{i,cp}"'] \arrow[d, dashed, "{no,*}"] \\
t_{WF} \arrow[r, dotted, no head] & t'_{WF} \arrow[r, dotted, no head] & s_{WF}
\end{tikzcd}$$

Lemma 3.4.11 Lemma 3.4.12

Abbildung 3.4:

Induktionsschritt:
Wir zerlegen die no-Reduktion von t zu t_{WF} in $t \xrightarrow{no,+} t' \xrightarrow{no,*} t_{WF}$ derart, daß sich eines der Gabeldiagramme der lcv-Reduktion aus Lemma 3.4.6 schließen läßt.
Die in Abbildung 3.4 enthaltenen Reduktionsdiagramme geben, für alle Gabeldiagramme einen Beweis der Hypothese in grafischer Form. Die Lemmata sind dabei entsprechend der Nummerierung anzuwenden und t'_{WF} ist ein Ausdruck in WHNF.
Die von t' ausgehende no-Reduktion zu t_{WF} ist um mindestens eine einzelne no-Reduktion verkürzt und wird daher wegen der in Lemma 3.4.10 vorgestellten Eigenschaften des Maßes ξ geringer bewertet als die von t ausgehende. Somit kann auf die Reduktion $t' \xrightarrow{no,*} t_{WF}$ die Induktionshypothese angewendet werden kann.
Bei den Diagrammen 2 und 3 ist kein weiterer Rückgriff auf die Induktionshypothese erforderlich, da wir die Lemma 3.4.11 und 3.4.12 benutzen können. Bei Diagramm 4 erfolgt bei der Reduktion $t'' \rightarrow t'_{WF}$ ein zweiten Rückgriff auf die Induktionsvoraussetzung. Dies ist aus den folgenden Gründen zulässig:

Die Hypothese beinhaltet eine Erhaltungsregel für $(no, lbeta)$-Reduktionen. So-

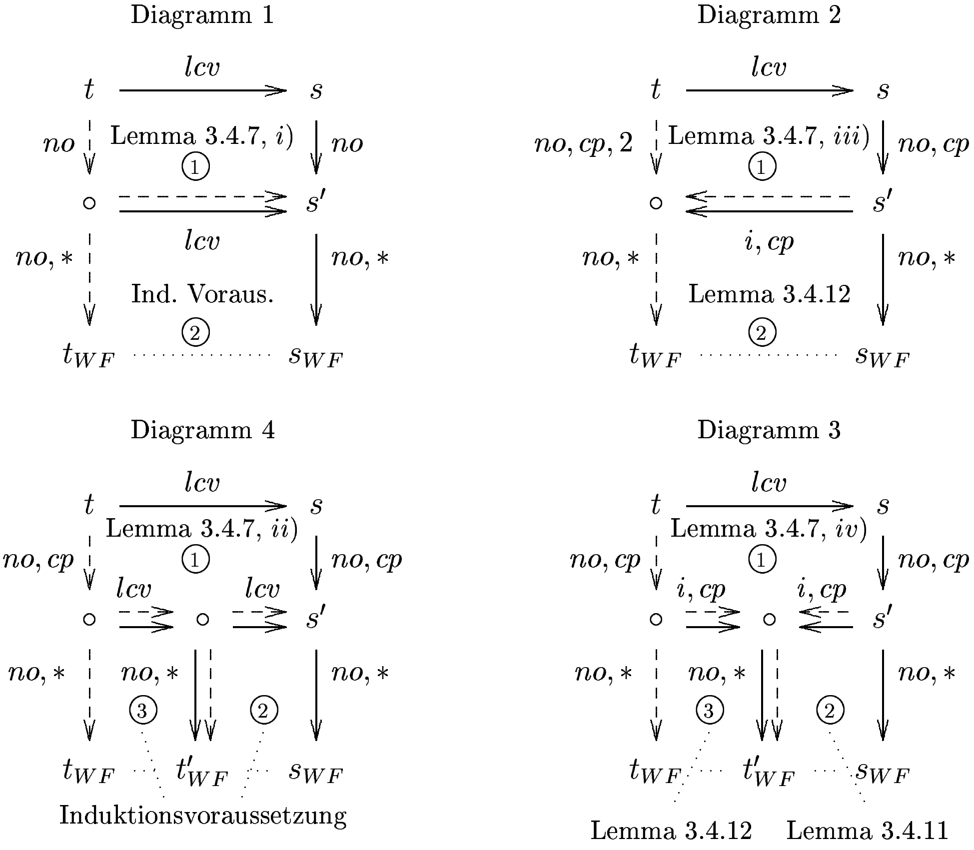


Diagramm 1 Diagramm 2 Diagramm 4 Diagramm 3

Abbildung 3.5:

mit folgt aus der ersten Anwendung der Induktionshypothese die Gültigkeit von $\#lbeta(t' \rightarrow t_{WF}) = \#lbeta(t'' \rightarrow t'_{WF})$. Das Maß ξ bleibt bei Anwendung einer *lcv*-Reduktion gemäß Lemma 3.4.10 gleich. Daraus folgt insgesamt, daß die Bewertung der von t' und t'' ausgehenden no-Reduktionen identisch und damit kleiner als die Bewertung der von t ausgehenden no-Reduktion ist.

Bleibt die Erhaltung der Anzahl an $(no, lbeta)$-Reduktionen zu zeigen. Dies ist offensichtlich, da nur Diagramm 1 zu einer $(no, lbeta)$-Reduktion paßt und die beiden Lemma 3.4.11 und 3.4.12 eine Erhaltungsaussage beinhalten. □

Was wir soeben für eine von t ausgehende no-Reduktion gezeigt haben, zeigen wir nun genauso für eine von s ausgehende no-Reduktion. Wir benutzen dabei anstelle der Gabeldiagramme die Vertauschungsdiagramme aus Lemma 3.4.7.

Lemma 3.4.14. *Gegeben zwei Ausdrücke t, s mit $t \xrightarrow{lcv} s$ sowie ein Ausdruck s_{WF} in WHNF mit $s \xrightarrow{no,*} s_{WF}$. Dann existiert ein Ausdruck t_{WF} in WHNF, so daß $t \xrightarrow{no,*} t_{WF}$ und $\#lbeta(s \xrightarrow{no,*} s_{WF}) = \#lbeta(t \xrightarrow{no,*} t_{WF})$ gilt.*

Beweis. Die Argumentation erfolgt Induktions-basiert nach demselben Muster wie bei Lemma 3.4.13 nur diesmal nicht ausgehend von t, sondern ausgehend von s. Statt der Gabeldiagramme werden die Vertauschungsdiagramme aus Lemma 3.4.7 benutzt. Die in Abbildung 3.5 enthaltenen Reduktionsdiagramme geben, für alle Vertauschungsdiagramme einen Beweis der Hypothese in grafischer Form. □

Wir haben in den letzten beiden Lemmata zwei Existenz-quantifizierte Aussagen getroffen, die geeignet sind, die in der kontextuellen Äquivalenz der lcv-Reduktion enthaltene Übertragbarkeit der Konvergenz zu zeigen. Im Bezug auf die auch zu zeigende Übertragbarkeit der Divergenz sind die beiden Lemmata jedoch wertlos, da sie erfordern, daß die gegebene Reduktion zu WHNF führt. Wir werden nun für zwei Ausdrücke t und s mit $t \xrightarrow{lcv} s$ zeigen, daß die Endlichkeit aller von t ausgehenden no-Reduktionen die Endlichkeit aller von s ausgehenden no-Reduktionen impliziert und umgekehrt. Durch eine einfache Widerspruchsüberlegung können wir daraus die Übertragbarkeit der Divergenz folgern.

Lemma 3.4.15. *Gegeben zwei geschlossene Ausdrücke* t, s *mit* $t \xrightarrow{lcv} s$. *Dann gilt* $t\not\Uparrow \Rightarrow\ s\not\Uparrow$.

Beweis. Wir führen eine Induktion über die Menge aller Ausdrücke t, die wir wie folgt ordnen:

$$t > s \text{ gdw. } (L_{max}(t), \xi(t)) > (L_{max}(s), \xi(s))$$

Dabei sind die Tupel lexikographisch geordnet und $L_{max}(t)$ liefere die Anzahl an $(no, lbeta)$-Reduktionen derjenigen von t ausgehenden no-Reduktion, die über die größte Anzahl an $(no, lbeta)$-Reduktionen verfüge. Als Induktionshypothese benutzen wir die Hypothese des Lemmas. Die Argumentation ist der bei Lemma 3.4.13 angewendeten sehr verwandt.

Induktionsbeginn:
Ein Bewertung der Form $(0, 0)$ impliziert, daß sich t sich in WHNF befindet, woraus gemäß Lemma 3.4.8 folgt, daß sich auch s in WHNF befindet und wir die Hypothese für diesen Fall gezeigt haben.

Induktionsschritt:
Wir differenzieren danach, ob der no-Redex von t vom Typ nd ist oder nicht.

1. Fall: Der no-Redex von t ist vom Typ $llet$, $lapp$ oder $lbeta$.
Wir wählen Gabeldiagramm $i)$ aus Lemma 3.4.6. Diagramm 1 aus Abbildung 3.6 zeigt die entstehenden Reduktionszusammenhänge. Alle von t' ausgehenden no-Reduktion enthalten entweder weniger $(no, lbeta)$-Reduktionen oder t' wird durch das Maß ξ geringer bewertet als t. Somit können wir bei t' die Induktionshypothese anwenden. Da die no-Reduktion von s zu s' nicht vom Typ nd sein kann, ist die Endlichkeit aller von s ausgehenden no-Reduktionen gezeigt.

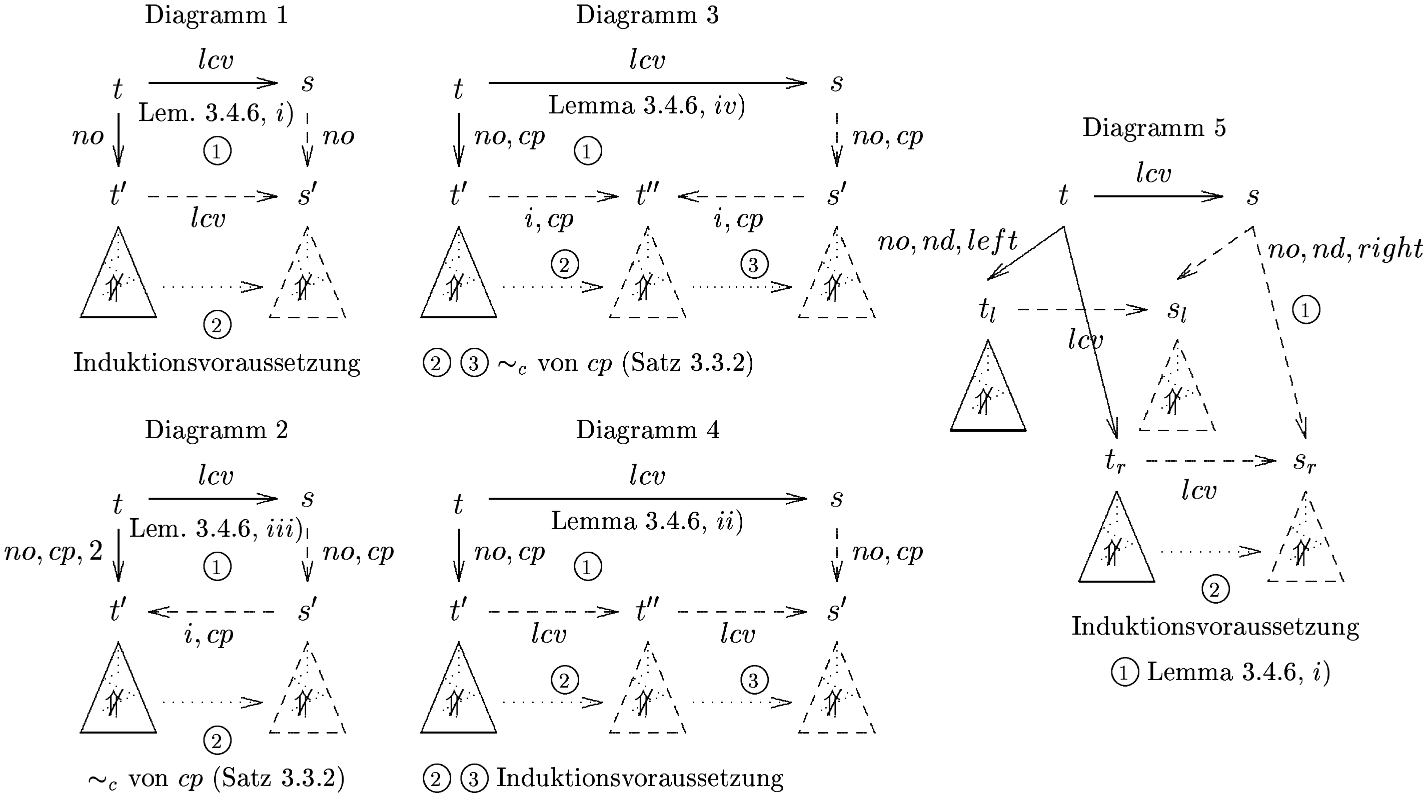


Abbildung 3.6:

2. Fall: Der no-Redex von t ist vom Typ *cp*.
Entsprechend der no-Reduktion wählen wir eines der Gabeldiagramme *i)* bis *iii)* aus Lemma 3.4.6. Die Diagramme 2 bis 4 aus Abbildung 3.6 zeigen die entstehenden Reduktionszusammenhänge. Für t' gilt dasselbe wie beim ersten Fall, somit kann bei t' die Induktionsvoraussetzung angewendet werden. Bei den Diagrammen 2 und 3 benutzen wir die in Satz 3.3.2 gezeigte Erhaltung der kontextuellen Äquivalenz durch eine *cp*-Reduktion, um die Endlichkeit aller von s ausgehenden no-Reduktionen zu folgern.
Bei Diagramm 4 ist bei t'' ein weiteres Mal die Anwendung der Induktionsvoraussetzung erforderlich. Eine einfache Überlegung zeigt, daß dies zulässig ist:
Da t'' durch Anwendung einer *lcv*-Reduktion von t' aus erreicht werden kann und werden gemäß Lemma 3.4.10 beide Ausdrücke durch das Maß ξ identisch bewertet. Ein einfache Widerspruchsüberlegung unter Zuhilfenahme von Lemma 3.4.12 zeigt die Gültigkeit von $L_{max}(t') \geq L_{max}(t'')$. Die Annahme $L_{max}(t') > L_{max}(t'')$ erzeugt einen Widerspruch zu der in Lemma 3.4.12 formulierten Erhaltung der $(no, lbeta)$-Reduktionen.

3. Fall: Der no-Redex von t ist vom Typ *nd*.
Der Beweis der Hypothese kann bei diesem Fall aus Diagramm5 von Abbildung 3.6 abgelesen werden. Die von t ausgehende no-Reduktion ist bei einem (no, nd)-

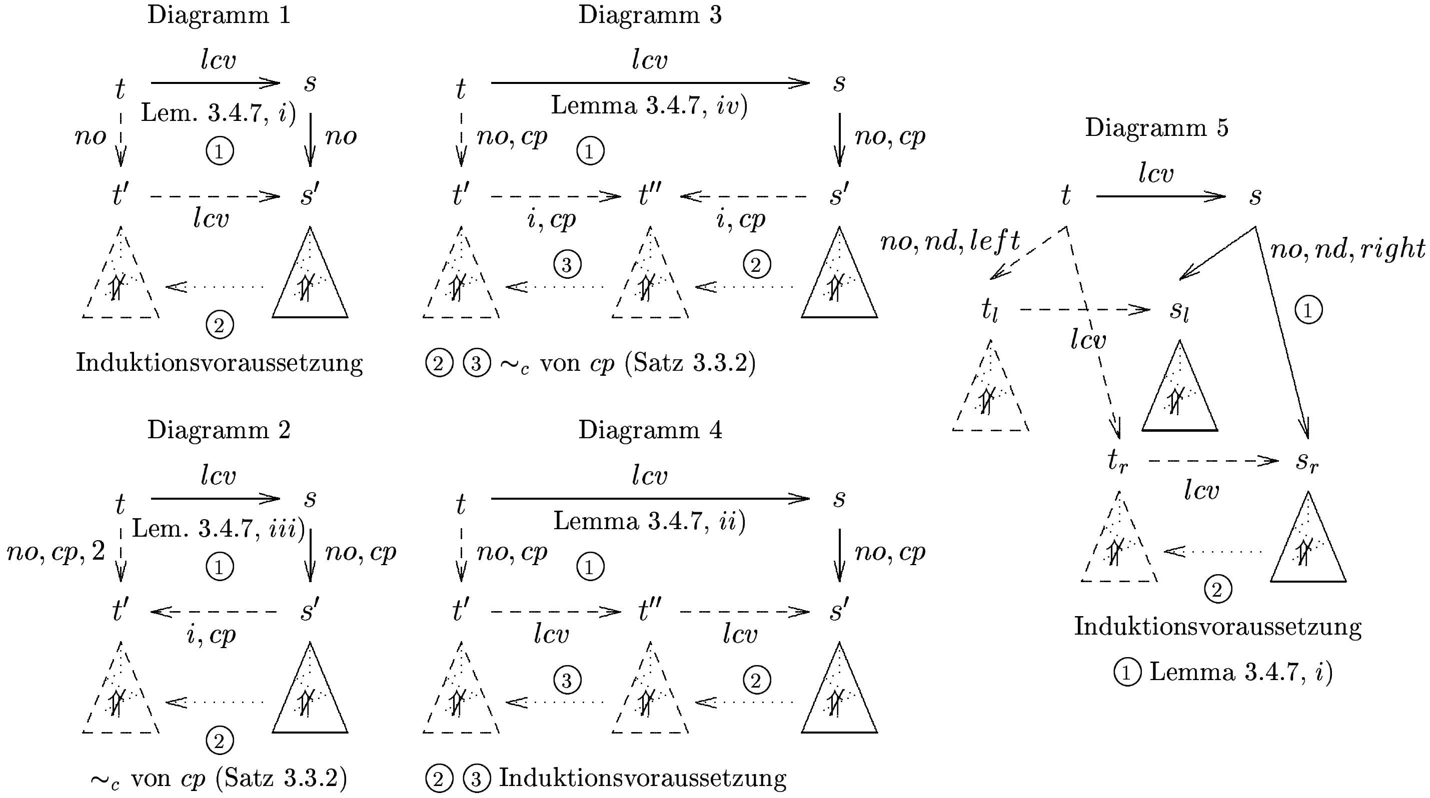


Abbildung 3.7:

Redex nicht eindeutig. Die Bewertung von t_l wie t_r gemäß des oben vorgestellten Maßes ist in beiden Fällen geringer als die von t. Die im ersten Fall benutzte Argumentation kann deshalb unter Zuhilfenahme der Induktionsvoraussetzung simultan auf beide "Zweige" angewendet werden und so gezeigt werden, daß alle von s_l wie s_r ausgehenden no-Reduktionen endlich sind. Da ausgehend von s mittels einer einzelnen no-Reduktion keine anderen Ausdrücke außer s_l und s_r erreichbar sind und alle von s_l wie s_r ausgehenden no-Reduktionen endlich sind, müssen auch alle von s ausgehenden no-Reduktionen endlich sein. □

Die Aussage des obigen Lemmas zeigen wir nun genauso ausgehend von s.

Lemma 3.4.16. *Gegeben zwei geschlossene Ausdrücke* t, s *mit* $t \xrightarrow{lcv,R[\cdot]} s$. *Dann gilt* $s{\not\Uparrow} \Rightarrow t{\not\Uparrow}$.

Beweis. Der Beweis ist dem des zuvor gezeigten Lemma 3.4.15 sehr verwandt, beide Beweise stehen in einem ähnlichen Verhältnis wie die Beweise von Lemma 3.4.13 und Lemma 3.4.14. Um eine unnötige Ausdehnung des *lcv*-Beweises zu vermeiden, soll der Beweis nur kurz skizziert werden:
Es wird eine Induktion unter Zuhilfenahme derselben Ordnung von Ausdrücken

wie beim zuvor gezeigten Lemma geführt und als Induktionshypothese die Hypothese des Lemmas für einen gegebenen Ausdruck s benutzt. Es werden die Vertauschungsdiagramme aus Lemma 3.4.7 verwendet sowie die in Satz 3.3.2 gezeigte kontextuelle Äquivalenz. Abbildung 3.7 gibt für alle Vertauschungsdiagramme einen Beweis in grafischer Form. □

Die Erhaltung der kontextuellen Äquivalenz ist nun nur noch eine einfache Schlußfolgerung aus den 4 zuvor bewiesenen Lemmata.

Satz 3.4.2. *Gegeben zwei geschlossene Ausdrücke t und s. Wenn $t \xrightarrow{lcv} s$, dann $t_1 \sim_c t_2$.*

Beweis. Die Hypothese folgt wegen der Kompatibilität der *lcv*-Reduktion aus Proposition 3.1.2 in Kombination mit den vier Lemmata 3.4.13, 3.4.14, 3.4.15, 3.4.16 und der folgenden einfachen Überlegung: Aus $(s{\not\Uparrow} \Rightarrow t{\not\Uparrow})$ folgt $(t{\Uparrow} \Rightarrow s{\Uparrow})$ durch logische Umkehrung, bzw. Widerspruch. □

Ein kurzer Einschub soll verdeutlichen, warum bei der *lcv*-Reduktion und allen weiteren Reduktion nicht weiter die bei der *ldel*-Reduktion zum Beweis der wechselseitigen Übertragbarkeit der Divergenz benutzte Methodik verwendet wurde. Direkt im Anschluß daran wird fortgefahren, indem die Erhaltung der kontextuellen Äquivalenz bei der *lcom*-Reduktion gezeigt wird.

Anmerkung zum Beweis der kontextuellen Äquivalenz der *lcv*-Reduktion

Ein kritischer Leser könnte an dieser Stelle die Frage stellen, warum bei obigem Beweis bei der wechselseitigen Übertragung der Divergenz nicht genauso wie bei der *ldel*-Reduktion vorgegangen wurde. So beinhalten doch die Lemmata 3.4.13 und 3.4.14 beide ein Erhaltungsgesetz für $(no, lbeta)$-Reduktionen. Warum sollte das Detail einer Reduktion zu WHNF in den Hypothesen dieser beiden Lemmata nicht fallengelassen werden und wie bei der *ldel*-Reduktion eine no-Reduktion zu einem beliebigen Ausdruck als gegeben angesehen werden?
Das Detail, eine no-Reduktion zu WHNF verlangen, ermöglicht eine vereinfachte Betrachtung, die bei der *lcv*-Reduktion nicht mehr zulässig wäre, wenn eine no-Reduktion zu einem beliebigen Ausdruck gegeben ist. Dieses soll nun erläutert werden.
Ein kritischer Punkt bei der in Lemma 3.4.4 verwendeten Beweismethodik ist, daß die gegebene no-Reduktion von t zu t' zu kurz sein kann um ein Gabeldiagramm zu schließen. Verlangt jedes Gabeldiagramm auf der linken Seite nur eine no-Reduktion, wie dies beispielsweise bei der *ldel*-Reduktion der Fall ist, so ist

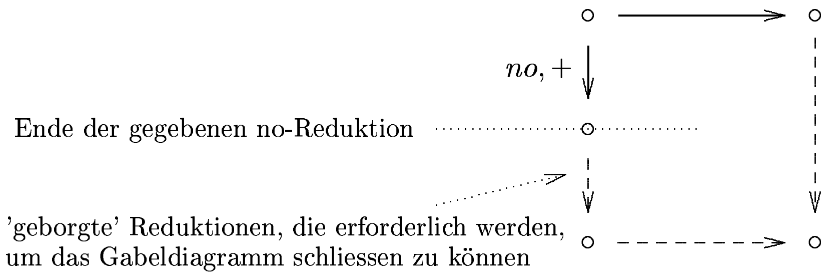


Abbildung 3.8:

dieser Fall nicht möglich. Aber das Diagramm *iii*) aus Lemma 3.4.6 zum Beispiel verlangt zwei (no, cp)-Reduktionen auf der linken Seite. Ist die gegebene no-Reduktion zu kurz müssen wir Reduktionen 'borgen', d.h. es entsteht ein Reduktionszusammenhang wie ihn Abbildung 3.8 zeigt. Diese geborgten Reduktionen sind Existenz-quantifiziert und müssen in die Hypothese aufgenommen werden. In der Folge tauchen sie im Beweis bei der Induktionsvoraussetzung wieder auf. Dieser Zusammenhang kann ausgesprochen störend wirken und im weiteren zu einem vollständig intransparenten Beweis führen. Bezogen auf die zuletzt untersuchte *lcv*-Reduktion ist in diesem Zusammenhang die zweite Anwendung der Induktionsvoraussetzung bei Diagramm 4 aus Abbildung 3.4 zu erwähnen. Wäre die Bedingung einer Reduktion zu WHNF fallengelassen worden, wäre in dem erwähnten Diagramm unterhalb der ② ein weiteres Subdiagramm aufgetaucht, daß dann unglücklicherweise sogar eine Vertauschung für eine *lcv*-Reduktion beinhaltet hätte, obwohl wir diese zu diesem Zeitpunkt noch nicht einmal bewiesen hätten.

Indem wir eine no-Reduktion zu WHNF verlangen, können wir dieses Problem umgehen, da eine no-Reduktion zu WHNF niemals zu kurz sein kann, um ein Gabeldiagramm zu schließen. Der Preis für das mehr an Transparenz sind zwei zusätzliche Lemmata, die die wechselseitige Übertragung der Endlichkeit aller no-Reduktionen zeigen.

3.4.4 kontextuelle Äquivalenz der *lcom*-Reduktion

Nun werden wir für *lcom*-Reduktion zeigen, daß sie bei ihrer Anwendung zu einem kontextuell äquivalenten Ausdruck führt. Der Beweis erfolgt dabei nach demselben Muster wie bei der *lcv*-Reduktion. Wir zeigen zunächst einen vollständigen Satz von Gabel- und Vertauschungsdiagrammen sowie die wechselseitige Erhaltung einer WHNF. Im Anschluß werden daraus 4 Lemmata gebildet, die in ihrer Gesamtheit die Schlußfolgerung erlauben, daß die *lcom*-Reduktion die kontextuelle Äquivalenz erhält.

Bei der *lcom*-Reduktion macht der Rückgriff auf das Kontextlemma wie auch bei der *lcv*-Reduktion keinen Sinn, bzw. erbringt keinerlei Vorteil. Die Ursache dafür ist die in Diagramm $v)$ vorkommende Verdopplung einer $lcom$-Reduktion durch eine $(no, llet)$-Reduktion. Eine Einschränkung der Redex-Positionen auf die Menge aller R -Kontexte führt nicht zu einem Verschwinden dieser Verdopplung.

Lemma 3.4.17. *Die lcom-Reduktion verfügt über den folgenden vollständigen Satz von Gabeldiagrammen:*

i) $\xleftarrow{no,a} \circ \xrightarrow{lcom} \;\rightsquigarrow\; \xrightarrow{lcom} \circ \xleftarrow{no,a}$ *für* $a \in \{llet, lapp, lbeta, cp, nd\}$

ii) $\xleftarrow{no,cp} \circ \xrightarrow{lcom} \;\rightsquigarrow\; \xrightarrow{lcom} \circ \xrightarrow{lcom} \circ \xleftarrow{no,cp}$

iii) $\xleftarrow{no,llet} \circ \xleftarrow{no,lapp,\omega} \circ \xleftarrow{no,llet} \circ \xleftarrow{no,lapp,\omega} \circ \xrightarrow{lcom} \;\rightsquigarrow$
$\xrightarrow{lcom} \circ \xleftarrow{no,llet} \circ \xleftarrow{no,lapp,\omega} \circ \xleftarrow{no,llet} \circ \xleftarrow{no,lapp,\omega}$ *wobei* $\omega > 0$

iv) $\xleftarrow{no,lapp,\omega} \circ \xrightarrow{lcom} \;\rightsquigarrow\; \xrightarrow{lcom} \circ \xleftarrow{no,lapp,\omega}$ *wobei* $\omega > 0$

v) $\xleftarrow{no,llet} \circ \xrightarrow{lcom} \;\rightsquigarrow\; \xrightarrow{lcom} \circ \xrightarrow{lcom} \circ \xleftarrow{no,llet}$

vi) $\xleftarrow{no,llet} \circ \xleftarrow{no,llet} \circ \xrightarrow{lcom} \;\rightsquigarrow\; \xrightarrow{lcom} \circ \xleftarrow{no,llet} \circ \xleftarrow{no,llet}$

Beweis. Um alle Gabeldiagramme zu erhalten ist es wie bei den zugehörigen Beweisen bei anderen Reduktionsarten erforderlich alle Überlappungsmöglichkeiten eines no-Redex und eines $lcom$-Redex zu untersuchen.
Es entsteht immer eine Gabeldiagramm der Form i), abgesehen von folgenden Ausnahmen:

ii) Der $lcom$-Redex befindet sich innerhalb des kopierbaren Subausdrucks des (no, cp)-Redex:

$$\begin{tikzcd}
L_R^*[\texttt{let } x = t_{lcom} \texttt{ in } W[x]] \arrow[rr, "lcom"] \arrow[d, "{no,cp}"] & & L_R^*[\texttt{let } x = t'_{lcom} \texttt{ in } W[x]] \arrow[d, "{no,cp}"] \\
L_R^*[\texttt{let } x = t_{lcom} \texttt{ in } W[t_{lcom}]] \arrow[r, "lcom"] & \circ \arrow[r, "lcom"] & L_R^*[\texttt{let } x = t'_{lcom} \texttt{ in } W[t'_{lcom}]]
\end{tikzcd}$$

Bei obigem Diagramm beinhalte der Ausdruck t_{lcom} einen $lcom$-Redex.

iii) Die $lcom$-Reduktion verändert den zu einem $(no, lapp)$-Redex gehörigen Let-Ausdruck und der W-Kontext des $(no, lapp)$-Redex beinhaltet einen L_L-Kontext:

$$\begin{tikzcd}
L_R^*[L_L[A_L^n[(\texttt{let } x = t_x \texttt{ in let } y = t_y \texttt{ in } s)t]]] \arrow[r, "\mathit{lcom}"] \arrow[d, "\mathit{no,lapp}"] & L_R^*[L_L[A_L^n[(\texttt{let } y = t_y \texttt{ in let } x = t_x \texttt{ in } s)t]]] \arrow[d, "\mathit{no,lapp}"] \\
L_R^*[L_L[A_L^n[\texttt{let } x = t_x \texttt{ in } ((\texttt{let } y = t_y \texttt{ in } s)t)]]] \arrow[d, "\mathit{no,lapp,n}"] & L_R^*[L_L[A_L^n[\texttt{let } y = t_y \texttt{ in } ((\texttt{let } x = t_x \texttt{ in } s)t)]]] \arrow[d, "\mathit{no,lapp,n}"] \\
L_R^*[L_L[\texttt{let } x = t_x \texttt{ in } A_L^n[(\texttt{let } y = t_y \texttt{ in } s)t]]] \arrow[d, "\mathit{no,llet}"] & L_R^*[L_L[\texttt{let } y = t_y \texttt{ in } A_L^n[(\texttt{let } x = t_x \texttt{ in } s)t]]] \arrow[d, "\mathit{no,llet}"] \\
L_R^*[\texttt{let } x = t_x \texttt{ in } L_L[A_L^n[(\texttt{let } y = t_y \texttt{ in } s)t]]] \arrow[d, "\mathit{no,lapp,n+1}"] & L_R^*[\texttt{let } y = t_y \texttt{ in } L_L[A_L^n[(\texttt{let } x = t_x \texttt{ in } s)t]]] \arrow[d, "\mathit{no,lapp,n+1}"] \\
L_R^*[\texttt{let } x = t_x \texttt{ in } L_L[\texttt{let } y = t_y \texttt{ in } A_L^n[st]]] \arrow[d, "\mathit{no,llet}"] & L_R^*[\texttt{let } y = t_y \texttt{ in } L_L[\texttt{let } x = t_x \texttt{ in } A_L^n[st]]] \arrow[d, "\mathit{no,llet}"] \\
L_R^*[\texttt{let } x = t_x \texttt{ in let } y = t_y \texttt{ in } L_L[A_L^n[st]]] \arrow[r, "\mathit{lcom}"] & L_R^*[\texttt{let } y = t_y \texttt{ in let } x = t_x \texttt{ in } L_L[A_L^n[st]]]
\end{tikzcd}$$

iv) Die *lcom*-Reduktion verändert den zu einem $(no, lapp)$-Redex gehörigen Let-Ausdruck jedoch beinhaltet der W-Kontext des $(no, lapp)$-Redex keinen L_L-Kontext:

$$\begin{tikzcd}
L_R^*[A_L^n[(\texttt{let } x = t_x \texttt{ in let } y = t_y \texttt{ in } s)t]] \arrow[r, "\mathit{lcom}"] \arrow[d, "\mathit{no,lapp}"] & L_R^*[A_L^n[(\texttt{let } y = t_y \texttt{ in let } x = t_x \texttt{ in } s)t]] \arrow[d, "\mathit{no,lapp}"] \\
L_R^*[A_L^n[\texttt{let } x = t_x \texttt{ in } ((\texttt{let } y = t_y \texttt{ in } s)t)]] \arrow[d, "\mathit{no,lapp,n}"] & L_R^*[A_L^n[\texttt{let } y = t_y \texttt{ in } ((\texttt{let } x = t_x \texttt{ in } s)t)]] \arrow[d, "\mathit{no,lapp,n}"] \\
L_R^*[\texttt{let } x = t_x \texttt{ in } A_L^n[(\texttt{let } y = t_y \texttt{ in } s)t]] \arrow[d, "\mathit{no,lapp,n+1}"] & L_R^*[\texttt{let } y = t_y \texttt{ in } A_L^n[(\texttt{let } x = t_x \texttt{ in } s)t]] \arrow[d, "\mathit{no,lapp,n+1}"] \\
L_R^*[\texttt{let } x = t_x \texttt{ in let } y = t_y \texttt{ in } A_L^n[st]] \arrow[r, "\mathit{lcom}"] & L_R^*[\texttt{let } y = t_y \texttt{ in let } x = t_x \texttt{ in } A_L^n[st]]
\end{tikzcd}$$

v) Die *lcom*-Reduktion bewegt einen Let-Ausdruck aus dem umgebenden L_R^*-Kontext eines $(no, llet)$-Redex in den $(no, llet)$-Redex hinein:

$$\begin{tikzcd}
L_R^*[\texttt{let } x = t_x \texttt{ in let } y = (\texttt{let } z = t_z \texttt{ in } t_y) \texttt{ in } W[y]] \arrow[r, "\mathit{lcom}"] \arrow[d, "\mathit{no,llet}"] & L_R^*[\texttt{let } y = (\texttt{let } z = t_z \texttt{ in } t_y) \texttt{ in let } x = t_x \texttt{ in } W[y]] \arrow[d, "\mathit{no,llet}"] \\
L_R^*[\texttt{let } x = t_x \texttt{ in } (\texttt{let } z = t_z \texttt{ in } (\texttt{let } y = t_y \texttt{ in } W[y]))] \arrow[r, "\mathit{lcom,2}"] & L_R^*[\texttt{let } z = t_z \texttt{ in let } y = t_y \texttt{ in let } x = t_x \texttt{ in } W[y]]
\end{tikzcd}$$

vi) Die *lcom*-Reduktion verändert den inneren Let-Ausdruck eines $(no, llet)$-Redex:

$$\begin{tikzcd}
L_R^*[\texttt{let } x = (\texttt{let } y = t_y \texttt{ in let } z = t_z \texttt{ in } t) \texttt{ in } W[x]] \arrow[r, "\mathit{lcom}"] \arrow[d, "\mathit{no,llet}"] & L_R^*[\texttt{let } x = (\texttt{let } z = t_z \texttt{ in let } y = t_y \texttt{ in } t) \texttt{ in } W[x]] \arrow[d, "\mathit{no,llet}"] \\
L_R^*[\texttt{let } y = t_y \texttt{ in let } x = (\texttt{let } z = t_z \texttt{ in } t) \texttt{ in } W[x]] \arrow[d, "\mathit{no,llet}"] & L_R^*[\texttt{let } z = t_z \texttt{ in let } x = (\texttt{let } y = t_y \texttt{ in } t) \texttt{ in } W[x]] \arrow[d, "\mathit{no,llet}"] \\
L_R^*[\texttt{let } y = t_y \texttt{ in let } z = t_z \texttt{ in let } x = t_x \texttt{ in } W[x]] \arrow[r, "\mathit{lcom}"] & L_R^*[\texttt{let } z = t_z \texttt{ in let } y = t_y \texttt{ in let } x = t_x \texttt{ in } W[x]]
\end{tikzcd}$$

□

Wir benutzen die Gabeldiagramme um einen vollständigen Satz von Vertauschungsdiagrammen zu entwickeln.

Lemma 3.4.18. *Bei Anwendung einer lcom-Reduktion können folgende Vertauschungsdiagramme vorkommen:*

i) $lcom \circ (no, a) \rightsquigarrow (no, a) \circ lcom$ *für* $a \in \{llet, lapp, lbeta, cp, nd\}$

ii) $lcom \circ (no, cp) \rightsquigarrow (no, cp) \circ lcom \circ lcom$

iii) $lcom \circ (no, lapp, \omega) \circ (no, llet) \circ (no, lapp, \omega) \circ (no, llet) \rightsquigarrow$
$(no, lapp, \omega) \circ (no, llet) \circ (no, lapp, \omega) \circ (no, llet) \circ lcom$ *wobei* $\omega > 0$

iv) $lcom \circ (no, lapp, \omega) \rightsquigarrow (no, lapp, \omega) \circ lcom$ *wobei* $\omega > 0$

v) $lcom \circ (no, llet) \rightsquigarrow (no, llet) \circ lcom \circ lcom$

vi) $lcom \circ (no, llet) \circ (no, llet) \rightsquigarrow (no, llet) \circ (no, llet) \circ lcom$

Beweis. Jedes der Gabeldiagramme der *lcom*-Reduktion aus Lemma 3.4.17 kann gemäß der in Abschnitt 2.4.2 beschriebenen Methodik direkt in ein Vertauschungsdiagramm überführt werden. Ein einfacher Abgleich zeigt, daß jedes der oben aufgeführten Vertauschungsdiagramme aus einem zugehörigen Gabeldiagramm hergeleitet wurde. ☐

Im Anschluß an die Vertauschungsdiagramme wird die wechselseitige Erhaltung einer WHNF gezeigt.

Lemma 3.4.19. *Gegeben zwei geschlossene Ausdrücke* t *und* t'. *Wenn* $t \xrightarrow{lcom} t'$, *dann gilt* t *in WHNF* $\Leftrightarrow$ t' *in WHNF.*

Beweis. Aus den Gabeldiagrammen der *lcom*-Reduktion folgt, daß t' einen no-Redex besitzt, wenn t einen no-Redex besitzt. Umgekehrt folgt aus den Vertauschungsdiagrammen, daß t einen no-Redex besitzt, wenn t' einen no-Redex besitzt. Ein Ausdruck befindet sich per Definition in WHNF gdw. er keinen no-Redex besitzt. Somit folgt aus der logischen Umkehrung beider Implikationen sofort die Hypothese. ☐

In den nun folgenden zwei Lemmata wird die wechselseitige Übertragung einer no-Reduktion zu WHNF gezeigt.

Lemma 3.4.20. *Gegeben zwei Ausdrücke* t, s *mit* $t \xrightarrow{lcom} s$ *sowie ein Ausdruck* t_{WF} *in WHNF mit* $t \xrightarrow{no,m} t_{WF}$, *wobei* m *die Anzahl einzelner Reduktionen bezeichnet, über die die Reduktion von* t *zu* t_{WF} *verfügt. Dann existiert ein Ausdruck* s_{WF} *in WHNF, so daß* $s \xrightarrow{no,m} s_{WF}$ *gilt.*

Diagramm 1

$$\begin{tikzcd}
t \arrow[r, two heads, "lcom"] \arrow[d, "{no,m}"'] \arrow[dr, phantom, "\text{Lemma 3.4.17} \; ①"] & s \arrow[d, dashed, "{no,m}"] \\
t' \arrow[r, dashed, two heads, "lcom"'] \arrow[d, "{no,n}"'] \arrow[dr, phantom, "\text{Ind. Voraus.} \; ②"] & \circ \arrow[d, dashed, "{no,n}"] \\
t_{WF} \arrow[r, dotted, no head] & s_{WF}
\end{tikzcd}$$

Diagramm 2

$$\begin{tikzcd}
t \arrow[rr, two heads, "lcom"] \arrow[d, "{no,a}"'] \arrow[drr, phantom, "\text{Lemma 3.4.17} \; ①"] & & s \arrow[d, dashed, "{no,a}"] \\
t' \arrow[r, dashed, two heads, "lcom"] \arrow[d, "{no,n}"'] \arrow[dr, phantom, "②"] & t'' \arrow[r, dashed, two heads, "lcom"] \arrow[d, dashed, shift right, "{no,n}"'] \arrow[d, shift left] \arrow[dr, phantom, "③"] & \circ \arrow[d, dashed, "{no,n}"] \\
t_{WF} \arrow[r, dotted, no head] & t'_{WF} \arrow[r, dotted, no head] & s_{WF} \\
& \text{Induktionsvoraussetzung} &
\end{tikzcd}$$

wobei $a \in \{llet, cp\}$
wobei $m > 0, n \geq 0$

Abbildung 3.9:

Beweis. Wir führen eine Induktion über die Länge der no-Reduktion von t zu t_{WF} und benutzen als Induktionshypothese die Hypothese des Lemmas.

Induktionsbeginn:
Die Länge der no-Reduktion von t zu t_{WF} beträgt 0, d.h. es gilt $t \equiv t_{WF}$ bzw. t in WHNF. Aus t in WHNF folgt gemäß Lemma 3.4.19 s in WHNF. Indem wir $s' \equiv s$ wählen haben wir die Hypothese für diesen Fall gezeigt.

Induktionsschritt:
Wir bilden eine Zerlegung $t \xrightarrow{no,+} t' \xrightarrow{no,*} t_{WF}$, so daß die entstehende Gabel $t' \xleftarrow{no,+} t \xrightarrow{lcom} s$ mittels eines Gabeldiagramms der *lcom*-Reduktion aus Lemma 3.4.17 geschlossen werden kann. Abbildung 3.9 enthält zwei Reduktionsdiagramme, aus denen der Beweis der Hypothese für alle Gabeldiagramme der *lcom*-Reduktion abgelesen werden kann. Dabei ist Diagramm 2 bei den Gabeldiagrammen *ii*) und *v*) aus Lemma 3.4.17 zu benutzen. In allen anderen Fällen ist der Beweis aus Diagramm 1 ablesbar.
Die von t' ausgehende no-Reduktion zu t_{WF} ist kürzer als die von t ausgehende, somit können wir bei t' die Induktionshypothese anwenden. Bei Diagramm 2 erfolgt bei t'' eine zweite Anwendung der Induktionshypothese. Dies ist zulässig, da aus der ersten Anwendung der Induktionshypothese mitunter folgt, daß die Länge der no-Reduktion von t'' zu t'_{WF} der Länge der no-Reduktion von t' zu t_{WF} entspricht.
Alle Gabeldiagramme der *lcom*-Reduktion verfügen über die Eigenschaft, auf der linken Seite genauso viele einzelne no-Reduktionen aufzuweisen, wie auf der rechten Seite. Daraus folgt die in der Hypothese enthaltene Erhaltung der Länge der no-Reduktion. □

Was zuletzt für die von t ausgehenden no-Reduktionen zu WHNF gezeigt wurde, werden wir jetzt für eine von s ausgehende no-Reduktion zu WHNF zeigen.

Diagramm 1

$$\begin{tikzcd}
t \arrow[r, two heads, "lcom"] \arrow[d, dashed, "{no,m}"'] \arrow[dr, phantom, "\text{Lemma 3.4.18 } ①"] & s \arrow[d, "{no,m}"] \\
\circ \arrow[r, dashed, two heads, "lcom"'] \arrow[d, dashed, "{no,n}"'] \arrow[dr, phantom, "\text{Ind. Voraus. } ②"] & s' \arrow[d, "{no,n}"] \\
t_{WF} \arrow[r, dotted, no head] & s_{WF}
\end{tikzcd}$$

Diagramm 2

$$\begin{tikzcd}
t \arrow[rr, two heads, "lcom"] \arrow[d, dashed, "{no,a}"'] \arrow[drr, phantom, "\text{Lemma 3.4.18 } ①"] & & s \arrow[d, "{no,a}"] \\
\circ \arrow[r, dashed, two heads, "lcom"] \arrow[d, dashed, "{no,n}"'] \arrow[dr, phantom, "③"] & \circ \arrow[r, dashed, two heads, "lcom"] \arrow[d, "{no,n}"'] \arrow[d, dashed, shift left] \arrow[dr, phantom, "②"] & s' \arrow[d, "{no,n}"] \\
t_{WF} \arrow[r, dotted, no head] & t'_{WF} \arrow[r, dotted, no head] & s_{WF}
\end{tikzcd}$$

Induktionsvoraussetzung

wobei $a \in \{llet, cp\}$
wobei $m > 0, n \geq 0$

Abbildung 3.10:

Lemma 3.4.21. *Gegeben zwei Ausdrücke t, s mit $t \xrightarrow{lcom} s$ sowie ein Ausdruck s_{WF} in WHNF mit $s \xrightarrow{no,m} s_{WF}$, wobei m die Anzahl einzelner Reduktionen bezeichnet, über die die Reduktion von s zu s_{WF} verfügt. Dann existiert ein Ausdruck t_{WF} in WHNF, so daß $t \xrightarrow{no,m} t_{WF}$ gilt.*

Beweis. Der Beweis erfolgt nach demselben Muster wie bei dem Lemma zuvor, nur geht die Argumentation diesmal von s aus und es werden die Vertauschungsdiagramme der *lcom*-Reduktion anstelle der Gabeldiagramme benutzt. Es wird eine Induktion über die Länge der no-Reduktion von s zu s_{WF} geführt und als Induktionshypothese die Hypothese des Lemmas benutzt. Die no-Reduktion von s zu s_{WF} wird derart zerlegt, daß eines der Vertauschungsdiagramme aus Lemma 3.4.18 paßt. Abbildung 3.10 enthält zwei Reduktionsdiagramme, aus denen die induktive Beweisführung für alle Vertauschungsdiagramme der *lcom*-Reduktion abgelesen werden kann. □

Nachdem wir die wechselseitige Übertragbarkeit einer no-Reduktion zu WHNF gezeigt haben, werden wir nun die wechselseitige Übertragbarkeit der Endlichkeit aller no-Reduktionen zeigen. Der Zweck diesen Schrittes wurde bereits bei der *lcv*-Reduktion erläutert.

Lemma 3.4.22. *Gegeben zwei geschlossene Ausdrücke t, s mit $t \xrightarrow{lcom} s$. Dann gilt $t\not\Uparrow \Rightarrow s\not\Uparrow$.*

Beweis. Wir führen eine Induktion über die längste von t ausgehende no-Reduktion und benutzen als Induktionshypothese die Hypothese des Lemmas.

Induktionsbeginn:
Die längste von t ausgehende no-Reduktion hat die Länge 0, d.h. t befindet sich in WHNF. Daraus folgt gemäß Lemma 3.4.19, daß sich auch s in WHNF befindet. Wir wählen $s' \equiv s$ und haben die Hypothese für diesen Fall gezeigt.

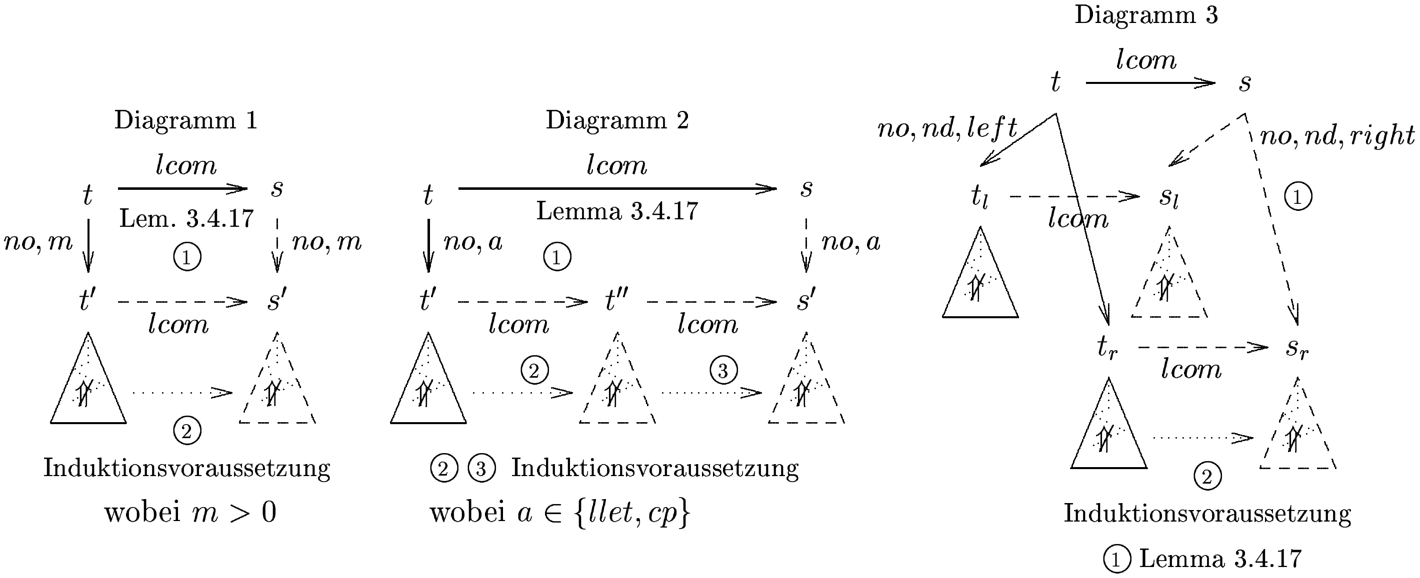


Abbildung 3.11:

Induktionsschritt:
Wir differenzieren nach dem no-Redex von t:

1. Fall: Der no-Redex ist vom Typ *llet*, *lapp*, *lbeta*, *cp*
Wir spalten einen Teil der von t ausgehenden no-Reduktion ab, derart, daß wir eines der Gabeldiagramme der *lcom*-Reduktion aus Lemma 3.4.17 schließen können. In diesem Zusammenhang ist zu beachten, daß keines der Gabeldiagramme, das zu einem no-Redex des Typs *llet*, *lapp*, *lbeta* oder *cp* paßt, über eine *nd*-Reduktion verfügt, der abgespaltene Teil somit keine *nd*-Reduktionen beinhaltet. Die Diagramme 1 und 2 aus Abbildung 3.11 zeigen die weitere Argumentation auf. Dabei ist im Falle der Gabeldiagramme *ii*) sowie *v*) Diagramm 2 zu benutzen, in allen anderen Fällen Diagramm 1.
Bei t' kann stets die Induktionshypothese angewendet werden, da die längste von t' ausgehende no-Reduktion kürzer sein muß als die Längste von t ausgehende. Bei Diagramm 2 wird bei t'' ein weiteres Mal die Induktionsvoraussetzung angewendet. Die zweite Anwendung ist zulässig, da die Annahme, daß ausgehend von t'' eine no-Reduktion existiert, die länger ist, als die längste von t' ausgehende no-Reduktion, in Verbindung mit dem in Lemma 3.4.21 beinhalteten Erhaltungsgesetz bezüglich der Länge der no-Reduktionen sofort einen Widerspruch erzeugt. Die no-Reduktion von s zu s' kann bei den betrachteten Fällen keine *nd*-Reduktionen beinhalten. Da gemäß Induktionsvoraussetzung alle von s' ausgehenden no-Reduktionen endlich sind, sind dies folglich auch alle von s ausgehenden.

2. Fall: Der no-Redex ist vom Typ *nd*
In diesem Fall ist die von t ausgehende no-Reduktion nicht eindeutig. Wir verfolgen beide Alternativen der *nd*-Reduktion nachdem selben Muster wie beim 1.

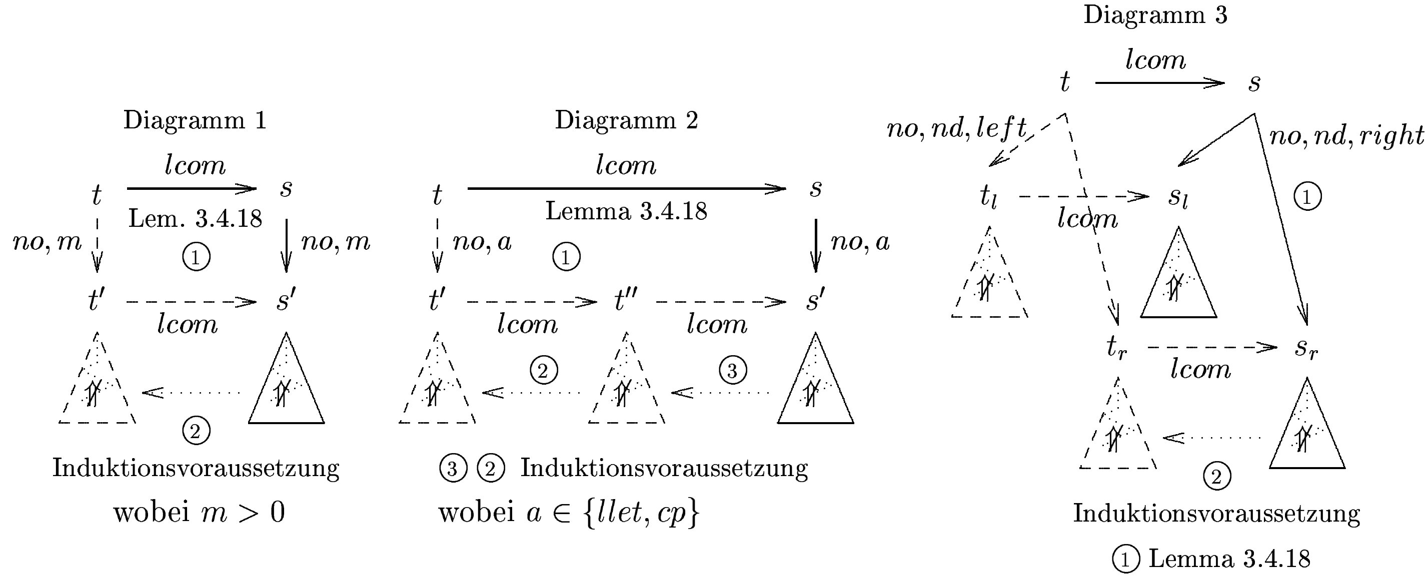


Abbildung 3.12:

Fall. Diagramm 3 aus Abbildung 3.11 zeigt die entstehenden Reduktionszusammenhänge. Da alle von s_l und s_r ausgehenden no-Reduktionen gemäß Induktionsvoraussetzung endlich sind und von s aus keinen weiteren Ausdrücke außer s_l und s_r erreichbar sind, müssen folglich auch alle von s ausgehenden no-Reduktionen endlich sein. □

Die Übertragbarkeit der Endlichkeit aller no-Reduktionen zeigen wir nun in gleicher Weise umgekehrt von s ausgehend.

Lemma 3.4.23. *Gegeben zwei geschlossene Ausdrücke t, s mit $t \xrightarrow{lcom} s$. Dann gilt $s\not\Uparrow \Rightarrow t\not\Uparrow$.*

Beweis. Der Beweis erfolgt simultan zu dem des Lemmas zuvor. Es wird eine Induktion über die Länge der längsten von s ausgehenden no-Reduktion geführt und als Induktionshypothese die Hypothese des Lemmas verwendet. Statt der Gabeldiagramme werden die Vertauschungsdiagramme der *lcom*-Reduktion aus Lemma 3.4.18 verwendet. Abbildung 3.10 enthält drei Reduktionsdiagramme, aus denen die induktive Beweisführung für alle Fälle abgelesen werden kann, die in Abhängigkeit von den verschiedenen Vertauschungsdiagrammen entstehen können. □

Aus den vier zuvor bewiesenen Lemmata werden wir nun die Erhaltung der kontextuellen Äquivalenz bei Anwendung einer *lcom*-Reduktion folgern, wie wir dies auch bei der Betrachtung der *lcv*-Reduktion im Abschnitt zuvor getan haben.

Satz 3.4.3. *Gegeben zwei Ausdrücke t_1 und t_2. Wenn $t_1 \xrightarrow{lcom} t_2$, dann $t_1 \sim_c t_2$.*

Beweis. Die Hypothese folgt wegen der Kompatibilität der *lcom*-Reduktion direkt aus Proposition 3.1.2 in Kombination mit den vier Lemmata 3.4.20, 3.4.21, 3.4.22, 3.4.23 und der folgenden einfachen Überlegung: Aus $(s\not\Uparrow \Rightarrow \ t\not\Uparrow)$ folgt $(t\Uparrow \Rightarrow \ s\Uparrow)$ durch logische Umkehrung, bzw. Widerspruch. □

3.4.5 kontextuelle Äquivalenz der *ucp*-Reduktion

Wir werden nun die Erhaltung der kontextuellen Äquivalenz durch eine *ucp*-Reduktion zeigen. Der Name *ucp* steht für "unique copy". Die *ucp*-Reduktion ermöglicht es, eindeutig gebundene Subausdrücke, deren bindende Variable sich nicht unterhalb einer Abstraktion befindet kopieren zu können. Die *ucp*-Reduktion wird in Kapitel 6, wo eine abstrakte Maschine zur maschinellen Nachahmung der no-Reduktion des Λ_{let}-Kalküls vorgestellt wird, intensiv genutzt werden.
Der Beweis der kontextuellen Äquivalenz der *ucp*-Reduktion, erfordert eine gewisse Vorarbeit. Wir werden damit beginnen, die Eigenschaften einer $(ldel, L_R^*[\cdot])$-Reduktion zu untersuchen, und zeigen, daß eine $(ldel, L_R^*[\cdot])$-Reduktion im Gegensatz zu einer *ldel*-Reduktion bei ihrer Anwendung die Anzahl einzelner no-Reduktionen bis zum Erreichen einer WHNF vergleichbar der *lcom*-Reduktion gleichhält. Danach werden wir *lcom*- und $(ldel, L_R^*[\cdot])$-Reduktionen zu einer einzelnen Reduktion bezeichnet mit *lcld* zusammenfassen und einige Eigenschaften dieser Reduktion zeigen. Letzteres geschieht um die späteren Beweise transparenter gestalten zu können. Bevor wir dann die Erhaltung der kontextuelle Äquivalenz durch eine *ucp*-Reduktion zeigen werden, werden wir noch eine vereinfachte Form der *ucp*-Reduktion mit dem Namen *ucpw* einführen. Auch dies geschieht zu dem Zweck, die späteren Beweise transparenter zu gestalten

Eigenschaften der $(ldel, L_R^*[\cdot])$-Reduktion

Die *ldel*-Reduktion verfügt über die negative Eigenschaft, daß ihre Anwendung zu einer Verkürzung der no-Reduktion führen kann, was eine induktive Argumentation erschwert. Ein Blick in den Beweis von Lemma 3.4.1, wo die Gabeldiagramme der *ldel*-Reduktion vorgestellt werden, zeigt, daß dies der Fall ist, wenn *ldel* eine let-Bindung eines *llet*- oder *lapp*-Redex löscht. Im Kontext des Beweises der kontextuellen Äquivalenz der *ucp*-Reduktion werden wir diesen speziellen Fall jedoch nicht benötigen und können mit einer eingeschränkten Form der *ldel*-Reduktion operieren. Wir schränken die *ldel*-Reduktion auf die $(ldel, L_R^*[\cdot])$-Reduktion ein, und werden diese nun untersuchen.

Lemma 3.4.24. *Der vollständige Satz von Gabeldiagrammen der* $(ldel, L_R^*[\cdot])$*-Reduktion lautet:*

$$\xleftarrow{no,a} \circ \xrightarrow{ldel,L_R^*[\cdot]} \quad \rightsquigarrow \quad \xrightarrow{ldel,L_R^*[\cdot]} \circ \xleftarrow{no,a} \quad \textit{für } a \in \{llet, lapp, lbeta, cp, nd\}$$

Beweis. Der vollständige Satz von Gabeldiagrammen der $(ldel, L_R^*[\cdot])$-Reduktion ist eine Untermenge des in Lemma 3.4.1 vorgestellten vollständigen Satzes von Gabeldiagrammen der *ldel*-Reduktion. Die mit *ii*) und *iii*) gekennzeichneten Diagramme aus Lemma 3.4.1 sind bei der Einschränkung auf einen L_R^*-Kontext nicht möglich, da bei diesen nicht innerhalb eines L_R^*-Kontexts reduziert wird.
Bleibt in einem zweiten Schritt zu zeigen, daß die Reduzierung innerhalb eines L_R^*-Kontexts gewährleistet bleibt. Dies zeigt eine einfache Überprüfung aller möglichen Typen von no-Redizes. Wir betrachten stellvertretend den Fall eines $(no, llet)$-Redex, bei allen anderen Typen ist eine Argumentation nach demselben Schema möglich.
Eine $(no, llet)$-Reduktion hat die Form

$$\begin{array}{ll} & L_R^*[\texttt{let } x = (\texttt{let } y = t_y \texttt{ in } t_x) \texttt{ in } W[x]] \\ \xrightarrow{no,llet} & L_R^*[\texttt{let } y = t_y \texttt{ in let } x = t_x \texttt{ in } W[x]] \end{array}$$

Ein *ldel*-Redex innerhalb des Subausdrucks `let` $y = t_y$ `in` t_x ist nicht möglich, da dieser sich innerhalb eines L_L-Kontext befinden würde. Ein *ldel*-Redex innerhalb des L_R^*-Kontexts oder innerhalb von $W[x]$ ist unproblematisch, da dieser sich auch nach Anwendung der $(no, llet)$-Reduktion wieder in einem L_R^*-Kontext befinden würde. □

Wir werden neben den Gabeldiagrammen auch noch die Vertauschungsdiagramme benötigen.

Lemma 3.4.25. *Der vollständige Satz von Vertauschungsdiagrammen der* $(ldel, L_R^*[\cdot])$*-Reduktion lautet:*

$$(ldel, L_R^*[\cdot]) \circ (no, a) \ \rightsquigarrow \ (no, a) \circ (ldel, L_R^*[\cdot]) \ \textit{für } a \in \{llet, lapp, lbeta, cp, nd\}$$

Beweis. Eine einfache Herleitung aus dem Satz von Gabeldiagrammen, wie in Abschnitt 2.4.2 beschrieben. □

Wir bilden nun zwei einfache Erweiterungen der zuvor vorgestellten Lemmata, indem wir die no-Reduktion auf beliebige Länge ausdehnen.

Lemma 3.4.26. *Gegeben zwei Ausdrücke* t *und* s *mit* $t \xrightarrow{ldel,L_R^*[\cdot]} s$ *sowie ein Ausdruck* t' *mit* $t \xrightarrow{no,m} t'$ *und* $m \geq 0$*. Dann existiert ein Ausdruck* s'*, so daß die Reduktionszusammenhänge des untenstehenden Diagramms erfüllt werden:*

$$\begin{tikzcd}
t \arrow[r, "{ldel, L_R^*[\cdot]}"] \arrow[d, "{no,m}"'] & s \arrow[d, dashed, "{no,m}"] \\
t' \arrow[r, dashed, "{ldel, L_R^*[\cdot]}"'] & s'
\end{tikzcd}$$

Beweis. Eine einfache Induktion über die Länge m der von t ausgehenden no-Reduktion unter Zuhilfenahme des vollständigen Satzes von Gabeldiagrammen aus Lemma 3.4.24. ☐

Was wir für die Vergabelung gezeigt haben, zeigen wir jetzt genauso für die Vertauschung.

Lemma 3.4.27. *Gegeben zwei Ausdrücke t und s mit $t \xrightarrow{ldel,L_R^*[\cdot]} s$ sowie ein Ausdruck s' mit $s \xrightarrow{no,m} s'$ und $m \geq 0$. Dann existiert ein Ausdruck s', so daß die Reduktionszusammenhänge des untenstehenden Diagramms erfüllt werden:*

$$\begin{tikzcd}
t \arrow[r, "{ldel, L_R^*[\cdot]}"] \arrow[d, dashed, "{no,m}"'] & s \arrow[d, "{no,m}"] \\
t' \arrow[r, dashed, "{ldel, L_R^*[\cdot]}"'] & s'
\end{tikzcd}$$

Beweis. Eine einfache Induktion über die Länge m der von s ausgehenden no-Reduktion unter Zuhilfenahme des vollständigen Satzes von Vertauschungsdiagrammen aus Lemma 3.4.25. ☐

Von großem Wert im Kontext späterer Beweise wird die Erhaltung der Länge der no-Reduktion bei den beiden zuletzt gezeigten Lemmata sein.

Die *lcld*-Reduktion und ihre Eigenschaften

Die nun vorgestellte *lcld*-Reduktion ist keine vollständig neue Reduktion, es handelt sich vielmehr um eine Kombination von $(ldel, L_R^*[\cdot])$- und $(lcom, *)$-Reduktionen, wie wir sie beim Beweis der Erhaltung der kontextuellen Äquivalenz durch eine *ucp*-Reduktion häufig benötigt werden. Die *lcld*-Reduktion wird wie folgt formal definiert:

Definition 3.4.3. *Gegeben zwei Ausdrücke t und t'. Wir definieren:*

$t \xrightarrow{lcld} t'$ gdw. zwei Ausdrücke s und s' existieren, so daß $t \xrightarrow{ldel,L_R^*[\cdot]} s$, $s \xrightarrow{lcom,*} s'$ und $t' \xrightarrow{ldel,L_R^*[\cdot]} s'$ gilt.

$$\begin{tikzcd}
t \arrow[r, "{ldel, L^*_R[\cdot]}"] \arrow[d, dashed, "{\begin{array}{l} no, m \\ \text{Lemma 3.4.27} \\ \text{⑤} \end{array}}"] & t' \arrow[r, "{lcom, *}"] \arrow[d] \arrow[d, dashed, shift left=2, "{\begin{array}{l} no, m \\ \text{Lemma 3.4.23} \\ \text{③} \end{array}}"] & s' \arrow[d] \arrow[d, dashed, shift left=2, "{\begin{array}{l} no, m \\ \text{Lemma 3.4.26} \\ \text{①} \end{array}}"] & s \arrow[l, "{ldel, L^*_R[\cdot]}"'] \arrow[d, "{no, m}"] \\
t_{WF} \arrow[r, dashed, "{\begin{array}{c} ldel, L^*_R[\cdot] \\ \text{Lemma 3.4.3} \\ \text{⑥} \end{array}}"'] & t'_{WF} \arrow[r, dashed, "{\begin{array}{c} lcom, * \\ \text{Lemma 3.4.19} \\ \text{④} \end{array}}"'] & s'_{WF} & s_{WF} \arrow[l, dashed, "{\begin{array}{c} ldel, L^*_R[\cdot] \\ \text{Lemma 3.4.3} \\ \text{②} \end{array}}"]
\end{tikzcd}$$

Abbildung 3.13:

Die *lcld*-Reduktion werden wir beim Beweis der kontextuellen Äquivalenz der *ucp*-Reduktion benötigen, um eine Übertragbarkeit der no-Reduktion bei gleichbleibender Anzahl an no-Schritten zeigen zu können. Es sei als Randnotiz angemerkt, daß die *lcld*-Reduktion nicht kompatibel sein kann, da die $(ldel, L^*_R[\cdot])$-Reduktion nicht kompatibel ist.

Wir werden nun die wechselseitige Übertragbarkeit einer no-Reduktion zu WHNF über eine *lcld*-Reduktion hinweg zeigen.

Lemma 3.4.28. *Gegeben zwei Ausdrücke t und s mit $t \xrightarrow{lcld} s$ sowie ein Ausdruck s_{WF} in WHNF mit $s \xrightarrow{no,m} s_{WF}$ und $m \geq 0$. Dann existiert ein Ausdruck t_{WF} in WHNF, so daß die folgenden Reduktionszusammenhänge erfüllt werden:*

$$\begin{tikzcd}
t \arrow[r, "lcld"] \arrow[d, dashed, "{no, m}"'] & s \arrow[d, "{no, m}"] \\
t_{WF} \arrow[r, dash, dotted] & s_{WF}
\end{tikzcd}$$

Beweis. Wir zeigen die Hypothese, indem wir den in Abbildung 3.13 gezeigten Reduktionszusammenhängen entsprechend der Nummerierung folgen.

Wir zerlegen zunächst die *lcld*-Reduktion in ihre einzelnen Bestandteile und führen die Argumentation dann über die einzelnen Bestandteile hinweg. Wir übertragen zunächst mittels Lemma 3.4.26 die von s ausgehende no-Reduktion nach s', wobei die Länge der no-Reduktion dabei unverändert bleibt. Gemäß der in Lemma 3.4.3 formulierten Erhaltung einer WHNF durch eine *ldel*-Reduktion muß sich s'_{WF} wie s_{WF} in WHNF befinden. Im Anschluß benutzen wir Lemma 3.4.23 in transitiv -reflexiver Form, um die no-Reduktion nach t' zu übertragen. Dabei bleibt wiederum die Länge der no-Reduktion unverändert. t'_{WF} muß sich gemäß der in Lemma 3.4.19 gezeigten Erhaltung einer WHNF durch *lcom*-Reduktionen wie s'_{WF} in WHNF befinden. Mittels Lemma 3.4.27 übertragen wir die von t'

$$\begin{tikzcd}
t \arrow[r, "{ldel, L_R^*[\cdot]}"] \arrow[d] \arrow[dr, phantom, "\substack{no, m \\ \text{Lemma 3.4.26} \\ \text{①}}"] & t' \arrow[r, "{lcom, *}"] \arrow[d, dashed] \arrow[d, shift left] \arrow[dr, phantom, "\substack{no, m \\ \text{Lemma 3.4.22} \\ \text{③}}"] & s' \arrow[d, dashed] \arrow[d, shift left] \arrow[dr, phantom, "\substack{no, m \\ \text{Lemma 3.4.27} \\ \text{⑤}}"] & s \arrow[l, "{ldel, L_R^*[\cdot]}"'] \arrow[d, dashed, "{no, m}"] \\
t_{WF} \arrow[r, dashed, "\substack{ldel, L_R^*[\cdot] \\ \text{Lemma 3.4.3} \\ \text{②}}"'] & t'_{WF} \arrow[r, dashed, "\substack{lcom, * \\ \text{Lemma 3.4.19} \\ \text{④}}"'] & s'_{WF} & s_{WF} \arrow[l, dashed, "\substack{ldel, L_R^*[\cdot] \\ \text{Lemma 3.4.3} \\ \text{⑥}}"]
\end{tikzcd}$$

Abbildung 3.14:

ausgehende no-Reduktion nach t, wobei wiederum die Länge der no-Reduktion unverändert bleibt. Lemma 3.4.3 sichert uns, daß sich t_{WF} wie t'_{WF} in WHNF befindet. □

Die Erhaltung der Länge der no-Reduktion, wie sie in obigem Lemma gezeigt wird, werden wir später benötigen, um induktiv argumentieren zu können. Wir werden nun die "Umkehrung" des obigen Lemmas, d.h. die Folgerung einer no-Reduktion zu WHNF ausgehend von s wenn eine von t ausgehende no-Reduktion gegeben ist, zeigen.

Lemma 3.4.29. *Gegeben zwei Ausdrücke t und s mit $t \xrightarrow{lcld} s$ sowie ein Ausdruck t_{WF} in WHNF mit $t \xrightarrow{no,m} t_{WF}$ und $m \geq 0$. Dann existiert ein Ausdruck s_{WF} in WHNF, so daß die folgenden Reduktionszusammenhänge erfüllt werden:*

$$\begin{tikzcd}
t \arrow[r, "lcld"] \arrow[d, "{no, m}"'] & s \arrow[d, dashed, "{no, m}"] \\
t_{WF} \arrow[r, dotted, no head] & s_{WF}
\end{tikzcd}$$

Beweis. Der Beweis erfolgt nach demselben Schema wie bei Lemma 3.4.28. Zunächst zerlegen wir die *lcld*-Reduktion in ihre Bestandteile. Danach folgen wir den Lemmata des Reduktionsdiagramms aus Abbildung 3.14 gemäß der Numerierung und zeigen auf diesem Wege die Existenz der drei Ausdrücke s'_{WF}, t'_{WF} und t_{WF}. Die Erhaltung der WHNF von s_{WF} bis zu t_{WF} sichern uns wiederum die Lemma 3.4.3 sowie 3.4.19. □

Aufsplittung der *ucp*-Reduktion; die *ucpw*-Reduktion

Um den im nächsten Abschnitt folgenden Beweis zu erleichtern, definieren wir nun eine abgeschwächte Form der *ucp*-Reduktion, die wir mit *ucpw* bezeichnen.

Definition 3.4.4. *Die ucpw-Reduktion wird wie folgt definiert:*

$$C[\texttt{let } x = t_x \texttt{ in } R[x]] \xrightarrow{ucpw} C[\texttt{let } x = t_x \texttt{ in } R[t_x]], \textit{ falls}$$

1) t_x *ein Let-Ausdruck oder eine Applikation ist und*
2) die Variable x *im Ausdruck* $D[x]$ *genau einmal vorkommt*

Offensichtlich ist eine *ucpw*-Reduktion eine *ucp*-Reduktion, bei der verlangt wird, daß der zu kopierende Ausdruck eine Applikation oder ein Let-Ausdruck ist. Den Grund der Einführung der *ucpw*-Reduktion verdeutlicht das nun folgende Lemma:

Lemma 3.4.30. *Eine ucp-Reduktion entspricht:*

- *einer ucpw-Reduktion, falls der zu kopierende Ausdruck ein Let-Ausdruck oder eine Applikation ist.*
- *einer lcv-Reduktion, falls der zu kopierende Ausdruck eine Variable ist.*
- *einer cp-Reduktion, falls der zu kopierende Ausdruck eine Abstraktion oder die Konstante* `choice` *ist.*

Beweis. Offensichtlicher Sachverhalt. □

Für die *lcv*- und *cp*-Reduktion haben wir die kontextuelle Äquivalenz bereits gezeigt. Es reicht somit die kontextuelle Äquivalenz für eine *ucpw*-Reduktion zu zeigen, um daraus die kontextuelle Äquivalenz der *ucp*-Reduktion folgern zu können.

Erhaltung der kontextuellen Äquivalenz durch die *ucpw*-Reduktion

Wir führen den Beweis unter Zuhilfenahme des Kontextlemmas, wie wir dies bereits bei anderen Reduktionen in diesem Abschnitt kennengelernt haben. Wir zeigen zunächst einen vollständigen Satz von Vertauschungs- und Gabeldiagrammen der $(ucpw, R[\cdot])$-Reduktion. Die Einschränkung auf einen R-Kontext verhindert dabei das auftreten der sonst sehr störenden möglichen Verdoppelung eines *ucpw*-Redex durch eine (no, cp)-Reduktion.

Lemma 3.4.31. *Die* $(ucpw, R[\cdot])$*-Reduktion verfügt über den folgenden vollständigen Satz von Gabeldiagrammen:*

i) $\xleftarrow{no,a} \circ \xrightarrow{ucpw,R[\cdot]} \;\rightsquigarrow\; \xrightarrow{ucpw,R[\cdot]} \circ \xleftarrow{no,a}$ *für* $a \in \{llet, lapp, lbeta, cp, nd\}$

ii) $\xleftarrow{no,c} \circ \xrightarrow{ucpw,R[\cdot]} \;\rightsquigarrow\; \xrightarrow{ucpw,R[\cdot]} \circ \xrightarrow{lcld} \circ \xleftarrow{no,c}$ *für* $c \in \{lbeta, nd, cp\}$

iii) $\xleftarrow{no,lapp,m} \circ \xrightarrow{ucpw,R[\cdot]} \quad \rightsquigarrow \quad \xrightarrow{ucpw,R[\cdot]} \circ \xrightarrow{lcld} \circ \xleftarrow{no,lapp,m}$ *für* $m > 0$

iv) $\xleftarrow{no,llet} \circ \xrightarrow{ucpw,R[\cdot]} \quad \rightsquigarrow \quad \xrightarrow{ucp,R[\cdot]} \circ \xrightarrow{lcld} \circ \xleftarrow{no,lapp,*}$

v) $\xleftarrow{no,llet} \circ \xrightarrow{ucpw,R[\cdot]} \quad \rightsquigarrow \quad \xrightarrow{ucp,R[\cdot]} \circ \xrightarrow{lcld} \circ \xleftarrow{no,llet} \circ \xleftarrow{no,lapp,*}$

Beweis. Der Beweis erfolgt vergleichbar den Beweisen der Gabeldiagramme bei den anderen Reduktionen. Folgendes Standarddiagramm kann vorkommen:

$$\begin{tikzcd}
t \arrow[r, "ucpw{,}R[\cdot]"] \arrow[d, "no{,}a"] & t_1 \arrow[d, "no{,}a"] \\
t' \arrow[r, "ucpw{,}R[\cdot]"] & t'_1
\end{tikzcd}$$

für $a \in \{lapp, llet, lbeta, cp, nd_{left}, nd_{right}\}$

Wir betrachten alle Sonderfälle, bei denen ein vom obigen Fall abweichendes Diagramm vorkommt. Die dabei vorkommenden Folgen von $(ldel, L^*_R[\cdot])$- und $lcom$-Reduktionen können dabei stets zu einer $lcld$-Reduktion zusammengezogen werden.

ii.1) $(no, lbeta)$ im L_L-Subkontext eines W-Kontext

$$\begin{tikzcd}
L^*_R[\texttt{let } x = A^n_L[(\lambda y.t)t_y] \texttt{ in } W[x]] \arrow[r, "ucpw{,}R[\cdot]"] \arrow[d, "no{,}lbeta"] & L^*_R[\texttt{let } x = \cdots] \texttt{ in } W[A^n_L[(\lambda y.t)t_y]] \arrow[d, "no{,}lbeta"] \\
L^*_R[\texttt{let } x = A^n_L[\texttt{let } y = t_y \texttt{ in } t] \texttt{ in } W[x]] \arrow[d, "ucpw{,}R[\cdot]"] & L^*_R[\texttt{let } x = \cdots] \texttt{ in } W[A^n_L[\texttt{let } y = t_y \texttt{ in } t]] \arrow[d, "ldel{,}L^*_R[\cdot]"] \\
L^*_R[\texttt{let } x = \cdots \texttt{ in } W[A^n_L[\texttt{let } y = t_y \texttt{ in } t]]] \arrow[r, "ldel{,}L^*_R[\cdot]"] & L^*_R[W[A^n_L[\texttt{let } y = t_y \texttt{ in } t]]]
\end{tikzcd}$$

ii.2) (no, nd_a) im L_L-Subkontext eines W-Kontext

$$\begin{tikzcd}
L^*_R[\texttt{let } x = A^n_L[\texttt{choice } s] \texttt{ in } W[x]] \arrow[r, "ucpw{,}R[\cdot]"] \arrow[d, "no{,}nd_a"] & L^*_R[\texttt{let } x = \cdots] \texttt{ in } W[A^n_L[\texttt{choice } s]] \arrow[d, "no{,}nd_a"] \\
L^*_R[\texttt{let } x = A^n_L[K_a\, s] \texttt{ in } W[x]] \arrow[d, "ucpw{,}R[\cdot]"] & L^*_R[\texttt{let } x = \cdots] \texttt{ in } W[A^n_L[K_a\, s]] \arrow[d, "ldel{,}L^*_R[\cdot]"] \\
L^*_R[\texttt{let } x = \cdots \texttt{ in } W[A^n_L[K_a\, s]]] \arrow[r, "ldel{,}L^*_R[\cdot]"] & L^*_R[W[A^n_L[K_a\, s]]]
\end{tikzcd}$$

wobei $a \in \{left, right\}$, $K_{right} \equiv \lambda x.\lambda y.y$ und $K_{left} \equiv \lambda x.\lambda y.x$.

ii.3) (no, cp) und das Kopierziel befindet sich im L_L-Subkontext eines W-Kontext

$$\begin{tikzcd}
L^*_R[\texttt{let } x = t_x \texttt{let } y = A^n_L[x] \texttt{ in } W[y]] \arrow[r, "ucpw{,}R[\cdot]"] \arrow[d, "no{,}cp"] & L^*_R[\texttt{let } x = t_x \texttt{let } y = A^n_L[x] \texttt{ in } W[A^n_L[x]]] \arrow[d, "no{,}cp"] \\
L^*_R[\texttt{let } x = t_x \texttt{let } y = A^n_L[t_x] \texttt{ in } W[y]] \arrow[d, "ucpw{,}R[\cdot]"] & L^*_R[\texttt{let } x = t_x \texttt{let } y = A^n_L[x] \texttt{ in } W[A^n_L[t_x]]] \arrow[d, "ldel{,}L^*_R[\cdot]"] \\
L^*_R[\texttt{let } x = t_x \texttt{let } y = A^n_L[t_x] \texttt{ in } W[A^n_L[t_x]]] \arrow[r, "ldel{,}L^*_R[\cdot]"] & L^*_R[\texttt{let } x = t_x \texttt{ in } W[A^n_L[t_x]]]
\end{tikzcd}$$

iii) $(no, lapp)$ im L_L-Subkontext eines W-Kontext

$$\begin{tikzcd}
L^*_R[\mathtt{let}\ x = A^n_L[(\mathtt{let}\ y = t_y\ \mathtt{in}\ s)t]\ \mathtt{in}\ W[x]] \arrow[r, "\mathit{ucpw},R[\cdot]"] \arrow[d, "\mathit{no,lapp}"] & L^*_R[\mathtt{let}\ x = \cdots\ \mathtt{in}\ W[A^n_L[(\mathtt{let}\ y = t_y\ \mathtt{in}\ s)t]]] \arrow[d, "\mathit{no,lapp}"] \\
L^*_R[\mathtt{let}\ x = A^n_L[\mathtt{let}\ y = t_y\ \mathtt{in}\ (st)]\ \mathtt{in}\ W[x]] \arrow[d, "\mathit{no,lapp,n}"] & L^*_R[\mathtt{let}\ x = \cdots\ \mathtt{in}\ W[A^n_L[\mathtt{let}\ y = t_y\ \mathtt{in}\ (st)]]] \arrow[d, "\mathit{no,lapp,n}"] \\
L^*_R[\mathtt{let}\ x = \mathtt{let}\ y = t_y\ \mathtt{in}\ A^n_L[st]\ \mathtt{in}\ W[x]] \arrow[d, "\mathit{ucpw},R[\cdot]"] & L^*_R[\mathtt{let}\ x = \cdots\ \mathtt{in}\ W[\mathtt{let}\ y = t_y\ \mathtt{in}\ A^n_L[st]]] \arrow[d, "\mathit{ldel},L^*_R[\cdot]"] \\
L^*_R[\mathtt{let}\ x = \cdots\ \mathtt{in}\ W[\mathtt{let}\ y = t_y\ \mathtt{in}\ A^n_L[st]]] \arrow[r, "\mathit{ldel},L^*_R[\cdot]"] & L^*_R[W[\mathtt{let}\ y = t_y\ \mathtt{in}\ A^n_L[st]]]
\end{tikzcd}$$

Die nun folgenden zwei Fälle nehmen eine Sonderstellung ein, da sich bei ihnen die $(ucpw, R[\cdot])$-Reduktion zu einer $(ucp, R[\cdot])$-Reduktion wandeln kann.

iv) $(no, llet)$ und $ucpw$-Kopie in einen W-Kontext hinein und der W-Kontext besitzt keinen L_L-Subkontext

$$\begin{tikzcd}
L^*_R[\mathtt{let}\ x = (\mathtt{let}\ y = t_y\ \mathtt{in}\ t_x)\ \mathtt{in}\ L^m_R[A^n_L[x]]] \arrow[r, "\mathit{ucpw},R[\cdot]"] \arrow[d, "\mathit{no,llet}"] & L^*_R[\mathtt{let}\ x = \cdots\ \mathtt{in}\ L^m_R[A^n_L[\mathtt{let}\ y = t_y\ \mathtt{in}\ t_x]]] \arrow[d, "\mathit{no,lapp,n}"] \\
L^*_R[\mathtt{let}\ y = t_y\ \mathtt{in}\ \mathtt{let}\ x = t_x\ \mathtt{in}\ L^m_R[A^n_L[x]]] \arrow[d, "\mathit{ucp},R[\cdot]"] & L^*_R[\mathtt{let}\ x = \cdots\ \mathtt{in}\ L^m_R[\mathtt{let}\ y = t_y\ \mathtt{in}\ A^n_L[t_x]]] \arrow[dd, "\mathit{ldel},L^*_R[\cdot]"] \\
L^*_R[\mathtt{let}\ y = t_y\ \mathtt{in}\ \mathtt{let}\ x = t_x\ \mathtt{in}\ L^m_R[A^n_L[t_x]]] \arrow[d, "\mathit{ldel},L^*_R[\cdot]"] & \\
L^*_R[\mathtt{let}\ y = t_y\ \mathtt{in}\ L^m_R[A^n_L[t_x]]] \arrow[r, "\mathit{lcom,m}"] & L^*_R[L^m_R[\mathtt{let}\ y = t_y\ \mathtt{in}\ A^n_L[t_x]]]
\end{tikzcd}$$

v) $(no, llet)$ und $ucpw$-Kopie in einen W-Kontext hinein und der W-Kontext besitzt einen L_L-Subkontext

$$\begin{tikzcd}
L^*_R[\mathtt{let}\ x = (\mathtt{let}\ y = t_y\ \mathtt{in}\ t_x)\ \mathtt{in}\ L^m_R[L_L[A^n_L[x]]]] \arrow[r, "\mathit{ucpw},R[\cdot]"] \arrow[d, "\mathit{no,llet}"] & L^*_R[\mathtt{let}\ x = \cdots\ \mathtt{in}\ L^m_R[L_L[A^n_L[\mathtt{let}\ y = t_y\ \mathtt{in}\ t_x]]]] \arrow[d, "\mathit{no,lapp,n}"] \\
L^*_R[\mathtt{let}\ y = t_y\ \mathtt{in}\ \mathtt{let}\ x = t_x\ \mathtt{in}\ L^m_R[L_L[A^n_L[x]]]] \arrow[d, "\mathit{ucp},R[\cdot]"] & L^*_R[\mathtt{let}\ x = \cdots\ \mathtt{in}\ L^m_R[L_L[\mathtt{let}\ y = t_y\ \mathtt{in}\ A^n_L[t_x]]]] \arrow[d, "\mathit{no,llet}"] \\
L^*_R[\mathtt{let}\ y = t_y\ \mathtt{in}\ \mathtt{let}\ x = t_x\ \mathtt{in}\ L^m_R[L_L[A^n_L[t_x]]]] \arrow[d, "\mathit{ldel},L^*_R[\cdot]"] & L^*_R[\mathtt{let}\ x = \cdots\ \mathtt{in}\ L^m_R[\mathtt{let}\ y = t_y\ \mathtt{in}\ L_L[A^n_L[t_x]]]] \arrow[d, "\mathit{ldel},L^*_R[\cdot]"] \\
L^*_R[\mathtt{let}\ y = t_y\ \mathtt{in}\ L^m_R[L_L[A^n_L[t_x]]]] \arrow[r, "\mathit{lcom,m}"] & L^*_R[L^m_R[\mathtt{let}\ y = t_y\ \mathtt{in}\ L_L[A^n_L[t_x]]]]
\end{tikzcd}$$

□

Neben einem vollständigen Satz von Gabeldiagrammen werden wir auch einen vollständigen Satz von Vertauschungsdiagrammen für die $(ucpw, R[\cdot])$-Reduktion benötigen. Letzterer wird nun vorgestellt.

Lemma 3.4.32. *Die $(ucpw, R[\cdot])$-Reduktion verfügt über den folgenden vollständigen Satz von Vertauschungsdiagrammen:*

i) $(ucpw, R[\cdot]) \circ (no, a) \rightsquigarrow (no, a) \circ (ucpw, R[\cdot])$
für $a \in \{llet, lapp, lbeta, cp, nd\}$

ii) $(ucpw, R[\cdot]) \circ (no, b) \rightsquigarrow (no, b) \circ (ucpw, R[\cdot]) \circ lcld$
für $b \in \{lbeta, nd, cp\}$

iii) $(ucpw, R[\cdot]) \circ (no, lapp, m) \rightsquigarrow (no, lapp, m) \circ (ucpw, R[\cdot]) \circ lcld$
für $m > 0$

iv.1) $(ucpw, R[\cdot]) \circ (no, lapp, +) \rightsquigarrow (no, llet) \circ (ucp, R[\cdot]) \circ lcld$

iv.2) $(ucpw, R[\cdot]) \circ (no, c) \rightsquigarrow (no, llet, +) \circ (no, c) \circ (ucp, R[\cdot]) \circ lcld$
für $c \in \{llet, lapp, lbeta, cp, nd\}$

v) $(ucpw, R[\cdot]) \circ (no, lapp, *) \circ (no, llet) \rightsquigarrow (no, llet) \circ (ucp, R[\cdot]) \circ lcld$

Beweis. Alle Gabeldiagramme aus Lemma 3.4.31 bis auf das mit *iv)* gekennzeichnete können wie in Abschnitt 2.4.2 beschrieben direkt in ein Vertauschungsdiagramm überführt werden. Wir betrachten diesen Sonderfall:
Verfügt die $(no, lapp, *)$-Reduktion aus Gabeldiagramm *iv)* mindestens über eine no-Reduktion entsteht das mit *iv.1)* gekennzeichnete Vertauschungsdiagramm. Entspricht die $(no, lapp, *)$-Reduktion der Identität, wird das Vertauschungsdiagramm *iv.2)* erhalten. Zur Bildung des Vertauschungsdiagramms *iv.2)* wird die Kenntnis aus Lemma 3.4.31 über das Aussehen der Kontexte benutzt, in denen die involvierten *lcom* und *ldel*-Reduktionen vorkommen. □

Nachdem wir einen vollständigen Satz von Gabel- und Vertauschungsdiagrammen für die $(ucpw, R[\cdot])$-Reduktion entwickelt haben, werden wir nun die Erhaltung einer WHNF durch die $(ucpw, R[\cdot])$-Reduktion untersuchen.

Lemma 3.4.33. *Gegeben zwei geschlossene Ausdrücke* t *und* s *mit* $t \xrightarrow{ucpw} s$. *Dann gilt:*

1. t *in WHNF* $\Rightarrow$ s *in WHNF*
2. s *in WHNF* $\Rightarrow$ *es existiert ein Ausdruck* t' *mit* $t \xrightarrow{no,*} t'$ *und* t' *in WHNF*

Beweis.
1.
Gemäß Satz 2.3.1 hat jeder Ausdruck in WHNF die Form $L_R^*[r]$, mit r eine

Abstraktion oder die Konstante `choice` . Es ist offensichtlich, daß eine *ucpw*-Reduktion einen Ausdruck aus der Form $L_R^*[r]$ nicht herausführen kann. Wegen der in Satz 2.3.1 formulierten gdw.-Beziehung ist das Resultat der *ucpw*-Reduktion somit wieder in WHNF.

2.
Die Annahme, daß sich s in WHNF befindet und t einen no-Redex besitzt, führt in Verbindung mit allen Gabeldiagrammen außer dem mit *iv)* gekennzeichneten unter der Nebenbedingung, daß die involvierte $(no, lapp, *)$-Reduktion der Identität entspricht, sofort zu einem Widerspruch.
Bleibt somit zu zeigen, daß im Falle von Gabeldiagramm *iv)* und dem Sonderfall, daß die $(no, lapp, *)$-Reduktion der Identität entspricht, stets ein Ausdruck t' mit $t \xrightarrow{no,*} t'$ und t' in WHNF existiert. Wir weisen den Platzhaltern aus Gabeldiagramm *iv)* wie folgt Ausdrücke zu:

$$t'' \xleftarrow{no,llet} t \xrightarrow{ucpw,R[\cdot]} s \quad \rightsquigarrow \quad t'' \xrightarrow{ucp,R[\cdot]} s' \xrightarrow{lcld} s$$

Aus der in Lemma **??** gezeigten Erhaltung einer WHNF durch eine *lcld*-Reduktion folgt, daß sich s' in WHNF befinden muß, da sich s in WHNF befindet. Die $(ucp, R[\cdot])$-Reduktion von t'' zu s' kann keiner (i, cp)- oder *lcv*-Reduktion entsprechen, da dann gemäß Korollar 2.3.2 bzw. Lemma 3.4.8 auch t'' in WHNF sein müßte, was jedoch ein Widerspruch zu der in Lemma 2.3.6 getroffenen Aussage wäre, daß ein $(no, llet)$-Redex einen Ausdruck nicht in WHNF überführen kann. Somit kann die $(ucp, R[\cdot])$-Reduktion von t'' zu s' nur einer (no, cp)- oder *ucpw*-Reduktion entsprechen. Diese beiden Möglichkeiten werden nun betrachtet:

1. Entspricht die $(ucp, R[\cdot])$-Reduktion einer (no, cp)-Reduktion, so führt eine no-Reduktion von t zu s'. s' wiederum befindet sich in WHNF. Indem wir $t' \equiv s'$ wählen, haben wir die Hypothese gezeigt.

2. Entspricht die $(ucp, R[\cdot])$-Reduktion einer $(ucpw, R[\cdot])$-Reduktion, dann muß bei dieser Reduktion genau wieder der gerade betrachtete Sonderfall vorliegen, da in allen anderen Fällen das Aussehen der Gabeldiagramme einen Widerspruch impliziert. Wir wenden die Argumentation rekursiv auf die $(ucp, R[\cdot])$-Reduktion von t'' nach s' an. Dieser Prozess muß nach endlich vielen Rekursionen bei der in 1. betrachteten Möglichkeit stoppen, da eine alleinig aus *llet*-Reduktionen bestehende Reduktion gemäß Korollar 2.2.1 stets endlich beschränkt ist und eine *llet*-Reduktion einen Ausdruck nicht in WHNF überführen kann. □

Eine aus obigem Lemma folgerbare Aussage ist, daß eine *ucpw*-Reduktion einen Ausdruck in WHNF überführen kann. Diese starke Eigenschaft der *ucpw*-Reduktion soll an einem Beispiel verdeutlicht werden:

Beispiel 3.4.1. Die *ucpw*-Reduktion

$$\texttt{let}\ x = t_x\ \texttt{in}\ x \xrightarrow{ucpw} \texttt{let}\ x = t_x\ \texttt{in}\ t_x$$

$$\text{mit } t_x \equiv \texttt{let}\ y = t_y\ \texttt{in}\ \lambda z.z$$

zeigt ein Beispiel, wie eine *ucpw*-Reduktion einen Ausdruck in WHNF überführt.

Wir werden nun die Übertragbarkeit einer $(no,*)$-Reduktion durch eine $(ucpw, R[\cdot])$-Reduktion untersuchen, bevor wir zum Beweis der kontextuellen Äquivalenz übergehen.

Lemma 3.4.34. *Gegeben zwei Ausdrücke* t, s *mit* $t \xrightarrow{ucpw,R[\cdot]} s$ *sowie ein Ausdruck* t_{WF} *in WHNF mit* $t \xrightarrow{no,*} t_{WF}$. *Dann existiert ein Ausdruck* s_{WF} *in WHNF , so daß die folgenden Reduktionszusammenhänge erfüllt werden:*

$$\begin{tikzcd}
t \arrow[r, "{ucpw, R[\cdot]}"] \arrow[d, "{no,*}"'] & s \arrow[d, dashed, "{no,*}"] \\
t_{WF} \arrow[r, dotted, no head] & s_{WF}
\end{tikzcd}$$

Beweis. Wir führen eine Induktion über die Länge der no-Reduktion von t zu t_{WF} und benutzen als Induktionshypothese die Hypothese des Lemmas.

Induktionsbeginn:
Die no-Reduktion von t zu t_{WF} hat die Länge 0, d.h t befindet sich in WHNF. Aus Lemma 3.4.33 folgt daraus, daß sich auch s in WHNF befindet. Wir wählen $s_{WF} \equiv s$ und haben die Hypothese für diesen Fall gezeigt.

Induktionsschritt:
Wir zerlegen die no-Reduktion von t zu t_{WF} in $t \xrightarrow{no,+} t' \xrightarrow{no,*} t_{WF}$ derart, daß sich eines der Gabeldiagramme der $(ucpw, R[\cdot])$-Reduktion aus Lemma 3.4.31 schließen läßt.
Die von t' ausgehende no-Reduktion ist um mindestens eine einzelne no-Reduktion verkürzt, so daß auf t' die Induktionshypothese angewendet werden kann. Die in Abbildung 3.17 enthaltenen Reduktionsdiagramme geben für alle Gabeldiagramme einen Beweis der Hypothese in grafischer Form. Die Lemmata sind dabei entsprechend der Nummerierung anzuwenden. t'_{WF} ist dabei stets ein Ausdruck in WHNF. Diagramme 2 ist bei den mit *iv*) und *v*) gekennzeichneten Gabeldiagrammen aus Lemma 3.4.31 zu wählen, falls die $(ucp, R[\cdot])$-Reduktion einer *lcv*- oder *cp*-Reduktion entspricht. In allen anderen Fällen ist der Beweis aus Diagramm 1 ablesbar. Bei Diagramm 2 ist kein weiterer Rückgriff auf die Induktionshypothese erforderlich, da wir die in Satz 3.4.2 und 3.3.1 gezeigte kontextuelle Äquivalenz einer *lcv*- bzw. *cp*-Reduktion benutzen können, um die Existenz des Ausdrucks t'_{WF} in WHNF zu erhalten. □

Diagramm 1

$$\begin{tikzcd}
t \arrow[rr, "{ucpw, R[\cdot]}"] \arrow[d, "{no,+}"'] \arrow[dr, phantom, "\text{Lemma 3.4.31 ①}"] & & s \arrow[d, dashed, "{no,*}"] \\
t' \arrow[r, dashed, shift left] \arrow[r, shift right, "{ucpw, R[\cdot]}"'] \arrow[d, "{no,*}"'] \arrow[dr, phantom, "\text{Induktionsvoraussetzung ②}"] & \circ \arrow[r, dashed, shift left] \arrow[r, shift right, "{lcld, 0/1}"'] \arrow[d, dashed, shift right] \arrow[d, shift left, "{no,*}"] \arrow[dr, phantom, "\text{Lemma 3.4.29 ③}"] & \circ \arrow[d, dashed, "{no,*}"] \\
t_{WF} \arrow[r, dotted, no head] & t'_{WF} \arrow[r, dotted, no head] & s_{WF}
\end{tikzcd}$$

Diagramm 2

$$\begin{tikzcd}
t \arrow[rr, "{ucpw, R[\cdot]}"] \arrow[d, "{no,+}"'] \arrow[dr, phantom, "\text{Lemma 3.4.31 ①}"] & & s \arrow[d, dashed, "{no,*}"] \\
t' \arrow[r, dashed, shift left] \arrow[r, shift right, "{\{lcv, cp\}, R[\cdot]}"'] \arrow[d, "{no,*}"'] \arrow[dr, phantom, "\sim_c \text{ von } lcv \text{ und } cp \text{ ②}"] & \circ \arrow[r, dashed, shift left] \arrow[r, shift right, "lcld"'] \arrow[d, dashed, shift right] \arrow[d, shift left, "{no,*}"] \arrow[dr, phantom, "\text{Lemma 3.4.29 ③}"] & \circ \arrow[d, dashed, "{no,*}"] \\
t_{WF} \arrow[r, dotted, no head] & t'_{WF} \arrow[r, dotted, no head] & s_{WF}
\end{tikzcd}$$

Abbildung 3.15:

Was wir für eine von t ausgehende no-Reduktion zu WHNF gezeigt haben, werden wir nun genauso für eine von s ausgehende no-Reduktion zu WHNF zeigen.

Lemma 3.4.35. *Gegeben zwei Ausdrücke t, s mit $t \xrightarrow{ucpw,R[\cdot]} s$ sowie ein Ausdruck s_{WF} in WHNF mit $s \xrightarrow{no,*} s_{WF}$. Dann existiert ein Ausdruck t_{WF} in WHNF , so daß die folgenden Reduktionszusammenhänge erfüllt werden:*

$$\begin{tikzcd}
t \arrow[r, "{ucpw, R[\cdot]}"] \arrow[d, dashed, "{no,*}"'] & s \arrow[d, "{no,*}"] \\
t_{WF} \arrow[r, dotted, no head] & s_{WF}
\end{tikzcd}$$

Beweis. Wir führen eine Induktion über die Länge der no-Reduktion von s zu s_{WF} und benutzen als Induktionshypothese die Hypothese des Lemmas.

Induktionsbeginn:
Die no-Reduktion von s zu s_{WF} hat die Länge 0, d.h s befindet sich in WHNF. Aus Lemma 3.4.33 folgt daraus, daß ausgehend von t mittels einer $(no, *)$-Reduktion

$$\begin{tikzcd}
 & t \arrow[rr, "ucpw{,}\, R[\cdot]"] \arrow[d, dashed, "no{,}\, +"'] \arrow[drr, phantom, "\text{Lemma 3.4.32 } ①"] & & s \arrow[d, "no{,}\, +"] \\
\text{Diagramm 1} & \circ \arrow[r, dashed, shift left] \arrow[r, shift right, "ucpw{,}\, R[\cdot]"'] \arrow[d, dashed, "no{,}\, *"'] \arrow[dr, phantom, "\text{Induktionsvoraussetzung } ②"] & t' \arrow[r, dashed, shift left] \arrow[r, shift right, "lcld{,}\, 0/1"'] \arrow[d, shift right] \arrow[d, dashed, shift left, "no{,}\, m"] \arrow[dr, phantom, "\text{Lemma 3.4.28 } ③"] & s' \arrow[d, "no{,}\, m"] \\
 & t_{WF} \arrow[r, dotted, no head] & t'_{WF} \arrow[r, dotted, no head] & s_{WF}
\end{tikzcd}$$

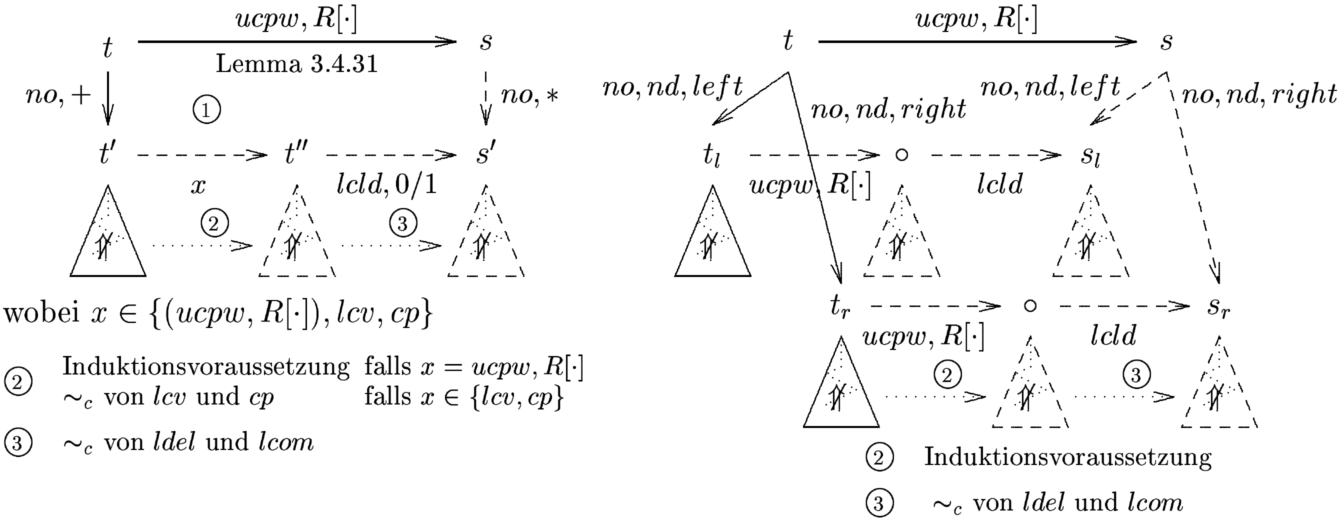
wobei $m \geq 0$

$$\begin{tikzcd}
 & t \arrow[rr, "ucpw{,}\, R[\cdot]"] \arrow[d, dashed, "no{,}\, +"'] \arrow[drr, phantom, "\text{Lemma 3.4.32 } ①"] & & s \arrow[d, "no{,}\, +"] \\
\text{Diagramm 2} & \circ \arrow[r, dashed, shift left] \arrow[r, shift right, "\{lcv{,}\, cp\}{,}\, R[\cdot]"'] \arrow[d, dashed, "no{,}\, *"'] \arrow[dr, phantom, "\sim_c \text{ von } lcv \text{ und } cp \; ②"] & t' \arrow[r, dashed, shift left] \arrow[r, shift right, "lcld"'] \arrow[d, shift right] \arrow[d, dashed, shift left, "no{,}\, m"] \arrow[dr, phantom, "\text{Lemma 3.4.28 } ③"] & s' \arrow[d, "no{,}\, m"] \\
 & t_{WF} \arrow[r, dotted, no head] & t'_{WF} \arrow[r, dotted, no head] & s_{WF}
\end{tikzcd}$$

Abbildung 3.16:

ein Ausdruck t_{WF} in WHNF erreichbar ist, womit wir die Hypothese für diesen Fall gezeigt haben.

Induktionsschritt:
Wir zerlegen die no-Reduktion von s zu s_{WF} in $s \xrightarrow{no,+} s' \xrightarrow{no,*} s_{WF}$ derart, daß sich eines der Vertauschungsdiagramme der $(ucpw, R[\cdot])$-Reduktion aus Lemma 3.4.32 schließen läßt.
Die in Abbildung 3.16 enthaltenen Reduktionsdiagramme geben, für alle Vertauschungsdiagramme einen Beweis der Hypothese in grafischer Form. Die Lemmata sind dabei entsprechend der Nummerierung anzuwenden. t'_{WF} ist stets ein Ausdruck in WHNF. Die von t' ausgehende no-Reduktion ist um mindestens eine einzelne no-Reduktion gegenüber der von s ausgehenden verkürzt, da Lemma 3.4.28 die Länge der no-Reduktion von s' zu WHNF auf t' überträgt. Somit können wir auf t' die Induktionshypothese anwenden.
Diagramme 2 ist bei den mit *iv)* und *v)* gekennzeichneten Gabeldiagrammen aus Lemma 3.4.31 zu wählen, sofern die $(ucp, R[\cdot])$-Reduktion einer *lcv*- oder *cp*-Reduktion entspricht. In allen anderen Fällen ist der Beweis aus Diagramm 1 ablesbar. Bei Diagramm 2 ist kein weiterer Rückgriff auf die Induktionshypothese erforderlich, da wir die in Satz 3.4.2 und 3.3.1 gezeigte kontextuelle Äquivalenz einer *lcv*- bzw. *cp*-Reduktion benutzen können, um die Existenz des Ausdrucks



wobei $x \in \{(ucpw, R[\cdot]), lcv, cp\}$

② Induktionsvoraussetzung falls $x = ucpw, R[\cdot]$
$\sim_c$ von *lcv* und *cp* falls $x \in \{lcv, cp\}$

③ $\sim_c$ von *ldel* und *lcom*

② Induktionsvoraussetzung

③ $\sim_c$ von *ldel* und *lcom*

Abbildung 3.17:

t_{WF} in WHNF zu erhalten. □

Wir haben in den letzten beiden Lemmata zwei Existenz-quantifizierte Aussagen getroffen, die geeignet sind, die in der kontextuellen Äquivalenz der $(ucpw, R[\cdot])$-Reduktion enthaltene Übertragbarkeit der Konvergenz zu zeigen. Im Bezug auf die auch zu zeigende Übertragbarkeit der Divergenz sind die beiden Lemmata jedoch wertlos, da sie erfordern, daß die gegebene Reduktion zu WHNF führt. Wir werden nun wie bei der Betrachtung der *lcv*- und *lcom*-Reduktion für zwei Ausdrücke t und s mit $t \xrightarrow{ucpw, R[\cdot]} s$ zeigen, daß die Endlichkeit aller von t ausgehenden no-Reduktionen die Endlichkeit aller von s ausgehenden no-Reduktionen impliziert und umgekehrt. Durch eine einfache Widerspruchsüberlegung können wir daraus die Übertragbarkeit der Divergenz folgern.

Lemma 3.4.36. *Gegeben zwei geschlossene Ausdrücke* t, s *mit* $t \xrightarrow{ucpw, R[\cdot]} s$. *Dann gilt* $t\not\Uparrow \Rightarrow s\not\Uparrow$.

Beweis. Wir führen eine Induktion über die Länge der längsten von t ausgehenden no-Reduktion und benutzen als Induktionshypothese die Hypothese des Lemmas.

Induktionsbeginn:
Die längste von t ausgehende no-Reduktion hat die Länge 0, d.h t befindet sich in WHNF. Aus Lemma 3.4.33 folgt daraus, daß sich auch s in WHNF befindet, womit wir die Hypothese für diesen Fall gezeigt haben.

Induktionsschritt:
Wir differenzieren danach, ob der no-Redex von t vom Typ *nd* ist oder nicht.

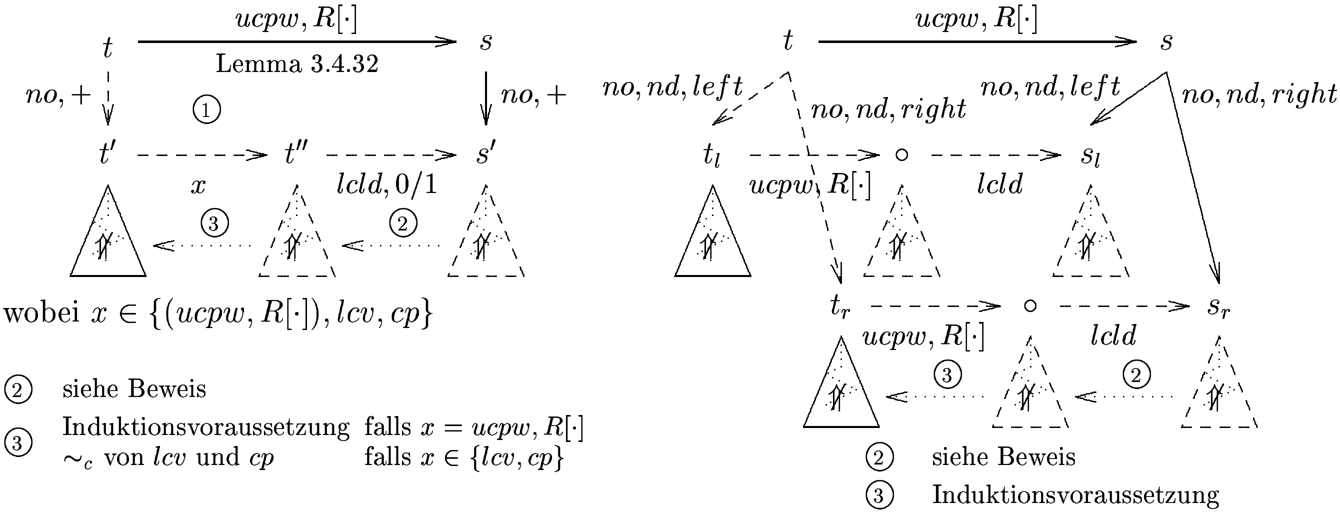
1. Fall: Der no-Redex von t ist vom Typ *llet*, *lapp*, *cp* oder *lbeta*.
In diesem Fall kann der Beweis der Hypothese aus dem linken Diagramm aus Abbildung 3.17 abgelesen werden. Wir spalten zunächst von der von t ausgehende no-Reduktion einen Teil $t \xrightarrow{no,+} t'$ ab, derart daß sich eines der Gabeldiagramme der $(ucpw, R[\cdot])$-Reduktion aus Lemma 3.4.31 schließen läßt. Die längste von t' ausgehende no-Reduktion ist mindestens eine no-Reduktion kürzer als die Längste von t ausgehende. In Abhängigkeit davon, ob wir die Sonderfälle von Diagramm *iv*) oder *v*) aus Lemma 3.4.31 vorliegen haben, bei denen in Folge einer $(ucp, R[\cdot])$-Reduktion eine *lcv*- oder *cp*-Reduktion auftritt, benutzen wir die Induktionsvoraussetzung oder die in Satz 3.4.2 bzw. Satz 3.3.2 bewiesene Erhaltung der kontextuellen Äquivalenz durch eine *lcv*- bzw. *cp*-Reduktion, um die Endlichkeit aller von t'' ausgehenden no-Reduktionen folgern zu können. Danach machen wir davon Gebrauch, daß eine *lcld*-Reduktion in einzelne *ldel*- und *lcom*-Reduktionen zerlegbar ist, die gemäß Satz 3.4.1 bzw. Satz 3.4.2 wiederum die kontextuelle Äquivalenz erhalten. Dieser Schritt entfällt bei Diagramm *i*) aus Lemma 3.4.31, da dort keine *lcld*-Reduktion vorkommt. Da die no-Reduktion von s zu s' über keine *nd*-Reduktionen verfügt und damit eindeutig ist und alle von s' ausgehenden no-Reduktionen endlich sind, müssen dies auch alle von s ausgehenden no-Reduktionen sein.

2. Fall: Der no-Redex von t ist vom Typ *nd*.
Der Beweis der Hypothese kann bei diesem Fall aus dem rechten Diagramm aus Abbildung 3.17 abgelesen werden. Die no-Reduktion ausgehend von t ist bei einem (no, nd)-Redex nicht eindeutig. Die längste von t_l wie t_r ausgehende no-Reduktion ist um mindestens eine no-Reduktion kürzer, als die Längste von t ausgehende. Die oben benutzte Argumentation kann deshalb unter Zuhilfenahme der Induktionsvoraussetzung simultan auf beide “Zweige” angewendet werden und so gezeigt werden, daß alle von s_l wie s_r ausgehenden no-Reduktionen endlich sind. Da ausgehend von s mittels einer einzelnen no-Reduktion keine anderen Ausdrücke außer s_l und s_r erreichbar sind und alle von s_l wie s_r ausgehenden no-Reduktionen endlich sind, müssen auch alle von s ausgehenden no-Reduktionen endlich sein. ☐

Nun gehen wir umgekehrt davon aus, daß ausgehend von s alle no-Reduktionen endlich sind und wollen diese Aussage auf alle von t ausgehenden no-Reduktionen übertragen.

Lemma 3.4.37. *Gegeben zwei geschlossene Ausdrücke* t, s *mit* $t \xrightarrow{ucpw,R[\cdot]} s$. *Dann gilt* $s\not\Uparrow \Rightarrow t\not\Uparrow$.

Beweis. Der Beweis ist dem von Lemma 3.4.36 sehr verwandt. Wir führen eine Induktion über die Länge der längsten von s ausgehenden no-Reduktion und benutzen als Induktionshypothese die Hypothese des Lemmas.



wobei $x \in \{(ucpw, R[\cdot]), lcv, cp\}$

② siehe Beweis

③ Induktionsvoraussetzung falls $x = ucpw, R[\cdot]$
$\sim_c$ von lcv und cp falls $x \in \{lcv, cp\}$

② siehe Beweis

③ Induktionsvoraussetzung

Abbildung 3.18:

Induktionsbeginn:
Die längste von s ausgehende no-Reduktion hat die Länge 0, d.h. s befindet sich in WHNF. Durch Anwendung von Lemma 3.4.33 erhalten wir die Existenz eines Ausdrucks t' in WHNF, für den $t \xrightarrow{no,\{llet,cp\},*} t'$ gilt. Da die no-Reduktion von t zu t' über keine *nd*-Reduktionen verfügt sind alle von t ausgehenden no-Reduktionen endlich und wir haben die Hypothese für diesen Fall gezeigt.

Induktionsschritt:
Wir differenzieren danach, ob der no-Redex von s vom Typ *nd* ist oder nicht.

1. Fall: Der no-Redex von s ist vom Typ *llet*, *lapp*, *cp* oder *lbeta*.
Wir bilden eine Zerlegung der von s ausgehenden no-Reduktion, derart daß sich eines der Vertauschungsdiagramme aus Lemma 3.4.32 schließen läßt. Das in Abbildung 3.18 enthaltene Reduktionsdiagramm zeigt die entstehenden Reduktionszusammenhänge. Zum Beweis ist der Nummerierung zu folgen.
Alle von s' ausgehenden no-Reduktionen müssen endlich sein, da per Voraussetzung alle von s ausgehenden no-Reduktionen endlich sind. Ferner ist die Längste von s' ausgehende no-Reduktion kürzer als die Längste von s ausgehende. Der mit ② gekennzeichnete Übergang benötigt besondere Aufmerksamkeit, da wir bei ③ die Induktionsvoraussetzung anwenden wollen. Diagramm 1) aus Lemma 3.4.32 ist unproblematisch, da die *lcld*-Reduktion dort nicht vorkommt, in allen anderen Fällen müssen wir jedoch zeigen, daß alle von t'' ausgehenden no-Reduktionen endlich sind und die längste von t'' ausgehende no-Reduktion nicht länger als die Längste von s' ausgehende ist. Dies zeigt folgende einfache Überlegung:

Jede *lcld*-Reduktion ist in einzelne *ldel*- und *lcom*-Reduktionen zerlegbar, für die wir die Erhaltung der kontextuellen Äquivalenz in Satz 3.4.1 und Satz 3.4.2 gezeigt haben. Daraus folgt, daß alle von t'' ausgehenden no-Reduktionen endlich

sein müssen, da alle von s' ausgehenden no-Reduktion endlich sind. Folgende einfache Widerspruchsüberlegung zeigt nun, daß die Längste von t'' ausgehende no-Reduktion nicht länger als die Längste von s' ausgehende ist:
Angenommen ausgehend von t'' existiert eine no-Reduktion r die länger als die längste von s' ausgehende no-Reduktion ist. Dann muß gemäß Lemma 3.4.29 ausgehend von s' eine no-Reduktion mit derselben Länge wie r existieren, was direkt zu einem Widerspruch führt.

Bei ③ können wir somit entweder die Induktionshypothese anwenden, oder wir Verwenden die in Satz 3.4.2 bzw. Satz 3.3.2 bewiesene Erhaltung der kontextuellen Äquivalenz durch eine *lcv*- oder *cp*-Reduktion, um die Endlichkeit aller von t'' ausgehenden no-Reduktionen zu folgern. Da die no-Reduktion von t zu t' über keine *nd*-Reduktionen verfügt, müssen somit auch alle von t ausgehenden no-Reduktionen endlich sein.

2. Fall: Der no-Redex von s ist vom Typ *nd*.
Die Verfahrensweise ist dieselbe wie in 3.4.36. Wir wenden die im ersten Fall benutzte Argumentation auf beide “Zweige” der *nd*-Reduktion an und erhalten die Hypothese durch die Vereinigung der zwei so bewiesenen partiellen Aussagen.

□

Die vier zuletzt bewiesenen ziehen wir nun zu einem einzelnen Lemma zusammen.

Satz 3.4.4. *Gegeben zwei geschlossene Ausdrücke t und s mit $t \xrightarrow{ucpw,R[\cdot]} s$. Dann gilt $(t\Downarrow \Leftrightarrow s\Downarrow)$ und $(t\Uparrow \Leftrightarrow s\Uparrow)$.*

Beweis. Die Erhaltung der Konvergenz zeigen die Lemmata 3.4.34 und 3.4.35. Die Erhaltung der Divergenz wird mittels einer einfachen Widerspruchsüberlegung aus den Lemmata 3.4.36 und 3.4.37 erhalten. □

Wir haben nun alle Voraussetzung geschaffen, um die Erhaltung der kontextuellen Äquivalenz durch eine *ucp*-Reduktion zeigen zu können.

Satz 3.4.5. *Gegeben zwei Ausdrücke t_1 und t_2.*

1. *Wenn $t_1 \xrightarrow{ucp} t_2$, dann $t_1 \sim_c t_2$.*
2. *Wenn $t_1 \xrightarrow{ucp,*} t_2$, dann $t_1\Downarrow \Leftrightarrow t_2\Downarrow$ und $t_1\Uparrow \Leftrightarrow t_2\Uparrow$.*

Beweis. Eine *ucp*-Reduktion entspricht gemäß Lemma 3.4.30 einer *ucpw*-, *lcv*- oder *cp*-Reduktion. Für die *lcv*- und *cp*-Reduktion wurde die Erhaltung der kontextuellen Äquivalenz bereits in Satz 3.4.2 bzw. Satz 3.3.2 gezeigt.

Bleibt die Erhaltung der kontextuellen Äquivalenz durch eine *ucpw*-Reduktion zu zeigen. Die *ucpw*-Reduktion ist kompatibel und der Redex-Begriff anwendbar. Lemma 3.4.4 zeigt deshalb in Kombination mit Lemma 3.2.3 die Erhaltung der kontextuellen Äquivalenz durch eine *ucpw*-Reduktion. □

3.5 Lambda-Lifting beim Λ_{let}-Kalkül

Die *ldel*- und *lcv*-Reduktion werden nun dazu verwendet, die Zulässigkeit einer wichtigen Programmtransformation mit dem Namen "Lambda-Lifting" beim Λ_{let}-Kalkül zu zeigen. Lambda-Lifting wird zur Lösung eines Effizienzproblems benutzt, auf das beim maschinellen Reduzieren getroffen wird, und dessen Ursache im Vorkommen von freien Variablen an ungünstigen Positionen begründet ist. Eine detailierte Darstellung der Problematik sowie der verschiedenen historisch entwickelten Lösungen enthält [PJ87]. Eine zentrale Rolle beim Lambda-Lifting übernimmt der Begriff des Superkombinators, der im Anschluß an diese Einleitung definiert wird. Wir werden hier nur eine vereinfachte Form des Lambda-Lifting betrachten und zeigen, daß jeder geschlossene Λ_{let}-Ausdruck in einen kontextuell äquivalenten Superkombinator überführt werden kann. Die erweiterte Form, wie in [PJ87] vorgestellt, ist daraus jedoch schnell herleitbar, indem zusätzlich Abstraktionen geliefert werden, was wegen der Existenz der *cp*-Reduktion eine einfache Erweiterung ist.

Definition 3.5.1. *Ein Λ_{let}-Ausdruck t ist ein n-*stelliger Superkombinator *gdw. t die Form $\lambda x_1.\lambda x_2.\ldots\lambda x_n.s$ hat und*

1. *s keine Abstraktion ist,*
2. *$V_{free}(t) = \emptyset$, d.h. ein Superkombinator verfügt über keine freien Variablen,*
3. *alle in s vorkommenden Abstraktionen selber Superkombinatoren sind,*
4. *$n \geq 0$, d.h. auch let-Ausdrücke, Applikationen oder die Konstante* `choice` *können einen Superkombinator bilden.*

Beispiel 3.5.1. Beispiele für Superkombinatoren sind die Ausdrücke

$$\lambda x.\lambda y.xy,\ \texttt{let}\ y = \lambda x.x\ \texttt{in}\ y,\ \texttt{choice}\ (\lambda x.xx),\ (\lambda x.\lambda y.xy)z$$

Keine Superkombinatoren sind

$\texttt{let}\ y = \lambda x.x\ \texttt{in}\ \lambda z.y$, da y in $\lambda z.y$ frei vorkommt
$\lambda x.x(\lambda z.zx)$, da x in $(\lambda z.zx)$ frei vorkommt

Wir werden nun zeigen, daß jeder geschlossene Ausdruck in einen kontextuell äquivalenten Ausdruck gewandelt werden kann, der ein Superkombinator ist. Dazu wird definieren wir folgenden Algorithmus:

Algorithmus 3.5.1. Gegeben sei der geschlossene Ausdruck t. Prüfe ob eine Zerlegung $t \equiv C[t_s]$ mit $t_s \equiv \lambda x_1.\ldots\lambda x_n.s$ und $n > 0$ existiert, bei der

1. s keine Abstraktion ist und
2. alle in s vorkommenden Abstraktionen Superkombinatoren sind und
3. $V_{free}(t_s) \neq \emptyset$ gilt.

Falls eine solche Zerlegung existiert, dann sei $\{y_1,\dots,y_n\}$ die Menge aller in t_s frei vorkommenden Variablen. Bilde den Ausdruck

$$t' \equiv C[(\lambda z_1.\dots\lambda z_n.t_s[z_1/y_1,\dots,z_n/y_n])y_1\cdots y_n]$$

wobei alle z_i frische Variablen sind und wende das Verfahren auf t' erneut an. Ansonsten, ist der gegebene Ausdruck t ein Superkombinator und wir sind am Ende.

Das obige Verfahren ist eine vereinfachte Version des in [PJ87] vorgestellten Verfahrens. Der Vorgang des Abstrahierens der freien Variablen wird als Liften, der gesamte oben beschriebene Vorgang als Lambda-Lifting, bezeichnet. Es ist offensichtlich, daß das Lambda-Lifting stets nach endlicher Zeit terminiert und einen Superkombinator liefert, da in jedem Schritt stets eine Abstraktion verlorengeht, die die Superkombinator-Eigenschaft verletzt, und durch eine Abstraktion ersetzt wird, die einen Superkombinator bildet. Um die Korrektheit des Lambda-Liftings zu zeigen definieren wir die folgende Reduktion:

Definition 3.5.2. *Wir definieren folgende Reduktion*

$$(llift) \qquad C[t] \xrightarrow{llift} C[(\lambda z_1,\dots,z_n.t[z_1/y_1,\dots,z_n/y_n])y_1\dots y_n],$$
wobei alle z_i frische Variablen

Wir zeigen nun, daß eine $llift$-Reduktion die kontextuelle Äquivalenz erhält.

Lemma 3.5.1. *t, s seien zwei Ausdrücke mit $t \xrightarrow{llift} s$. Dann $t \sim_c s$.*

Beweis. Für alle Kontexte C ist folgende Entwicklung möglich:

$$\begin{array}{ll}
 & C[(\lambda z_1,\dots,z_n.D[z_1,\dots,z_1])y_1\dots y_n] \text{ wobei } (*)\\
\xrightarrow{lbeta} & C[(\texttt{let } z_1 = y_1 \texttt{ in } \lambda z_2,\dots,z_n.D[z_1,\dots,z_1])y_2\dots y_n]\\
\xrightarrow{lcv,*} & C[(\texttt{let } z_1 = y_1 \texttt{ in } \lambda z_2,\dots,z_n.D[y_1,\dots,y_1])y_2\dots y_n]\\
\xrightarrow{ldel} & C[(\lambda z_2,\dots,z_n.D[y_1,\dots,y_1])y_2\dots y_n]
\end{array}$$

(*) $D[\cdot,\dots,\cdot]$ sei der Multikontext, bei dem alle Löcher die Positionen angeben, wo ein Vorkommen der Variable y_1 im Rahmen der Substitution durch z_1 ersetzt wurde.

Alle oben benutzten Reduktionen erhalten die kontextuelle Äquivalenz. Durch Induktion über die Anzahl n an Abstraktionen und wiederholte Anwendung der obigen Entwicklung erhalten wir

$$C[(\lambda z_1, \dots, z_n.t[z_1/y_1, \dots, z_n/y_n])y_1 \dots y_n] \sim_c C[t]$$

und damit die Hypothese. □

Korollar 3.5.1. *Lambda-Lifting erhält die kontextuelle Äquivalenz, d.h. der oben beschriebene Algorithmus transformiert in einen kontextuell äquivalenten Ausdruck.*

Beweis. Das Liften der freien Variablen entspricht immer genau einer $llift$-Reduktion. Das Lambda-Lifting entspricht somit einer $(llift, *)$-Reduktion. Die kontextuelle Äquivalenz folgt dann direkt aus dem oben gezeigten Lemma und der Transitivität der kontextuellen Äquivalenz. □

Es sei an dieser Stelle angemerkt, daß das Liften von anderen Ausdrücken als Variablen beim Λ_{let}-Kalkül Beschränkungen unterliegt. Mit diesem Thema beschäftigt sich ein späterer Abschnitt mit dem Titel “fully-lazy Lambda-Lifting”. Abschließend soll das Lambda-Lifting anhand eines Beispiels noch einmal demonstriert werden:

Beispiel 3.5.2. Wir zeigen in Anlehnung an [PJ87] eine etwas stärkere Version, bei der zu einem Ausdruck der Form $L_R^*[s]$ geliftet wird, bei dem s weder eine Abstraktion noch ein let-Ausdruck ist. Zu liften sei der Ausdruck

$$t \equiv \texttt{let}\ x = t_x\ \texttt{in}\ \texttt{let}\ y = t_y\ \texttt{in}\ \lambda z.zxy$$

mit Superkombinatoren t_x und t_y. Wir liften zu

$$t_l \equiv \texttt{let}\ a = \lambda b.\lambda c.\lambda z.zbc\ \texttt{in}\ (\texttt{let}\ x = t_x\ \texttt{in}\ (\texttt{let}\ y = t_y\ \texttt{in}\ axy))$$

Wir zeigen $t \sim_c t_l$, indem wir mittels Reduktionen, die alle die kontextuelle Äquivalenz erhalten, t_l zu t reduzieren:

$$\begin{array}{rl}
& \texttt{let}\ a = \lambda b.\lambda c.\lambda z.zbc\ \texttt{in}\ \texttt{let}\ x = t_x\ \texttt{in}\ \texttt{let}\ y = t_y\ \texttt{in}\ (ax)y \\
\xrightarrow{cp} & \texttt{let}\ a = \lambda b.\lambda c.\lambda z.zbc\ \texttt{in}\ \texttt{let}\ x = t_x\ \texttt{in}\ \texttt{let}\ y = t_y\ \texttt{in}\ ((\lambda b.\lambda c.\lambda z.zbc)x)y \\
\xrightarrow{ldel} & \texttt{let}\ x = t_x\ \texttt{in}\ \texttt{let}\ y = t_y\ \texttt{in}\ ((\lambda b.\lambda c.\lambda z.zbc)x)y \\
\xrightarrow{lbeta} & \texttt{let}\ x = t_x\ \texttt{in}\ \texttt{let}\ y = t_y\ \texttt{in}\ (\texttt{let}\ b = x\ \texttt{in}\ (\lambda c.\lambda z.zbc))y \\
\xrightarrow{lcv} & \texttt{let}\ x = t_x\ \texttt{in}\ \texttt{let}\ y = t_y\ \texttt{in}\ (\texttt{let}\ b = x\ \texttt{in}\ (\lambda c.\lambda z.zxc))y \\
\xrightarrow{ldel} & \texttt{let}\ x = t_x\ \texttt{in}\ \texttt{let}\ y = t_y\ \texttt{in}\ (\lambda c.\lambda z.zxc)y \\
\xrightarrow{lbeta} & \texttt{let}\ x = t_x\ \texttt{in}\ \texttt{let}\ y = t_y\ \texttt{in}\ \texttt{let}\ c = y\ \texttt{in}\ \lambda z.zxc \\
\xrightarrow{lcv} & \texttt{let}\ x = t_x\ \texttt{in}\ \texttt{let}\ y = t_y\ \texttt{in}\ \texttt{let}\ c = y\ \texttt{in}\ \lambda z.zxy \\
\xrightarrow{ldel} & \texttt{let}\ x = t_x\ \texttt{in}\ \texttt{let}\ y = t_y\ \texttt{in}\ \lambda z.zxy
\end{array}$$

3.6 Abschließende Bemerkungen

Im Zentrum dieses Kapitels stand der Begriff der kontextuellen Äquivalenz. Zwei kontextuell äquivalente Ausdrücke wurden als gleichwertig bzw. austauschbar angesehen. Motivation für die Einführung der kontextuellen Äquivalenz war die Bereitstellung eines Gleichheitsverständnisses auf Ausdrucksebene, das die Durchführung nichttrivialer Programmtransformationen wie Lambda-Lifting ermöglicht. Die kontextuelle Äquivalenz ist definitorisch identisch zu der Beobachtungsäquivalenz (observational equivalence) aus [MS99] sowie [MSC99] und steht in der Tradition von [AO93]. Zu trennen ist die kontextuelle Äquivalenz jedoch von der in [Abr90] vorgestellten applikativen Bisimulation, da diese auf Kontextebene eine applikativ eingeschränkte Sicht beinhaltet.

Nach verschiedenen Definitionen wurde zu Beginn dieses Kapitels ein Kontextlemma gezeigt, mit dessen Hilfe Beweise der Erhaltung der kontextuellen Äquivalenz durch Reduktionen oder andere Transformationen auf Ausdrucksebene vereinfacht werden können. Der Begriff Kontextlemma wird dabei gegenüber dem allgemein üblichen Verständnis, das in der Tradition von [Mil77] steht, in einer verallgemeinerten Form benutzt, da das Loch eines W- oder R-Kontexts nicht unbedingt den no-Redex beinhalten muß. Im Anschluß wurde für eine Vielzahl von Reduktionen gezeigt, daß sie die kontextuelle Äquivalenz erhalten. Darunter befanden sich auch die 4 Reduktionen *llet*, *lapp*, *lbeta*, *cp* aus der Definition des Λ_{let}-Kalküls. Als besonders schwierig bzw. aufwendig erwies sich der Beweis der Erhaltung der kontextuellen Äquivalenz durch die *ucp*-Reduktion, die eindeutig gebundene Ausdrücke kopieren kann, sofern sich das Kopierziel nicht innerhalb einer Abstraktion befindet. Auf diesen Sachverhalt wurde bereits in der Einführung im Abschnitt über verwandte Arbeiten im Kontext der Arbeiten [MS99] und [MSC99], die auf syntaktisch eingeschränkten Ausdrucksmengen operieren, hingewiesen. Eine Diskussion, ob die dort getroffenen syntaktischen Einschränkungen tatsächlich so unproblematisch sind, wie von den Autoren behauptet, wäre durchaus von Interesse, wird in der vorliegenden Arbeit jedoch nicht geführt.
Zum Abschluß dieses Kapitels wurde gezeigt, daß Lambda-Lifting beim Λ_{let}-Kalkül eine zulässige Transformation auf Ausdrucksebene ist. Damit wurde für eine im Zusammenhang mit der Implementation verzögernd auswertender funktionaler Sprachen grundlegende Programmtransformation die Zulässigkeit gezeigt, da jeder geschlossene Λ_{let}-Ausdruck damit in einen kontextuell äquivalenten Λ_{let}-Ausdruck transformiert werden kann, der einen Superkombinator bildet.

Kapitel 4

deterministische Subausdrücke

Im letzten Kapitel wurde für eine Vielzahl von Reduktionen die Bewahrung der kontextuellen Äquivalenz gezeigt. Jedoch differenziert keine der dort vorgestellten Reduktionen danach, ob an einem Redex direkt oder indirekt die zur Repräsentation des Nichtdeterminismus benutzte Konstante `choice` beteiligt ist. Nun stellt sich die Frage, welchen Wert eine solche Differenzierung nach dem Vorkommen von `choice` haben könnte. Die Antwort auf diese Frage wird erhalten, wenn mit dem bestehenden Satz an Reduktionen versucht wird, beim Λ_{let}-Kalkül interessante Optimierungen wie das Liften ganzer Subausdrücke, eine einfache Erweiterung des zuvor vorgestellten Liftens von Variablen, zu leisten. Es zeigt sich schnell, daß die bestehenden kopierfähigen Reduktionen wie *cp*, *lcv* und *ucp* hier nicht ausreichen, wenn beispielsweise eine Applikation geliftet werden soll, weil dann das Kopieren einer Applikation unter eine Abstraktion erforderlich wird, was keine dieser Reduktionen leisten kann. Das Kopieren einer Applikation ist, wie bereits früher ausgeführt, wegen möglicher `choice`-Vorkommen nicht immer zulässig. Als Lösung dieser Problematik werden wir im Anschluß an diese Einleitung den Begriff des "deterministischen Subausdrucks" vereinbaren und zeigen, daß ein solcher Ausdruck unter Erhaltung der kontextuellen Äquivalenz stets kopierbar ist.

Eine weitere Fragestellung ist, welche Eigenschaften der Λ_{let}-Kalkül besitzt, wenn die Menge der betrachteten Ausdrücke auf Λ, die Ausdrucksmenge des klassischen Lambda-Kalküls, eingeschränkt wird. Wir würden ein Verwandtschaft zum klassischen Lambda-Kalkül erwarten und auch wünschen, da der klassische Lambda-Kalkül als Grundlage allen funktionalen Programmierens angesehen wird. Am Ende dieses Kapitels wird gezeigt werden, daß der klassische Lambda-Kalkül und der Λ_{let}-Kalkül bezogen auf die Menge aller geschlossenen Λ-Ausdrücke ein gleichartiges Terminierungsverhalten zeigen.

Zu Beginn des nächsten Abschnitts wird eine Reduktion mit dem Namen *detpar* eingeführt. Die *detpar*-Reduktion wird auf der Basis deterministischer Subausdrücke definiert und erzeugt eine *detpar*-Normalform. Die Idee der *detpar*-Normalform soll kurz anhand eines Beispiels näher erläutert werden, da sie wesentlich für das Verständnis der in diesem Kapitel benutzen Methodiken ist. Gegeben seien die beiden Ausdrücke

$$\begin{aligned} & t \equiv \texttt{let}\ x = \lambda a.a\ \texttt{in}\ \texttt{let}\ y = xx\ \texttt{in}\ (\lambda z.y)x \\ \text{und}\quad & s \equiv \texttt{let}\ x = \lambda a.a\ \texttt{in}\ (\lambda z.xx)x \end{aligned}$$

Indem bei beiden Ausdrücken durch (wiederholte) Substitution alle let-Bindungen aufgelöst werden, können wir einen gemeinsamen Ausdruck

$$t_R \equiv (\lambda z.(\lambda a.a)(\lambda a.a))(\lambda a.a)$$

erreichen. Dieser Ausdruck ist eine Art deterministische Normalform von s und t; über t_R kann eine Verbindung zwischen beiden Ausdrücken hergestellt werden. Im ersten Teil dieses Kapitels werden wir zeigen, daß zwei Ausdrücke mit derselben *detpar*-Normalform kontextuell äquivalent sind. Das zugehörige Lemma trägt den Namen *detpar*-Lemma, da der oben erfolgte Substitutionsprozess mittels der *detpar*-Reduktion erreicht wird. Das *detpar*-Lemma übernimmt eine ähnliche Funktion wie das Kontextlemma aus dem vorhergehenden Kapitel, es ist im wesentlichen ein Beweiswerkzeug. Mit seiner Hilfe wird für zwei Reduktionen, die auf der Basis deterministischer Subausdrücke formuliert sind, gezeigt werden, daß sie die kontextuelle Äquivalenz bewahren. Eine der dabei betrachteten Reduktionen trägt den Namen *dcp* und erlaubt das Kopieren deterministischer Subausdrücke. Die *dcp*-Reduktion ist eine Voraussetzung für die Betrachtung des fully-lazy Lambda-Lifting im darauffolgenden Abschnitt. Es wird gezeigt werden, daß fully-lazy Lambda-Lifting in einer beschränkten Form beim Λ_{let}-Kalkül möglich ist. Weitere Verwendung findet das *detpar*-Lemma bei der Betrachtung der Beziehung zwischen klassischem Lambda-Kalkül und dem Λ_{let}-Kalkül.

4.1 Definition der *detpar*-Reduktion

Wir werden nun den Begriff eines deterministischen Subausdrucks kennenlernen.

Definition 4.1.1. *t sei ein beliebiger Ausdruck. Ein Subausdruck t_s von t ist* deterministisch *gdw.*

1. *die Konstante* `choice` *in t_s nicht vorkommt und*
2. *t_s geschlossen ist oder alle freien Variablen in t_s sind in einem übergeordneten Kontext let-gebunden und binden einen deterministischen Subausdruck.*

$V_{det}(t)$ bezeichnet die Menge aller Variablen aus t, die einen deterministischen Subausdruck binden.

Ein Ausdruck ist nichtdeterministisch, wenn er nicht deterministisch im Sinne der zuletzt getroffenen Definition ist.

Beispiel 4.1.1. Gegeben sei der Ausdruck

$$t \equiv \texttt{let}\ x = \texttt{choice}\ \ \texttt{in}\ (\texttt{let}\ y = x\,(\lambda a.a)\ \texttt{in}\ (\texttt{let}\ z = \lambda b.b\ \texttt{in}\ (zz)))$$

Die Subausdrücke von t sind wie folgt einzuordnen:

nichtdeterministisch in t
`choice` , x, $x(\lambda a.a)$, a, b, $\texttt{let}\ y = x\,(\lambda a.a)\ \texttt{in}\ (\texttt{let}\ z = \lambda b.b\ \texttt{in}\ (zz))$

deterministisch in t
$\lambda a.a$, $\lambda b.b$, z, (zz), $(\texttt{let}\ z = \lambda b.b\ \texttt{in}\ (zz))$

Bei diesem Beispiel gilt somit $V_{det}(t) = \{z\}$.

Auf Basis deterministischer Subausdrücke definieren wird nun die *detpar*-Reduktion.

Definition 4.1.2. *Gegeben ein Ausdruck t. Wir definieren die Reduktion $\xrightarrow{detpar}$ wie folgt:*

- $x \xrightarrow{detpar} x$ *wobei x eine Variable oder die Konstante* `choice`
- *Wenn $\lambda x.s$ ein Subausdruck von t und $s \xrightarrow{detpar} s'$,*
  dann $\lambda x.s \xrightarrow{detpar} \lambda x.s'$
- *Wenn $(s_1 s_2)$ ein Subausdruck von t, $s_1 \xrightarrow{detpar} s_1'$, $s_2 \xrightarrow{detpar} s_2'$,*
  dann $s_1\,s_2 \xrightarrow{detpar} s_1'\,s_2'$
- *Wenn $(\texttt{let}\ x = t_x\ \texttt{in}\ s)$ ein Subausdruck von t, $t_x \xrightarrow{detpar} t_x'$, $s \xrightarrow{detpar} s'$,*
  dann $\texttt{let}\ x = t_x\ \texttt{in}\ s \xrightarrow{detpar} \begin{cases} s'[t_x'/x] & \text{falls } x \in V_{det}(t) \\ \texttt{let}\ x = t_x'\ \texttt{in}\ s' & \text{sonst} \end{cases}$

Die *detpar*-Reduktion ist eine parallel operierende Reduktion ähnlich *cppar*, jedoch ist *detpar* im Gegensatz zu *cppar* eindeutig und eröffnet keine Wahlmöglichkeiten. Offensichtlich löst eine *detpar*-Reduktion alle let-Bindungen der Form $\texttt{let}\ x = t_x\ \texttt{in}\ s$ auf, bei denen t_x ein deterministischer Subausdruck ist, indem

alle Vorkommen der Variable x in s durch t_x ersetzt werden. Wie bereits erwähnt ist die Idee, die bei der *detpar*-Reduktion verfolgt wird, eine *detpar*-Normalform zu erhalten, bei dem aller in let-Bindungen enthaltene Determinismus offengelegt wurde.
Das nun folgende Beispiel zeigt die Anwendung einer *detpar*-Reduktion.

Beispiel 4.1.2. t_d sei ein deterministischer Ausdruck und t_n ein nichtdeterministischer Ausdruck. Gegeben sei der Ausdruck

$$\texttt{let } a = t_d \texttt{ in let } b = t_n \texttt{ in let } c = aa \texttt{ in let } d = ac \texttt{ in } ddb$$

Gemäß Definition gilt $V_{det}(t) = \{a, c, d\}$. Wir entwickeln die *detpar*-Reduktion

$$\begin{array}{rl}
& ddb \xrightarrow{detpar} ddb,\ ac \xrightarrow{detpar} ac \\
\Rightarrow & \texttt{let } d = ac \texttt{ in } ddb \xrightarrow{detpar} (ac)(ac)b, \\
& aa \xrightarrow{detpar} aa \\
\Rightarrow & \texttt{let } c = aa \texttt{ in } (\texttt{let } d = ac \texttt{ in } ddb) \xrightarrow{detpar} (a(aa))(a(aa))b, \\
& t_n \xrightarrow{detpar} t_n' \\
\Rightarrow & \texttt{let } b = t_n \texttt{ in } (\texttt{let } c = aa \texttt{ in let } d = ac \texttt{ in } ddb) \xrightarrow{detpar} \texttt{let } b = t_n' \texttt{ in } (a(aa))(a(aa))b, \\
& t_d \xrightarrow{detpar} t_d' \\
\Rightarrow & \qquad \texttt{let } a = t_d \texttt{ in } (\texttt{let } b = t_n \texttt{ in let } c = aa \texttt{ in let } d = ac \texttt{ in } ddb) \\
& \xrightarrow{detpar} \texttt{let } b = t_n' \texttt{ in } (t_d(t_dt_d))(t_d(t_dt_d))b
\end{array}$$

Wir werden nun einige Eigenschaften der *detpar*-Reduktion in Form von Korollaren festhalten, die direkt aus der Definition hergeleitet werden können.

Korollar 4.1.1. *t, s seien zwei Ausdrücke mit $t \xrightarrow{detpar} s$. Dann gilt*

t deterministisch $\Rightarrow$ s deterministisch

t nichtdeterministisch $\Rightarrow$ s nichtdeterministisch

Beweis. Bei der Ersetzung einer Variable x in einem deterministischen Ausdruck entsteht wiederum ein deterministischer Ausdruck, da die zu ersetzende Variable x per Definition einen deterministischen Ausdruck band.

In einem nichtdeterministischen Ausdruck muß die Konstante `choice` oder eine Variable x vorkommen, die einen nichtdeterministischen Ausdruck bindet. Die Konstante `choice` bzw. die Variable x bleiben durch den Ersetzungsprozess unberührt und der Ausdruck somit nichtdeterministisch. □

Korollar 4.1.2. *detpar ist eindeutig, d.h. jedem Ausdruck t wird eindeutig ein Ausdruck t' zugeordnet.*

Korollar 4.1.3. *s sei ein beliebiger Ausdruck. Es gilt*

1. $L_R[s] \xrightarrow{detpar} \begin{cases} s'[x/t_x] & \text{falls in } L_R \text{ ein determ. Ausdruck gebunden} \\ L'_R[s'] & \text{sonst} \end{cases}$
   wobei $s \xrightarrow{detpar} s'$ *und* t_x *ein durch den* L_R*-Kontext gebundener deterministischer Ausdruck ist.*
2. $A_L[s] \xrightarrow{detpar} A'_L[s']$ *wobei* $s \xrightarrow{detpar} s'$
3. $L_R^m[s] \xrightarrow{detpar} L_R^n{}'[s'[\overrightarrow{x/t_x}]]$
   wobei $s \xrightarrow{detpar} s'$, $m \geq n$ *und* $\overrightarrow{x/t_x}$ *eine Folge von Ersetzungen von Variablen durch deterministische Subausdrücke ist.*
4. $A_L^n[s] \xrightarrow{detpar} A_L^n{}'[s']$ *wobei* $s \xrightarrow{detpar} s'$, $n > 0$

Beweis. Alle Aussagen folgen direkt aus der Definition von $\xrightarrow{detpar}$. Die letzten beiden Aussagen sind einfache transitive Erweiterungen der ersten beiden Aussagen. □

Im weiteren werden noch zwei Lemmata gezeigt, die wir in späteren Beweisen benötigen werden. Das erste Lemma zeigt, daß zwei Ausdrücke, die durch *detpar* auf dieselbe *detpar*-Normalform abgebildet werden, auch nach der Umhüllung mit einem beliebigen Kontext auf dieselbe *detpar*-Normalform abgebildet werden. Dies ist eine der Kompatibilität verwandte Eigenschaft. Das zweite Lemma steht im Kontext späterer no-Betrachtungen und zeigt Eigenschaften von *detpar* im Bezug auf W-Kontexte.

Lemma 4.1.1. *Gegeben drei Ausdrücke* t, t', s. *Wenn* $t \xrightarrow{detpar} s$ *und* $t' \xrightarrow{detpar} s$. *Dann existiert für alle Kontexte* C *genau ein Ausdruck* s_C, *so daß* $C[t] \xrightarrow{detpar} s_C$ *und* $C[t'] \xrightarrow{detpar} s_C$ *gilt.*

Beweis. Eine einfache strukturelle Induktion über den Aufbau des Kontexts C. Die Eindeutigkeit von s' folgt aus Korollar 4.1.2. □

Lemma 4.1.2. *s sei ein beliebiger Ausdruck. Folgendes gilt*

1. $W[s] \xrightarrow{detpar} W'[s'[\overrightarrow{x/t_x}]]]$ *wobei* $s \xrightarrow{detpar} s'$ *und* $\overrightarrow{x/t_x}$ *eine Folge von Ersetzungen von Variablen durch deterministische Subausdrücke ist.*
2. $W[x] \xrightarrow{detpar} W'[x]$, *wenn* x *in* W *frei vorkommt.*

Beweis. Eine einfache Induktion über die Struktur eines W-Kontext unter Zuhilfenahme von Korollar 4.1.3. Für einen strukturell nicht rekursiven W-Kontext, dieser hat die Form $L_R^*[A_L^*]$, ist die Hypothese offensichtlich.

Bleibt der Fall $L_R^*[\texttt{let}\ x = A_L^*\ \texttt{in}\ W[x]]$ eines strukturell rekursiven W-Kontext. Wir benutzen:

- $W[x] \xrightarrow{detpar} W'[x]$, wobei $x \in V_{free}(W)$ gemäß Induktionsvoraussetzung
- $A_L^*[s] \xrightarrow{detpar} A_L^*{}'[s']$, wobei $s \xrightarrow{detpar} s'$ gemäß Korollar 4.1.3

Daraus erhalten wir

$$\texttt{let}\ x = A_L^*[s]\ \texttt{in}\ W[x] \xrightarrow{detpar} \begin{cases} W'[x][A_L^*{}'[s']/x] \equiv W''[s'] & \text{falls } A_L^*[s] \text{ deterministisch} \\ \texttt{let}\ x = A_L^*{}'[s']\ \texttt{in}\ W'[x] \equiv W''[s'] & \text{sonst} \end{cases}$$

Daraus können wir in Kombination mit Korollar 4.1.3 schließen:

$$L_R^*[\texttt{let}\ x = A_L^*[s]\ \texttt{in}\ W[x]] \xrightarrow{detpar} L_R^*{}'[W''[s'][\overrightarrow{x/t_x}]] \equiv W'''[s'[\overrightarrow{x/t_x}]]$$

Entspricht s einer freien Variablen x, so erhalten wir $W'''[x]$, da x durch keine der Ersetzungen verändert werden kann. □

Bei obigem Lemma wird ausgiebig davon Gebrauch gemacht, daß für jeden W-Kontext und beliebige Kontexte A_L^* und L_R^* die Eigenschaft $W[A_L^*] \equiv W'$ sowie $L_R^*[W] \equiv W'$ gilt, d.h. das äußere Anfügen von L_R^*-Kontexten und das innere Anfügen von A_L^*-Kontexten einen W-Kontext stets strukturell bewahrt.

4.2 Eigenschaften von $(i, lbeta, *)$-Reduktionen

lbeta-Reduktionen werden später in diesem Abschnitt eine zentrale Rolle übernehmen. Wir werden die Vertauschungs- sowie Gabeldiagramme einer $(i, lbeta, *)$-Reduktion benötigen; diese werden in diesem Abschnitt vorgestellt. Dieser Abschnitt ist ein Einschub und hat keinen weiteren Hintergrund als die Beweise der späteren Abschnitte vorzubereiten.

Vertauschungsdiagramme einer $(i, lbeta, *)$-Reduktion

Lemma 4.2.1. *Jede $(i, lbeta, *)$-Reduktion kann gemäß eines der folgenden Vertauschungsdiagramme über eine no-Reduktion hinwegbewegt werden:*

1. $\xrightarrow{i,lbeta,*} \circ \xrightarrow{no,+} \quad \rightsquigarrow \quad \xrightarrow{no,+} \circ \xrightarrow{i,lbeta,*}$
2. $\xrightarrow{i,lbeta,*} \circ \xrightarrow{no,lbeta,+} \quad \rightsquigarrow \quad \xrightarrow{no,lbeta,+} \circ \xrightarrow{i,lbeta,*}$

Beweis. Zunächst wird die Hypothese für eine einzelne $(i, lbeta)$-Reduktion gezeigt; der Beweis hierfür kann auf einfache Weise aus dem in Lemma 2.4.8 geführten hergeleitet werden. Durch wiederholte Anwendung des zuletzt gezeigten Teillemma werden alle $(i, lbeta)$-Reduktionen nacheinander auf die rechte Seite der no-Reduktion bewegt. Die Vertauschungsdiagramme aus Lemma 2.4.7 zeigen, daß dabei eine Gruppe von $(no, lbeta)$-Reduktion erhalten bleibt. □

Gabeldiagramme einer $(i, lbeta, *)$-Reduktion

Lemma 4.2.2. *Die $(i, lbeta)$-Reduktion verfügt über den folgenden vollständigen Satz von Gabeldiagrammen:*

i): $\xleftarrow{no,a} \circ \xrightarrow{i,lbeta} \;\rightsquigarrow\; \xrightarrow{i,lbeta} \circ \xleftarrow{no,a}$
für $a \in \{llet, lapp, lbeta, cp, nd\}$

ii): $\xleftarrow{no,a} \circ \xrightarrow{i,lbeta} \;\rightsquigarrow\; \xrightarrow{no,lbeta} \circ \xleftarrow{no,a}$
für $a \in \{llet, lapp, lbeta, cp, nd\}$

iii): $\xleftarrow{no,cp} \circ \xrightarrow{i,lbeta} \;\rightsquigarrow\; \xrightarrow{i,lbeta} \circ \xrightarrow{i,lbeta} \circ \xleftarrow{no,cp}$

iv): $\xleftarrow{no,cp} \circ \xrightarrow{i,lbeta} \;\rightsquigarrow\; \xrightarrow{no,lbeta} \circ \xrightarrow{i,lbeta} \circ \xleftarrow{no,cp}$

Beweis. Ein einfache Herleitung der Gabeldiagramme aus den Vertauschungsdiagrammen von Lemma 2.4.7 gemäß der in Abschnitt 2.4.1 beschriebenen Methodik. □

Die oben gezeigten Gabeldiagramme werden nun auf no-Reduktionen, die aus mehr als einem Schritt bestehen ausgedehnt:

Lemma 4.2.3. *Gegeben drei Ausdrücke t, t', s mit $t \xrightarrow{no,m} t'$ und $t \xrightarrow{i,lbeta} s$, wobei m die Länge der no-Reduktion von t zu t' bezeichnet. Dann existiert ein Ausdruck t'', so daß die Reduktionszusammenhänge des folgenden Diagramms erfüllt werden:*

$$\begin{tikzcd}
t \arrow[r, "{i, lbeta}"] \arrow[d, "{no, m}"'] & s \arrow[dd, dashed, "{no, n}"] \\
t' \arrow[d, dashed, "{no, lbeta, *}"'] & \\
t'' \arrow[r, dashed, "{i, lbeta, *}"'] & \circ
\end{tikzcd}
\qquad \text{wobei } n \leq m$$

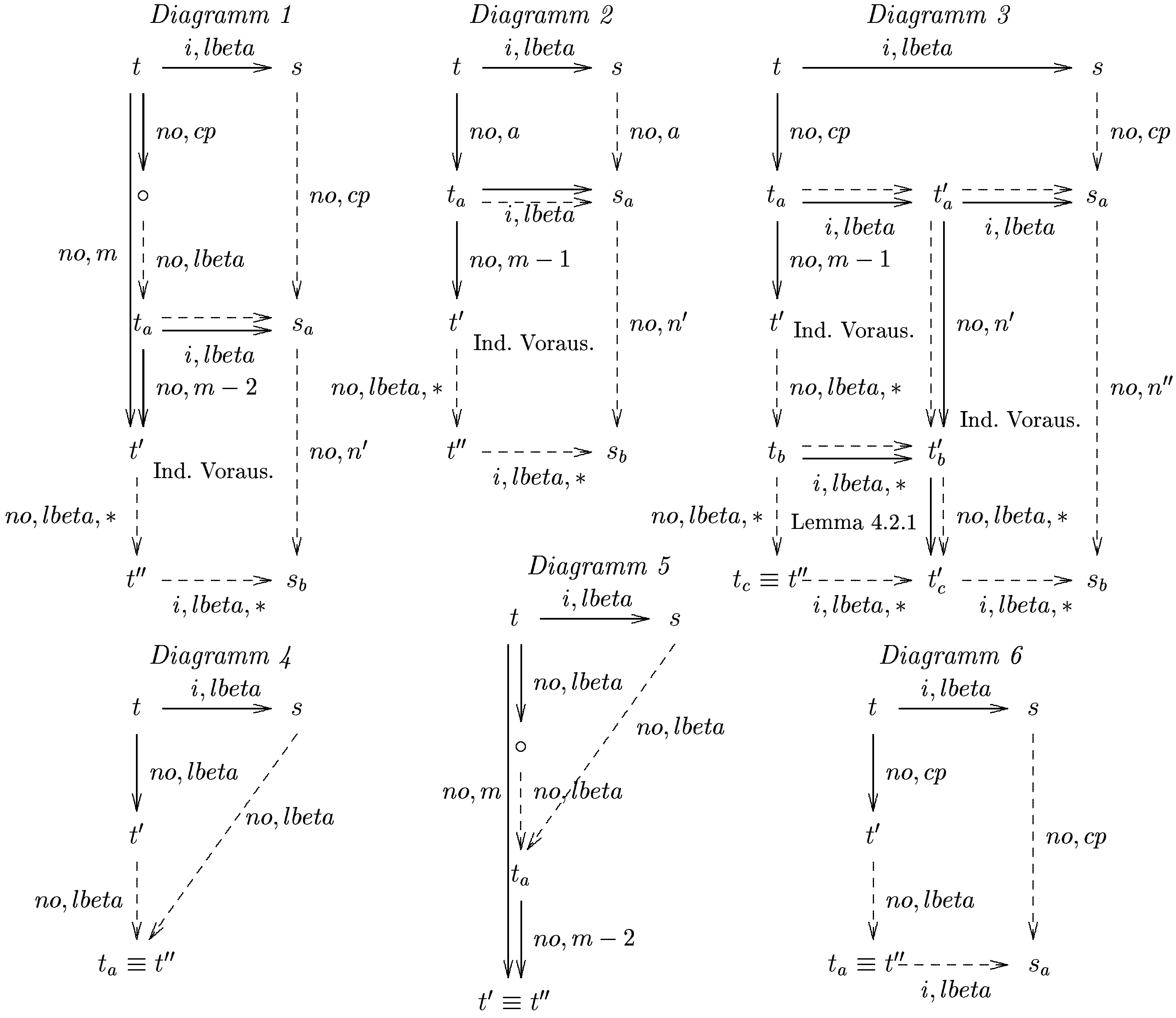


Abbildung 4.1: Reduktionsdiagramme zu Lemma 4.2.3

Beweis. Der Beweis erfolgt durch Induktion über die Anzahl der no-Schritte m.

Induktionsbeginn:
Für $m = 0$ wählen wir $t'' \equiv t$; alles weitere ist offensichtlich.

Induktionsschritt:
Bleibt $t \xrightarrow{no} t_a \xrightarrow{no,m-1} t'$ mit $m > 0$:
Die Korrektheit ist für alle Gabeldiagramme aus Lemma 4.2.1 zu zeigen:

i) Die entstehende Situation zeigt Diagramm 2 aus Abbildung 4.1. Bei der Reduktion $t_a \xrightarrow{i,lbeta} s_a$ können wir die Induktionsvoraussetzung benutzen, da die no-Reduktion von t_a zu t' kürzer als die Ausgangsreduktion ist. Gemäß Induktionsvoraussetzung gilt ferner $n' \leq m-1$. Wegen des Hinzukommens jeweils einer (no, a)-Reduktion auf beiden Seiten gilt somit $n \leq m$.

ii) Die entstehenden Möglichkeiten zeigen die Diagramme 4 und 5 aus Abbildung 4.1, je nachdem ob die no-Reduktion innerhalb eines Gabeldiagramms endet oder nicht. Alles weitere kann direkt aus den beiden Diagrammen abgelesen werden.

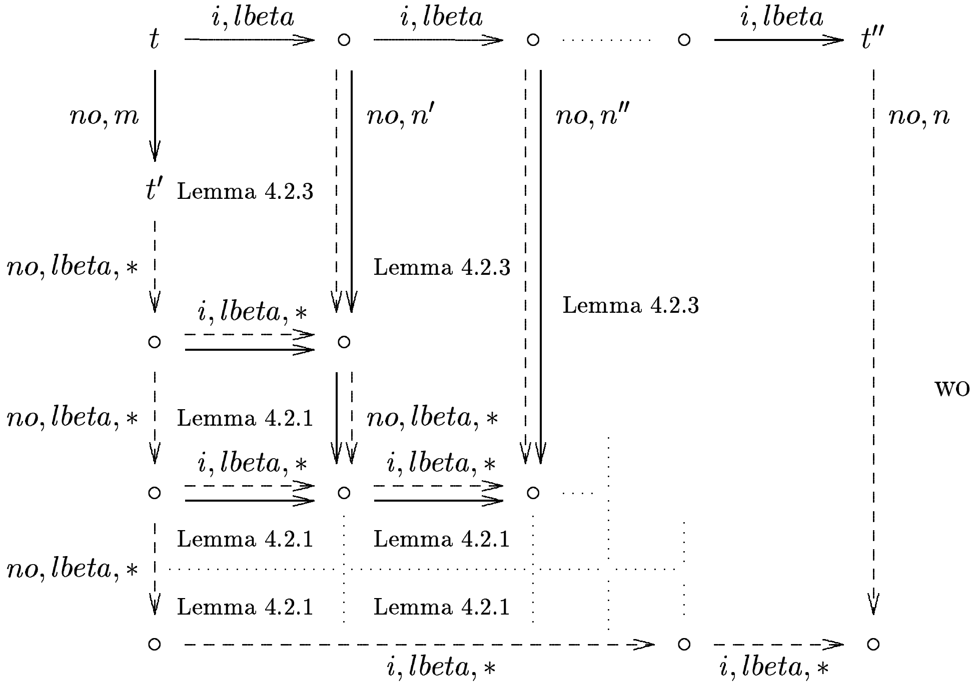
iii) Dies ist der aufwendigste Fall, er soll daher näher betrachtet werden. Es entsteht die in Diagramm 3 aus Abbildung 4.1 gezeigte Situation. Bei der Gabel $t' \xleftarrow{no,m-1} t_a \xrightarrow{i,lbeta} t'_a$ wenden wir Induktionsvoraussetzung an und erhalten die Existenz zweier Ausdrücke t_b und t'_b sowie die Aussage $n' \leq m-1$. Daraus folgt, daß wir die Induktionsvoraussetzung ein zweites Mal auf die Gabel $t'_b \xleftarrow{no,n'} t'_a \xrightarrow{i,lbeta} s_a$ anwenden können und wir erhalten die Existenz zweier Ausdrücke t'_c und s_b sowie die Aussage $n'' \leq n'$. Aus der zuvor in Lemma 4.2.1 gezeigten Vertauschungsaussage folgt die Existenz des Ausdrucks t_c. Wir ziehen die Ungleichungen über die Längen der no-Reduktionen zusammen und erhalten $n'' \leq n' \leq m-1$. Wir addieren die eine (no, cp)-Reduktion auf beiden Seiten und erhalten $n = n'' + 1 \leq m$. Wir wählen $t'' \equiv t_c$ und haben die Hypothese gezeigt.

iv) Die Situation bei Gabeldiagramm *iv)* geben die Diagramme 1 und 6 wieder. Wie bei *ii)* ist danach zu differenzieren, ob die gegebene no-Reduktion innerhalb eines Gabeldiagramms endet oder nicht. Falls die no-Reduktion über das Gabeldiagramm hinausläuft (Diagramm 1), so gilt gemäß Induktionsvoraussetzung $n' \leq m-2$. Daraus folgt $n = n'+1 \leq m-1 \leq m$. Alles weitere kann direkt aus den beiden Diagrammen abgelesen werden.

□

Das untenstehende Lemma ist eine einfache Erweiterung des zuvor bewiesenen Lemmas.

Lemma 4.2.4. *Gegeben drei Ausdrücke* t, t', s *mit* $t \xrightarrow{no,m} t'$ *und* $t \xrightarrow{i,lbeta} s$, *wobei* m *die Länge der no-Reduktion von* t *zu* t' *bezeichnet. Dann existiert ein Ausdruck* t'', *so daß die Reduktionszusammenhänge des folgenden Diagramms erfüllt werden:*



wobei $m \geq n' \geq n'' \cdots \geq n$

Abbildung 4.2: Grafische Repräsentation des Beweises von Lemma 4.2.4

$$\begin{tikzcd}
t \arrow[r, "{i,lbeta,*}"] \arrow[d, "{no,m}"'] & s \arrow[dd, dashed, "{no,n}"] \\
t' \arrow[d, dashed, "{no,lbeta,*}"'] & \\
t'' \arrow[r, dashed, "{i,lbeta,*}"'] & \circ
\end{tikzcd}$$

wobei $n \leq m$

Beweis. Durch eine transitive Anwendung von Lemma 4.2.3 in Kombination mit Lemma 4.2.1 über alle einzelnen Reduktionen der gegebenen $(i, lbeta, *)$-Reduktion hinweg. In Abbildung 4.2 ist grafisch dargestellt, wie die einzelnen Lemmata zu kombinieren sind, um die Hypothese zu zeigen. □

Wir werden im späteren Verlauf dieses Kapitels auch die all-quantifizierte Aussage benötigen, daß zwischen zwei mittels einer $(i, lbeta, *)$-Reduktion ineinander überführbaren Ausdrücken die Eigenschaft der Terminierung aller no-Reduktionen wechselseitig übertragbar ist. Zu diesem Zweck bilden wir die nun folgenden Lemmata.

Lemma 4.2.5. *Gegeben zwei Ausdrücke t, s mit $t \xrightarrow{i,lbeta} s$. Dann gilt $t\not\Uparrow \Rightarrow s\not\Uparrow$ und die längste von s ausgehende no-Reduktion ist kürzer oder gleich der längsten von t ausgehenden no-Reduktion.*

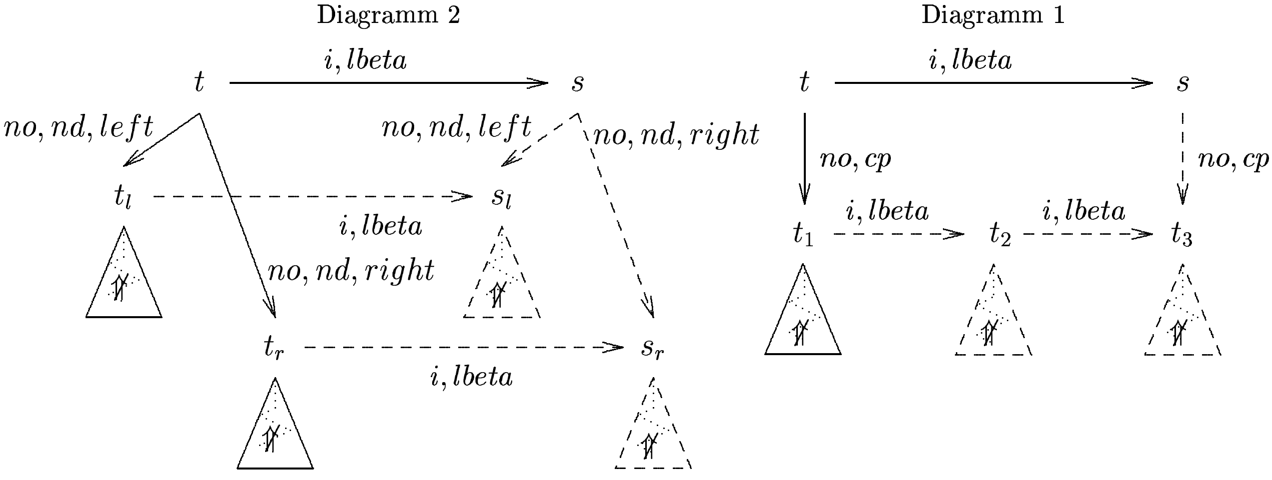


Abbildung 4.3:

Beweis. Wir führen den Beweis durch Induktion über die Länge der längsten von t ausgehenden no-Reduktion.

Induktionsanfang
Die längste von t ausgehende no-Reduktion hat die Länge 0, d.h. t befindet sich in WHNF. Daraus folgt gemäß Lemma 2.3.4, daß sich auch t' in WHNF befindet. Die Hypothese ist damit für diesen Fall gezeigt.

Induktionsschritt
Wir benutzen die Gabeldiagramme einer $(i, lbeta)$-Reduktion aus Lemma 4.2.2 und differenzieren nach dem Typ des no-Redex von t. Zwei Fälle werden stellvertretend betrachtet, alle übrigen Fälle sind simplifizierte Formen von diesen beiden:

1. t besitzt einen (no, cp)-Redex derart, daß Diagramm *iii)* aus Lemma 4.2.2 entsteht:
   In diesem Fall entstehen die in Diagramm 1 aus Abbildung 4.3 gezeigten Reduktionszusammenhänge. Die längste von t_1 ausgehend no-Reduktion ist um eine Reduktion kürzer als die Längste von t ausgehende, da wegen des (no, cp)-Redex nur eine no-Reduktion von t ausgeht. Da gemäß Voraussetzung alle von t ausgehenden no-Reduktionen endlich sind, sind dies auch alle von t_1 ausgehenden. Somit sind gemäß Induktionsvoraussetzung auch alle von t_2 ausgehenden no-Reduktionen endlich und die längste von t_2 ausgehende no-Reduktion ist nicht länger als die Längste von t_1 ausgehende. Dies erlaubt es uns ein zweites Mal die Induktionsvoraussetzung anwenden zu können und wir erhalten, daß alle von t_3 ausgehenden no-Reduktionen endlich sind und die längste von t_3 ausgehende no-Reduktion nicht länger als die Längste von t_2 und damit auch nicht länger als die Längste von t_1

ausgehende ist. Die Hypothese folgt nun daraus, daß von s nur eine no-Reduktion ausgeht, die zu t_3 führt und einen einzelnen no-Schritt umfaßt, was die Einhaltung der Längenbedingung sichert.

2. t besitzt einen no-Redex vom Typ nd:
   Es entstehen die in Diagramm2 aus Abbildung 4.3 gezeigten Reduktionszusammenhänge. Die längste von t_l wie t_r ausgehende no-Reduktion ist kürzer als die längste von t ausgehende. Dies erlaubt es uns bei t_l wie t_r die Induktionsvoraussetzung anwenden zu können und wir erhalten, daß alle von s_l wie s_r ausgehenden no-Reduktion endlich sind und zusätzlich keine der beiden länger als die längste von t_l oder t_r ausgehende no-Reduktion ist. Daraus folgt wiederum direkt die Hypothese, da von s aus in einem no-Schritt keine weiteren Ausdrücke außer s_l und s_r erreichbar sind.

Bei beiden oben betrachteten Fällen, bleibt die Länge der längsten no-Reduktion gleich. Zu einer Verkürzung der von s ausgehenden no-Reduktion kann jedoch Diagramm *iv)* aus Lemma 4.2.2 führen. □

Es sei angemerkt, daß die in der Hypothese des obigen Lemmas beinhaltete Implikation in isolierter Form, d.h. ohne die Verringerung des Längenmaßes, direkt aus der in Satz 3.3.1 bewiesene kontextuellen Äquivalenz einer $(i, lbeta)$-Reduktion gefolgert werden kann. Wir werden die Verringerung des Längenmaßes jedoch im Kontext späterer induktiver Argumentationen explizit benötigen, weshalb obiges Lemma zusätzlich erforderlich wurde.

Korollar 4.2.1. *Gegeben zwei Ausdrücke t, s mit $t \xrightarrow{i,lbeta,*} s$. Dann gilt $t{\not\Uparrow} \Rightarrow s{\not\Uparrow}$ und die längste von s ausgehende no-Reduktion ist kürzer oder gleich der längsten von t ausgehenden no-Reduktion.*

Beweis. Eine einfache reflexiv-transitive Erweiterung von Lemma 4.2.5. □

Wir zeigen abschließend noch die jeweilige Umkehrung der beiden obigen implikativen Aussagen.

Lemma 4.2.6. *Gegeben zwei Ausdrücke t, s mit $t \xrightarrow{i,lbeta} s$. Dann gilt $s{\not\Uparrow} \Rightarrow t{\not\Uparrow}$.*

Beweis. Dies folgt mittels einer einfachen Widerspruchsüberlegung aus den Divergenzeigenschaften der in Satz 3.3.1 gezeigten kontextuellen Äquivalenz einer $(i, lbeta)$-Reduktion. □

Korollar 4.2.2. *Gegeben zwei Ausdrücke t, s mit $t \xrightarrow{i,lbeta,*} s$. Dann gilt $s{\not\Uparrow} \Rightarrow t{\not\Uparrow}$.*

Beweis. Eine einfache reflexiv-transitive Erweiterung des zuletzt gezeigten Lemmas. □

4.3 Entwicklung des *detpar*-Lemma

Für die *detpar*-Reduktion werden nun zunächst Gabeldiagramme entwickelt, wie dies in den Kapiteln zuvor für andere Reduktionen geschehen ist. Dazu ist es, wie wir später sehen werden, sinnvoll, für die verschiedenen Redextypen des Λ_{let}-Kalkül eine Einordnung nach ihrem "Grad an Nichtdeterminismus" vorzunehmen. Der "Grad an Nichtdeterminismus" wird dadurch festgelegt, welche Subausdrücke eines Redex nichtdeterministisch sind und welche nicht. Die folgende Definition leistet diese Einordnung.

Definition 4.3.1. *t sei ein geschlossener Ausdruck und t_s ein nichtdeterministischer Subausdruck von t, der zugleich ein $\{llet, lapp, cp, lbeta, nd\}$-Redex ist.*

t_s heißt stark nichtdeterministischer Redex *falls t_s eine der folgenden Eigenschaften erfüllt:*

$(lbeta, nd)$ *$t_s \equiv rs$ und r ist eine Abstraktion oder die Konstante* `choice` *und s ist ein nichtdeterministischer Subausdruck.*

(cp) *$t_s \equiv \texttt{let}\ x = t_x\ \texttt{in}\ C[x]$ und t_x ist ein nichtdeterministischer Subausdruck sowie eine Abstraktion oder die Konstante* `choice` *.*

$(llet)$ *$t_s \equiv \texttt{let}\ x = (\texttt{let}\ y = t_y\ \texttt{in}\ t_x)\ \texttt{in}\ C[x]$ und t_x sowie t_y sind nichtdeterministische Subausdrücke.*

$(lapp)$ *$t_s \equiv (\texttt{let}\ x = t_x\ \texttt{in}\ r)s$ und t_x ist ein nichtdeterministischer Subausdruck.*

t_s heiß beschränkt nichtdeterministischer Redex *falls t_s eine der folgenden Eigenschaften erfüllt:*

$(lbeta, nd)$ *$t_s \equiv rs$ und r ist ein Abstraktion oder die Konstante* `choice` *und s ist ein deterministischer Subausdruck.*

$(llet)$ *$t_s \equiv \texttt{let}\ x = (\texttt{let}\ y = t_y\ \texttt{in}\ t_x)\ \texttt{in}\ C[x]$ und t_y ist ein nichtdeterministischer Subausdruck und t_x ist ein deterministischer Subausdruck.*

t_s heißt schwach nichtdeterministischer Redex *falls t_s weder ein stark noch ein beschränkt nichtdeterministischer Redex ist.*

Korollar 4.3.1. *Ein nd-Redex ist niemals schwach nichtdeterministisch oder deterministisch. Ein lbeta-Redex ist niemals schwach nichtdeterministisch.*

Wir führen nun eine Notation ein, die bei der Spezifikation einer Reduktion zuläßt, mitzuspezifizieren, welchen Grad an Nichtdeterminismus der reduzierte Redex besitzt.

Notation 4.3.1. *t sei ein beliebiger Ausdruck. Wir schreiben:*

$t \xrightarrow{no,strong} \circ$ *falls bei der Reduzierung des no-Redex in t ein stark nichtdeterministischer no-Redex reduziert wird.*

$t \xrightarrow{no,middle} \circ$ *falls bei der Reduzierung des no-Redex in t ein beschränkt nichtdeterministischer no-Redex reduziert wird.*

$t \xrightarrow{no,weak} \circ$ *falls bei der Reduzierung des no-Redex in t ein schwach nichtdeterministischer oder* ***deterministischer*** *no-Redex reduziert wird.*

$t \xrightarrow{no,det} \circ$ *falls bei der Reduzierung des no-Redex in t ein deterministischer no-Redex reduziert wird.*

Die Einteilung der Redizes in verschiedene Grade an Nichtdeterminismus erfolgt, da abhängig von dem Grad an Nichtdeterminismus unterschiedliche Gabeldiagramme für die *detpar*-Reduktion entstehen. Wir werden nun zeigen, welche Gabeldiagramme bei den verschiedenen nichtdeterministischen no-Redex-Formen möglich sind.

Lemma 4.3.1. *Die detpar-Reduktion verfügt über den folgenden vollständigen Satz von Gabeldiagrammen:*

$\xleftarrow{no,a,strong} \circ \xrightarrow{detpar} \quad \rightsquigarrow \quad \xrightarrow{detpar} \circ \xleftarrow{no,a,strong}$ *für* $a \in \{llet, lapp, lbeta, cp, nd\}$

$\xleftarrow{no,a,middle} \circ \xrightarrow{detpar} \quad \rightsquigarrow \quad \xrightarrow{detpar} \circ \xleftarrow{detpar} \circ \xleftarrow{no,a,middle}$ *für* $a \in \{llet, lbeta, nd\}$

$\xleftarrow{no,weak} \circ \xrightarrow{detpar} \quad \rightsquigarrow \quad \xrightarrow{detpar}$ *für* $a \in \{llet, lapp, cp\}$

Beweis. Eine einfache Betrachtung aller no-Reduktionen der verschiedenen Grade an Nichtdeterminismus unter Zuhilfenahme der aus Abschnitt 2.3.1 bekannten Eigenschaften der no-Reduktion sowie von Korollar 4.1.1 und Lemma 4.1.2:

$(no, llet)$, stark nichtdeterministisch

$$\begin{tikzcd}
L_R^*[\texttt{let } x = \texttt{let } y = t_y \texttt{ in } t_x \texttt{ in } W[x]] \arrow[r, "detpar"] \arrow[d, "{no,llet,strong}"] & L_R^*{}'[\texttt{let } x = \texttt{let } y = t_y' \texttt{ in } t_x' \texttt{ in } W'[x]] \arrow[d, "{no,llet,strong}"] \\
L_R^*[\texttt{let } y = t_y \texttt{ in let } x = t_x \texttt{ in } W[x]] \arrow[r, "detpar"] & L_R^*{}'[\texttt{let } y = t_y' \texttt{ in let } x = t_x' \texttt{ in } W'[x]]
\end{tikzcd}$$

Alle übrigen Reduktionstypen können nach demselben Schema gezeigt werden, der no-Redex kann bei einem stark nichtdeterministischen Subausdruck niemals aufgelöst werden.

Beschränkt nichtdeterministischer no-Redex:

$(no, llet)$

$$\begin{tikzcd}
L_R^*[\mathtt{let}\ x = \mathtt{let}\ y = t_y\ \mathtt{in}\ t_x\ \mathtt{in}\ W[x]] \arrow[rr, "detpar"] \arrow[d, "{no,llet,middle}"] & & L_R^*{}'[\mathtt{let}\ x = \mathtt{let}\ y = t'_y\ \mathtt{in}\ t'_x\ \mathtt{in}\ W'[x]] \arrow[d, "{no,llet,middle}"] \\
L_R^*[\mathtt{let}\ y = t_y\ \mathtt{in}\ \mathtt{let}\ x = t_x\ \mathtt{in}\ W[x]] \arrow[dr, "detpar"] & & L_R^*{}'[\mathtt{let}\ y = t'_y\ \mathtt{in}\ \mathtt{let}\ x = t'_x\ \mathtt{in}\ W'[x]] \arrow[dl, "detpar"] \\
 & L_R^*{}'[\mathtt{let}\ y = t'_y\ \mathtt{in}\ W''[t'_x]] &
\end{tikzcd}$$

$$\begin{tikzcd}
(no, lbeta) & W[(\lambda x.s)t] \arrow[rr, "detpar"] \arrow[d, "{no,lbeta,middle}"] & & W'[(\lambda x.s')t'] \arrow[d, "{no,lbeta,middle}"] \\
 & W[\mathtt{let}\ x = t\ \mathtt{in}\ s] \arrow[dr, "detpar"] & & W'[\mathtt{let}\ x = t'\ \mathtt{in}\ s'] \arrow[dl, "detpar"] \\
 & & W'[s'[t'/x]] &
\end{tikzcd}$$

$$\begin{tikzcd}
(no, nd) & W[\mathtt{choice}\ t] \arrow[rr, "detpar"] \arrow[d, "{no,nd,left,middle}"] & & W'[\mathtt{choice}\ t'] \arrow[d, "{no,nd,left,middle}"] \\
 & W[(\lambda x.\lambda y.x)t] \arrow[dr, "detpar"] & & W'[(\lambda x.\lambda y.x)t'] \arrow[dl, "detpar"] \\
 & & W''[(\lambda x.\lambda y.x)t'] &
\end{tikzcd}$$

$(no, nd, right)$ erfolgt nach demselben Schema; es ist zu beachten, daß der Ausdruck `choice` t durch die no-Reduktion deterministisch und damit in W kopierbar werden kann.

Schwach nichtdeterministischer oder deterministischer no-Redex:

$$\begin{tikzcd}
(no, llet) & L_R^*[\mathtt{let}\ x = \mathtt{let}\ y = t_y\ \mathtt{in}\ t_x\ \mathtt{in}\ W[x]] \arrow[dr, "detpar"] \arrow[dd, "{no,llet,weak}"] & \\
 & & L_R^*{}'[W'[t'_x]] \\
 & L_R^*[\mathtt{let}\ y = t_y\ \mathtt{in}\ \mathtt{let}\ x = t_x\ \mathtt{in}\ W[x]] \arrow[ur, "detpar"] &
\end{tikzcd}$$

$$\begin{tikzcd}
(no, lapp) & W[(\mathtt{let}\ x = t_x\ \mathtt{in}\ s)t] \arrow[dr, "detpar"] \arrow[dd, "{no,lapp,weak}"] & \\
 & & W'[s't'] \\
 & W[\mathtt{let}\ x = t_x\ \mathtt{in}\ (st)] \arrow[ur, "detpar"] &
\end{tikzcd}$$

$$\begin{tikzcd}
(no,cp) & L^*_R[\texttt{let}\ x = t_x\ \texttt{in}\ W[x]] \arrow[dr, "detpar"] \arrow[dd, "no{,}cp{,}weak"] & \\
 & & L^*_R{}'[W'[t'_x]] \\
 & L^*_R[\texttt{let}\ x = t_x\ \texttt{in}\ W[t_x]] \arrow[ur, "detpar"'] &
\end{tikzcd}$$

□

Korollar 4.3.2. *Wir können folgende transitive Erweiterungen der zuletzt gezeigten Gabeldiagramme bilden:*

$$\xleftarrow{no,\{llet,lapp,cp\},weak,*} \circ \xrightarrow{detpar} \quad \rightsquigarrow \quad \xrightarrow{detpar}$$

Korollar 4.3.3. *t sei ein beliebiger deterministischer Ausdruck. Wenn $t \xrightarrow{detpar} t'$, so kann t' nur einen lbeta-Redex als no-Redex besitzen.*

Wir werden nun zwei Lemmata entwickeln, die die wechselseitige Übertragbarkeit einer WHNF bei Anwendung einer *detpar*-Reduktion zeigen.

Lemma 4.3.2. *Gegeben ein geschlossener Ausdruck t in WHNF. Wenn $t \xrightarrow{detpar} t'$, dann ist auch t' in WHNF.*

Beweis. Dies folgt direkt aus Lemma 2.3.1 und Korollar 4.1.3. Jeder Ausdruck in WHNF hat die Form $L^*_R[r]$ mit r eine Abstraktion oder die Konstante `choice` . $L^*_R[r] \xrightarrow{detpar} L^*_R{}'[r']$ mit r' eine Abstraktion oder die Konstante `choice` . $L^*_R{}'[r']$ befindet sich wiederum in WHNF. □

Lemma 4.3.3. *Gegeben ein geschlossener Ausdruck t. Wenn $t \xrightarrow{detpar} s$ und s ist in WHNF, dann existiert ein Ausdruck t' in WHNF, so daß $t \xrightarrow{no,\{llet,lapp,cp\},*} t'$.*

Beweis. Folgende Widerspruchsüberlegungen zeigen, daß t in WHNF oder der no-Redex von t schwach nicht deterministisch oder deterministisch und vom Typ *llet*, *lapp* oder *cp* ist:

- Angenommen t hat einen stark nichtdeterministischen no-Redex. Dann hat gemäß Lemma 4.3.1 auch s einen stark nichtdeterministischen no-Redex. Dies ist ein Widerspruch zu der Voraussetzung, daß sich s in WHNF befindet.

- Dieselbe Argumentation läßt sich für eine beschränkt nichtdeterministischen no-Redex in Verbindung mit Lemma 4.3.1 und für deterministischen *lbeta*-Redex in Verbindung mit Lemma 4.3.5 führen.

Besitzt t einen no-Redex, so folgt aus den Gabeldiagrammen von Lemma 4.3.1, daß nach einer no-Reduktion $t \xrightarrow{no,\{llet,lapp,cp\},weak} t_a$ wiederum $t_a \xrightarrow{detpar} s$ gilt. Die Argumentation läßt sich nun rekursiv auf t_a anwenden. Es bleibt zu zeigen, daß dieser rekursive Prozess irgendwann immer stoppt. Jede *llet*, *lapp*, *cp*-Reduktion verringert das aus Abschnitt 3.4.3 bekannte Maß ξ. Daraus folgt, daß eine no-Reduktion, die nur aus *llet*, *lapp*, *cp*-Reduktionen besteht, endlich beschränkt ist, und damit der Prozess stets bei einem Ausdruck stoppt, auf den keine no-Reduktion mehr anwendbar ist und der damit in WHNF ist.

□

Korollar 4.3.3 deutet bereits an, daß wir bei einem deterministischen $(no, lbeta)$-Redex auf eine deutlich schwierigere Situation treffen als bei den anderen Typen von no-Redizes. Ein deterministischer $(no, lbeta)$-Redex bleibt nach der Anwendung von *detpar* bestehen und kann sich dabei sogar vervielfachen, wie das folgende Beispiel zeigt:

Beispiel 4.3.1. s, t seien zwei deterministische Ausdrücke mit $s \xrightarrow{detpar} s$, $t \xrightarrow{detpar} t$ und $(\texttt{let } x = t \texttt{ in } s) \xrightarrow{detpar} t'$.
Bei dem Ausdruck $\texttt{let } x = (\lambda y.s)t \texttt{ in } (xx)$ gilt folgendes:

$$\begin{tikzcd}
\texttt{let } x = (\lambda y.s)t \texttt{ in } (xx) \arrow[rr, "detpar"] \arrow[dd, "{no,lbeta}"] & & ((\lambda y.s)t)((\lambda y.s)t) \arrow[d, "{no,lbeta}"] \\
& & (\texttt{let } y = t \texttt{ in } s)((\lambda y.s)t) \arrow[d, "{i,lbeta}"] \\
\texttt{let } x = (\texttt{let } y = t \texttt{ in } s) \texttt{ in } (xx) \arrow[r, "detpar"] & t'\,t' & (\texttt{let } y = t \texttt{ in } s)(\texttt{let } y = t \texttt{ in } s) \arrow[l, "detpar"']
\end{tikzcd}$$

Offensichtlich kann die Anwendung einer $(no, lbeta)$-Reduktion vor Anwendung einer *detpar*-Reduktion wie "geshart" wirken, d.h. auf mehreren Subausdrücken gleichzeitig wirken. Ferner kann eine *detpar*-Reduktion weder zum Verschwinden eines $(no, lbeta)$-Redex führen, noch den Typ des no-Redex von *lbeta* fortbewegen, was wir in folgendem Lemma festhalten:

Lemma 4.3.4. *s, t, t' seien drei Ausdrücke, mit $t \xrightarrow{detpar} s$ und $t \xrightarrow{no,lbeta} t'$. Dann existiert ein Ausdruck s' mit $s \xrightarrow{no,lbeta} s'$, d.h. ein $(no, lbeta)$-Redex wird durch detpar bewahrt. Dabei gilt zusätzlich: Ist der $(no, lbeta)$-Redex in t ein deterministischer Subausdruck, so ist auch der $(no, lbeta)$-Redex in s deterministisch.*

Beweis. Wir wenden die *detpar*-Reduktion zunächst auf einen einzelnen *lbeta*-Redex an:

$$(\lambda x.s)t \xrightarrow{detpar} (\lambda x.s')t', \text{ wobei } s \xrightarrow{detpar} s' \text{ und } t \xrightarrow{detpar} t'$$

Gemäß Lemma 2.3.1 hat ein $(no, lbeta)$-Redex das Aussehen $W[(\lambda x.s)t]$. Wir wenden *detpar* auf einen $(no, lbeta)$-Redex an:

$$W[(\lambda x.s)t] \xrightarrow{detpar} W'[((\lambda x.s')t')[\overrightarrow{x/t_x}]]]$$

Wir lösen die Folge von Substitutionen auf:

$$((\lambda x.s')t')[\overrightarrow{x/t_x}] \equiv ((\lambda x.s')[\overrightarrow{x/t_x}])(t'[\overrightarrow{x/t_x}]) \equiv (\lambda x.s'')t''$$

Final benutzen wir Lemma wiederum 2.3.1, welches besagt, daß der Ausdruck $W'[(\lambda x.s'')t'']$ einen $(no, lbeta)$-Redex besitzt.
Die Bewahrung des Determinismus ist offensichtlich, da ein nichtdeterministischer Subausdruck durch die Anwendung einer *detpar*-Reduktion niemals deterministisch werden kann. □

Im weiteren werden wir nun das Reduktionsverhalten von *detpar* bei einem deterministischen $(no, lbeta)$-Redex näher untersuchen. In diesem Zusammenhang wird es erforderlich bei zwei Ausdrücken t, s mit $t \xrightarrow{detpar} s$ eine Möglichkeit bereitszustellen, die parallele Wirkungsweise einer $(no, lbeta)$-Reduktion in t bei s (siehe obiges Beispiel) verfolgen zu können. Wir benutzen dazu die übliche Methodik des Markierens von Redizes, in diesem Fall der *lbeta*-Redizes. Dies macht die Definition einer erweiterten Form der Λ_{let}-Ausdrücke erforderlich, bei der einzelne Ausdrücke eine Redex-Markierung tragen können:

Definition 4.3.2. *Die Menge der markierten Ausdrücke* Λ^+_{let} *wird wie folgt definiert:*

1. $x \in \Lambda^+_{let}$ *für eine Variable* x
2. $s, t \in \Lambda^+_{let} \;\Rightarrow\; \lambda x.s \in \Lambda^+_{let},\ (st) \in \Lambda^+_{let},\ (\mathtt{let}\ x = t\ \mathtt{in}\ s) \in \Lambda^+_{let},$
   $^{\underline{i}}((\lambda x.s)t) \in \Lambda^+_{let}$, *wobei* $i \in \mathbb{N}$

Wir definieren nun eine Markierungsfunktion $\sigma : \Lambda_{let} \mapsto \Lambda^+_{let}$, durch deren Anwendung auf einen gegebenen Ausdruck t jeder *lbeta*-Redex in t eine eindeutige Markierung erhält.

Definition 4.3.3. *Wir definieren eine Funktion* $mark : (\mathbb{N} \times \Lambda_{let}) \mapsto (\mathbb{N} \times \Lambda^+_{let})$ *wie folgt:*

$$\begin{array}{rcll}
mark\,(n,\, x) & := & (n,\, x) & \textit{wobei } x \textit{ eine Variable} \\
mark\,(n,\, \lambda x.s) & := & (n',\, \lambda x.s') & \textit{wobei } (n',\, s') = mark\,(n,\, s) \\
mark\,(n,\, st) & := & (n',\, s't') & \textit{wobei } (n',\, s') = mark\,(n,\, s), \\
 & & & (n'',\, t') = mark\,(n',\, t) \textit{ und } s \textit{ keine Abstraktion} \\
mark\,(n,\, (\lambda x.s)t) & := & (n'',\, ^{\underline{n}}((\lambda x.s')t')\,) & \textit{wobei } (n',\, s') = mark\,(n+1,\, s) \\
 & & & \textit{und } (n'',\, t') = mark\,(n',\, t) \\
mark\,(n,\, \mathtt{let}\ x = t_x\ \mathtt{in}\ s) & := & (n'',\, \mathtt{let}\ x = t'_x\ \mathtt{in}\ s') & \textit{wobei } (n',\, t'_x) = mark\,(n,\, t_x) \\
 & & & \textit{und } (n'',\, s') = mark\,(n',\, s)
\end{array}$$

Zusätzlich vereinbaren wir: $\sigma(t) := t'$ *falls* $(n', t') = mark\,(0,\, t)$.

Beispiel 4.3.2. Somit erhalten wir bei der Anwendung von σ auf den Beispielausdruck

$$t \equiv \texttt{let}\ x = (\lambda y.y)\ \texttt{in}\ ((\lambda z.z)x)((\lambda u.u)x)$$

als Ergebnis

$$\sigma(t) \equiv \texttt{let}\ x = (\lambda y.y)\ \texttt{in}\ {}^{\underline{0}}((\lambda z.z)x)^{\underline{1}}((\lambda u.u)x)$$

Nun definieren wir eine Variante der *detpar*-Reduktion, die das vom no-Markierungsalgorithmus her bekannte E-Label sowie die zuvor eingeführten Markierungen an *lbeta*-Redizes 'mitüberträgt'.

Definition 4.3.4. *Gegeben ein Ausdruck t. Wir definieren eine Variation der detpar-Reduktion, bezeichnet mit* $\xrightarrow{detpar^+}$, *wie folgt:*

- $x \xrightarrow{detpar^+} x$ *wobei x eine Variable oder die Konstante* `choice`
- *Wenn* $\lambda x.s$ *ein Subausdruck von t und* $s \xrightarrow{detpar^+} s'$,
  dann $\lambda x.s \xrightarrow{detpar^+} \lambda x.s'$
- *Wenn* $(s_1 s_2)$ *ein Subausdruck von t,* $s_1 \xrightarrow{detpar^+} s_1'$, $s_2 \xrightarrow{detpar^+} s_2'$,
  dann $s_1\, s_2 \xrightarrow{detpar^+} s_1'\, s_2'$
- *Wenn* $({}^{\underline{i}}((\lambda x.s_1)s_2)^E)$ *ein Subausdruck von t,* $s_1 \xrightarrow{detpar^+} s_1'$, $s_2 \xrightarrow{detpar^+} s_2'$,
  dann ${}^{\underline{i}}((\lambda x.s_1)s_2)^E \xrightarrow{detpar^+} {}^{\underline{i}}((\lambda x.s_1)s_2)^E$ *wobei optional entweder die* ${}^{\underline{i}}$*-Markierung oder das E-Label oder beides wegfallen kann.*
- *Wenn* $(\texttt{let}\ x = t_x\ \texttt{in}\ s)$ *ein Subausdruck von t,* $t_x \xrightarrow{detpar^+} t_x'$, $s \xrightarrow{detpar^+} s'$,
  dann $\texttt{let}\ x = t_x\ \texttt{in}\ s \xrightarrow{detpar^+} \begin{cases} s'[t_x'/x] & \text{falls } x \in V_{det}(t) \\ \texttt{let}\ x = t_x'\ \texttt{in}\ s' & \text{sonst} \end{cases}$

Korollar 4.3.4. *Die detpar$^+$-Reduktion entspricht bei Abstraktion von allen Markierungen der detpar-Reduktion.*

Offensichtlich können E-Label sowie ${}^{\underline{i}}$-Markierungen durch den potentiellen Substitutionsprozess bei einem L_R-Kontext vervielfacht werden, was beabsichtigt ist. Das nun folgende Beispiel zeigt die Wirkungsweise der drei zuletzt getroffenen Definitionen auf.

Beispiel 4.3.3. Gegeben sei der Λ^+_{let}-Ausdruck $t \equiv \texttt{let}\ x = (\lambda x.t_x)s\ \texttt{in}\ xx$ mit t_x und s als deterministischen Subausdrücken. Daraus ergibt sich:

$\sigma(t) \equiv \texttt{let}\ x = {}^{\underline{0}}((\lambda x.t_x)s)\ \texttt{in}\ xx$

$E(\sigma(t)) \equiv \texttt{let}\ x = {}^{\underline{0}}((\lambda x.t_x)s)^E\ \texttt{in}\ xx$, wobei dabei mit $E(t)$ der Ausdruck bezeichnet wird, der nach Anwendung des no-Markierungsalgorithmus erhalten wird.

$E(\sigma(t)) \xrightarrow{detpar^+} {}^{\underline{0}}((\lambda x.t_x)s)^E\,{}^{\underline{0}}((\lambda x.t_x)s)^E$

Die zuletzt getroffenen Definitionen versetzen uns in die Lage das nun folgende Lemma zu beweisen, das das Verhalten der *detpar*-Reduktion bei zwei Ausdrücken mit einem $(no, lbeta)$-Redex und einem gemeinsamen Ausdruck in *detpar*-Normalform beschreibt.

Lemma 4.3.5. *Gegeben 3 Ausdrücke t_1, s, t_2 mit $t_1 \xrightarrow{detpar} s$ und $t_2 \xrightarrow{detpar} s$. Zusätzlich sei der no-Redex der Ausdrücke t_1 sowie t_2 vom Typ lbeta und deterministisch. Dann existieren Ausdrücke t_1', s', t_2', so daß die Reduktionszusammenhänge des folgenden Diagramms erfüllt werden:*

$$\begin{tikzcd}
t_1 \arrow[rr, "detpar"'] \arrow[d, "{no, lbeta, det}"'] & & s & & t_2 \arrow[ll, "detpar"] \arrow[d, "{no, lbeta, det}"] \\
\circ \arrow[d, dashed, "{no, lbeta, *}"'] & & & & \circ \arrow[d, dashed, "{no, lbeta, *}"] \\
\circ \arrow[r, dashed, "{i, lbeta, *}"'] & t_1' \arrow[r, dashed, "detpar"'] & s' & t_2' \arrow[l, dashed, "detpar"] & \circ \arrow[l, dashed, "{i, lbeta, *}"]
\end{tikzcd}$$

Beweis. Wir benutzen den nun folgenden Markierungsalgorithmus, um eine Menge von *lbeta*-Redizes in t_1 sowie t_2 zu markieren, so daß durch deren Reduktion die gesuchten Ausdrücke t_1' bzw. t_2' erreicht werden. Die Markierung erfolgt dabei mittels des aus dem no-Markierungsalgorithmus bekannten E-Label.

i) Wir bilden die beiden Ausdrücke $t_1^e \equiv E(\sigma(t_1))$ und $t_2^e \equiv E(\sigma(t_2))$. D.h. wir kennzeichnen in t_1 und t_2 alle *lbeta*-Redizes eindeutig und versehen die no-Redizes mit einem E-Label.

iii) Wir wenden auf die beiden Ausdrücke t_1^e und t_2^e die *detpar*-Reduktion an und erhalten:

$$t_1^e \xrightarrow{detpar^+} s_1 \text{ und } t_2^e \xrightarrow{detpar^+} s_2$$

(Bei Abstrahierung von allen Markierungen entsprechen gemäß Voraussetzung die Ausdrücke s_1 und s_2 einander.)

$$\begin{tikzcd}
t_1 \arrow[rr, "detpar"'] \arrow[d, "{no, lbeta, det}"'] & & s \arrow[d, dashed, "{no, lbeta, det, +}"] & & t_2 \arrow[ll, "detpar"'] \arrow[d, "{no, lbeta, det}"] \\
\circ \arrow[dr, dashed, "{lbeta, det, *}"'] & & \circ \arrow[d, dashed, "detpar"] & & \circ \arrow[dl, dashed, "{lbeta, det, *}"] \\
& t_1' \arrow[r, dashed, "detpar"'] & s' & t_2' \arrow[l, dashed, "detpar"] &
\end{tikzcd}$$

Abbildung 4.4:

iv) Wir bestimmen, ob zwei Kontexte C_1 und C_2 existieren, die abstrahiert von allen Markierungen identisch sind, so daß

$$s_1 \equiv C_1[{}^{\underline{i}}r^E] \text{ und } s_2 \equiv C_2[{}^{\underline{i}}r]$$

gilt, d.h. wir prüfen ob ein *lbeta*-Redex in s_1 existiert der ein E-Label trägt, jedoch der *lbeta*-Redex an derselben Stelle in s_2 kein E-Label.
Existieren solche zwei Kontexte, heften wir an den *lbeta*-Redex in t_2^e, der die ${}^{\underline{i}}$-Markierung trägt ein E-Label und gehen zu Schritt *iii)*. Sonst fahren wir bei Schritt *v)* fort.

v) Wir bestimmen nun genau umgekehrt, ob zwei Kontexte C_1 und C_2 existieren, die abstrahiert von allen Markierungen identisch sind, so daß

$$s_1 \equiv C_1[{}^{\underline{i}}r] \text{ und } s_2 \equiv C_2[{}^{\underline{i}}r^E]$$

gilt, d.h. wir prüfen ob ein *lbeta*-Redex in s_2 existiert der ein E-Label trägt, jedoch der entsprechende *lbeta*-Redex in s_1 kein E-Label.
Existieren solche Kontexte, heften wir an den *lbeta*-Redex in t_1^e, der die ${}^{\underline{i}}$-Markierung trägt ein E-Label und gehen zu Schritt *iii)*.
Sonst stoppen wir, jetzt gilt $s_1 \equiv s_2$.

Der obige Algorithmus terminiert immer, da in jedem Schritt mindestens ein *lbeta*-Redex markiert wird und die insgesamte Anzahl an *lbeta*-Redizes in t_1 und t_2 endlich ist.
Die folgende Überlegung zeigt unter Zuhilfenahme von Abbildung 4.4 die Hypothese:

Wir betrachten zunächst das linke "Halbdiagramm" aus Abbildung 4.4, d.h. wir beschränken unsere Sicht auf t_1 und s. An genau all den Stellen, wo durch Anwendung des Markierungsalgorithmus in t_1 und s ein *lbeta*-Redex ein E-Label erhalten hat, tritt nach der Reduzierung der markierten Redizes in t_1' bzw. s'' ein Let-Ausdruck auf, bei dem ein deterministischer Subausdruck gebunden wird.

Kombinieren wie dies mit der Kenntnis, daß s die *detpar*-Normalform von t_1 ist und die Markierungen durch die $detpar^+$-Reduktion im Verlauf der Anwendung des Markierungsalgorithmus mitübertragen wurden, so ist es offensichtlich, daß t_1' und s'' wiederum dieselbe *detpar*-Normalform haben müssen. Dieselbe Argumentation läßt sich symmetrisch von t_2 aus anwenden, wobei die markierten *lbeta*-Redizes in s gleich bleiben. Daraus ergibt sich direkt die Gültigkeit des Diagramms aus Abbildung 4.4. Da die *lbeta*-Redizes der beiden Reduktionen $t_1 \xrightarrow{lbeta,det,+} t_1'$ und $t_2 \xrightarrow{lbeta,det,+} t_2'$ jeweils parallel vorliegen, d.h. einander nicht überschneiden, können sie in beliebiger Reihenfolge angeordnet werden. Indem zuerst alle no-Reduktionen und dann alle internen Reduktionen durchgeführt werden, ist die Hypothese gezeigt. □

Der in obigem Lemma enthaltene Algorithmus soll zusätzlich an einem Beispiel erläutert werden:

Beispiel 4.3.4. Wir definieren zunächst zwei fixe Ausdrücke die einen *lbeta*-Redex bilden:

$$r_1 \equiv (\lambda x.x)(\lambda y.y)$$
$$r_2 \equiv (\lambda x.x)(\texttt{let}\ z = \lambda y.y\ \texttt{in}\ z)$$

Auf Basis von r_1 und r_2 definieren wir die beiden Ausdrücke:

$$t_1 \equiv \texttt{let}\ x = r_2\ \texttt{in}\ (\texttt{let}\ y = xx\ \texttt{in}\ yyr_1)\ \text{und}$$
$$t_2 \equiv \texttt{let}\ x = r_1\ \texttt{in}\ (\texttt{let}\ y = r_2x\ \texttt{in}\ yyx)$$

Es gilt $r_2 \xrightarrow{detpar} r_1$, da $\lambda y.y$ ein geschlossener deterministischer Subausdruck ist. Ferner gilt $t_1 \xrightarrow{detpar} s$ und $t_2 \xrightarrow{detpar} s$ mit $s \equiv r_1r_1r_1r_1r_1$. Wir versehen nun die *lbeta*-Redizes bei t_1 und t_2 mit einer eindeutigen Markierung und erhalten:

$$\texttt{let}\ x = {}^{\underline{0}}r_2\ \texttt{in}\ (\texttt{let}\ y = xx\ \texttt{in}\ yy\,{}^{\underline{1}}r_1)\ \text{und}$$
$$\texttt{let}\ x = {}^{\underline{0}}r_1\ \texttt{in}\ (\texttt{let}\ y = {}^{\underline{1}}r_2x\ \texttt{in}\ yyx)$$

Nun markieren wir in beiden Ausdrücken den no-Redex:

$$t_1^e \equiv \texttt{let}\ x = {}^{\underline{0}}r_2^E\ \texttt{in}\ (\texttt{let}\ y = xx\ \texttt{in}\ yy\,{}^{\underline{1}}r_1)$$
$$t_2^e \equiv \texttt{let}\ x = {}^{\underline{0}}r_1\ \texttt{in}\ (\texttt{let}\ y = {}^{\underline{1}}r_2^Ex\ \texttt{in}\ yyx)$$

Wir wenden auf beide Ausdrücke die *detpar*-Reduktion an:

$$s_1 \equiv {}^{\underline{0}}r_1^E\ {}^{\underline{0}}r_1^E\ {}^{\underline{0}}r_1^E\ {}^{\underline{0}}r_1^E\ {}^{\underline{1}}r_1$$
$$s_2 \equiv {}^{\underline{1}}r_1^E\ {}^{\underline{0}}r_1\ {}^{\underline{1}}r_1^E\ {}^{\underline{0}}r_1\ {}^{\underline{0}}r_1$$

Offensichtlich besitzt ein *lbeta*-Redex mit der Kennung $\underline{0}$ in s_2 kein E-Label, während sein Partner in s_1 ein E-Label besitzt. Wir heften gemäß Schritt *iii)* ein E-Label an den *lbeta*-Redex mit der Kennung $\underline{0}$ in t_2^e und erhalten:

$$t_2^e \equiv \texttt{let}\ x = {}^{\underline{0}}r_1^E\ \texttt{in}\ (\texttt{let}\ y = {}^{\underline{1}}r_2^Ex\ \texttt{in}\ yyx)$$

Wir wenden wiederum auf beide Ausdrücke die *detpar*-Reduktion an:

$$\begin{aligned} s_1 &\equiv {}^{\underline{0}}r_1^E\ {}^{\underline{0}}r_1^E\ {}^{\underline{0}}r_1^E\ {}^{\underline{0}}r_1^E\ {}^{\underline{1}}r_1 \\ s_2 &\equiv {}^{\underline{1}}r_1^E\ {}^{\underline{0}}r_1^E\ {}^{\underline{1}}r_1^E\ {}^{\underline{0}}r_1^E\ {}^{\underline{0}}r_1^E \end{aligned}$$

Nun existiert ein *lbeta*-Redex in s_1 der kein E-Label trägt, während der zugehörige *lbeta*-Redex in s_2 ein E-Label trägt. Wir helfen entsprechend Schritt *v)* ein E-Label an den *lbeta*-Redex mit der Markierung ${}^{\underline{1}}$ in t_1^e und erhalten:

$$t_1^e \equiv \texttt{let}\ x = {}^{\underline{0}}r_2^E\ \texttt{in}\ (\texttt{let}\ y = xx\ \texttt{in}\ yy{}^{\underline{1}}r_1^E)$$

Wir wenden wiederum auf beide Ausdrücke die *detpar*-Reduktion an:

$$\begin{aligned} s_1 &\equiv {}^{\underline{0}}r_1^E\ {}^{\underline{0}}r_1^E\ {}^{\underline{0}}r_1^E\ {}^{\underline{0}}r_1^E\ {}^{\underline{1}}r_1^E \\ s_2 &\equiv {}^{\underline{1}}r_1^E\ {}^{\underline{0}}r_1^E\ {}^{\underline{1}}r_1^E\ {}^{\underline{0}}r_1^E\ {}^{\underline{0}}r_1^E \end{aligned}$$

Damit ist die gesuchte Äquivalenz bezüglich der E-Label gefunden. In beiden Ausdrücken t_1^e und t_2^e tragen alle *lbeta*-Redizes ein E-Label.

Wir verifizieren final, ob die entwickelte $(lbeta, par)$-Reduktion die gewünschten Eigenschaften besitzt und reduzieren

$$\begin{aligned} t_1^e &\xrightarrow{lbeta,par} t_1''' \equiv \texttt{let}\ x = l_2\ \texttt{in}\ (\texttt{let}\ y = xx\ \texttt{in}\ yyl_1) \\ t_2^e &\xrightarrow{lbeta,par} t_2''' \equiv \texttt{let}\ x = l_1\ \texttt{in}\ (\texttt{let}\ y = l_2x\ \texttt{in}\ yyx) \end{aligned}$$

wobei $l_1 \equiv \texttt{let}\ x = \lambda y.y\ \texttt{in}\ x$ und $l_2 \equiv \texttt{let}\ x = (\texttt{let}\ z = \lambda y.y\ \texttt{in}\ z)\ \texttt{in}\ x$.

Wir wenden auf t_1''' und t_2''' die *detpar*-Reduktion an und erhalten jeweils den gemeinsamen Nachfahren:

$$s_d \equiv ((\lambda y.y)(\lambda y.y))((\lambda y.y)(\lambda y.y))(\lambda y.y)$$

Wir verallgemeinern die Aussage des zuletzt gezeigten Lemmas nun derart, daß wir nicht mehr verlangen, daß der no-Redex des Ausdrucks t_2 deterministisch und vom Typ *lbeta* ist. Zusätzlich lassen wir nun die Fälle zu, daß der no-Redex deterministisch und vom Typ *llet*, *lapp* oder *cp* ist. Wir erhalten dann folgendes Lemma:

Lemma 4.3.6. *Gegeben 4 Ausdrücke* t_1, t_2, t_1', s *mit* $t_1 \xrightarrow{detpar} s$, $t_2 \xrightarrow{detpar} s$ *und* t_1 *besitze einen deterministischen* $(no, lbeta)$*-Redex, dessen Reduktion zu* t_1' *führe. Dann existieren Ausdrücke* s', t_2', *so daß die Reduktionszusammenhänge des folgenden Diagramms erfüllt werden:*

$$\begin{tikzcd}
t_1 \arrow[rr, "detpar"] \arrow[d, "{no, lbeta, det}"] & & s & & t_2 \arrow[ll, "detpar"'] \arrow[d, dashed, "{no, \{lapp, llet, cp\}, weak, *}"] \\
t_1' \arrow[ddd, dashed, "{no, lbeta, *}"] & & & & t_2' \arrow[llu, dashed, "detpar"] \arrow[d, dashed, "{no, lbeta, det}"] \\
& & & & \circ \arrow[dd, dashed, "{no, lbeta, *}"] \\
& & & & \\
\circ \arrow[r, dashed, "{i, lbeta, *}"'] & \circ \arrow[r, dashed, "detpar"'] & s' & \circ \arrow[l, dashed, "detpar"] & \circ \arrow[l, dashed, "{i, lbeta, *}"]
\end{tikzcd}$$

Beweis. Wir führen ausgehend von t_2 solange eine no-Reduktion, wie der zu reduzierende no-Redex deterministisch oder schwach nichtdeterministisch und vom Typ *lapp*, *llet* oder *cp* ist. Wegen der Reduzierung des Maßes ξ (das Maß ξ wird in Abschnitt 3.4.3 im Rahmen der Betrachtung der *lcv*-Reduktion definiert) durch jede *llet*, *lapp*, *cp*-Reduktion ist letztere no-Reduktion stets endlich beschränkt. Sei t_2' der Ausdruck, der so erreicht werde.
Gemäß Lemma 4.3.4 hat s einen deterministischen no-Redex vom Typ *lbeta*. Folgende Widersprüche zeigen, daß der no-Redex von t_2' deterministisch und vom Typ *lbeta* sein muß:

- wenn t_2' einen stark oder beschränkt nichtdeterministischen no-Redex hätte, entstände ein Widerspruch zu Lemma 4.3.1 in Verbindung mit der Eindeutigkeit des no-Redex in s.
- wenn t_2' einen deterministischen oder schwach nichtdeterministischen no-Redex hätte, der nicht vom Typ *lbeta* ist, dann entstände ein Widerspruch dazu wie weit die no-Reduktion ausgehend von t_2 zu entwickeln ist.
- wenn t_2' in WHNF wäre, entstände ein Widerspruch zu Lemma 4.3.2.

Somit muß der no-Redex von t_2' deterministisch und vom Typ *lbeta* sein. Die Hypothese folgt dann direkt aus Lemma 4.3.5. □

Denselben Zusammenhang, den wir im letzten Lemma für einen $(no, lbeta, det)$-Redex gezeigt haben, zeigen wir nun für einen stark nichtdeterministischen no-Redex.

Lemma 4.3.7. *Gegeben 4 Ausdrücke* t_1, t_2, t_1', s *mit* $t_1 \xrightarrow{detpar} s$, $t_2 \xrightarrow{detpar} s$ *und* t_1 *besitze einen stark nichtdeterministischen no-Redex, dessen Reduktion zu* t_1'

führe. Dann existieren Ausdrücke s', t_2', t_2'', so daß die Reduktionszusammenhänge des folgenden Diagramms erfüllt werden:

$$\begin{tikzcd}
t_1 \arrow[r, "detpar"] \arrow[dd, "{no, a, strong}"] & s & t_2 \arrow[l, "detpar"'] \arrow[d, dashed, "{no, \{lapp, llet, cp\}, weak, *}"] \\
 & a \in \{llet, lapp, cp, lbeta, nd\} & t_2'' \arrow[ul, dashed, "detpar"] \arrow[d, dashed, "{no, a, strong}"] \\
t_1' \arrow[r, dashed, "detpar"'] & s' & t_2' \arrow[l, dashed, "detpar"]
\end{tikzcd}$$

Beweis. Gemäß Lemma 4.3.1 hat s einen stark nichtdeterministischen no-Redex vom selben Typ wie t_1. Wir führen ausgehend von t_2 solange eine no-Reduktion, wie der zu reduzierende no-Redex deterministisch oder schwach nichtdeterministisch und vom Typ *lapp*, *llet* oder *cp* ist. Wegen der Reduzierung des Maßes ξ (das Maß ξ wird in Abschnitt 3.4.3 im Rahmen der Betrachtung der *lcv*-Reduktion definiert) durch jede *llet*, *lapp*, *cp*-Reduktion ist letztere no-Reduktion stets endlich beschränkt. t_2'' sei der so erreichte Ausdruck. Aus Korollar 4.3.2 folgt, daß t_2'' durch *detpar* auf denselben Ausdruck abgebildet wird wie t_2. Folgende Widersprüche zeigen, daß der no-Redex von t_2'' stark nichtdeterministisch und vom selben Typ wie der no-Redex von t_1 sein muß:

- wenn der no-Redex von t_2'' vom Typ *lbeta* und deterministisch wäre, entstände ein Widerspruch zu Lemma 4.3.4 in Verbindung mit der Eindeutigkeit des no-Redex in s.
- wenn t_2'' einen deterministischen oder schwach nichtdeterministischen no-Redex hätte, der nicht vom Typ *lbeta* ist, dann entstände ein Widerspruch dazu wie weit die no-Reduktion ausgehend von t_2 zu entwickeln ist.
- wenn t_2'' einen beschränkt nichtdeterministischen no-Redex hätte, entstände ein Widerspruch zu Lemma 4.3.1 in Verbindung mit der Eindeutigkeit des no-Redex in s.
- wenn t_2'' in WHNF wäre, entstände ein Widerspruch zu Lemma 4.3.2.

Wir wählen als t_2' den mittels einer $(no, strong)$-Reduktion von t_2'' aus erreichbaren Ausdruck. Die Hypothese ergibt sich dann direkt aus den Gabeldiagrammen von Lemma 4.3.1 zusammen mit der Eindeutigkeit des no-Redex kombiniert gemäß des Reduktionsdiagramms aus Abbildung 4.5. □

Was wir im letzten Lemma für stark-nichtdeterministische no-Redizes gezeigt haben, zeigen wir in dem nun folgenden Lemma für beschränkt nichtdeterministische no-Redizes.

$$\begin{tikzcd}
t_1 \arrow[rr, "detpar"] \arrow[d, "{no, a, strong}"'] & & s \arrow[d, dashed, "{no, a, strong}"] \arrow[ll, phantom, "\text{Lemma 4.3.1}" below, shift right=4ex] \arrow[rr, phantom, "\text{Lemma 4.3.1}" below, shift right=4ex] & & t_2'' \arrow[ll, "detpar"'] \arrow[d, "{no, a, strong}"] \\
t_1' \arrow[rr, dashed, "detpar"'] & & s' & & t_2' \arrow[ll, dashed, "detpar"]
\end{tikzcd}$$

$a \in \{llet, lapp, cp, lbeta, nd\}$

Abbildung 4.5:

$$\begin{tikzcd}
t_1 \arrow[rr, "detpar"] \arrow[d, "{no, a, middle}"'] & & s \arrow[d, dashed, "{no, a, middle}"] \arrow[ll, phantom, "\text{Lemma 4.3.1}" below, shift right=4ex] \arrow[rr, phantom, "\text{Lemma 4.3.1}" below, shift right=4ex] & & t_2'' \arrow[ll, "detpar"'] \arrow[d, "{no, a, middle}"] \\
t_1' \arrow[r, dashed, "detpar"'] & s_1 \arrow[dr, dash] & s'' \arrow[l, dashed, "detpar"'] \arrow[r, dashed, "detpar"'] & s_2 \arrow[dl, dash] & t_2' \arrow[l, dashed, "detpar"] \\
& & \equiv & &
\end{tikzcd}$$

$a \in \{llet, lbeta, nd\}$

Abbildung 4.6:

Lemma 4.3.8. *Gegeben 4 Ausdrücke* t_1, t_2, t_1', s *mit* $t_1 \xrightarrow{detpar} s$, $t_2 \xrightarrow{detpar} s$ *und* t_1 *besitze einen beschränkt nichtdeterministischen no-Redex, dessen Reduktion zu* t_1' *führe. Dann existieren Ausdrücke* s', t_2', t_2'', *so daß die Reduktionszusammenhänge des folgenden Diagramms erfüllt werden:*

$$\begin{tikzcd}
t_1 \arrow[r, "detpar"] \arrow[dd, "{no, a, middle}"] & s & t_2 \arrow[l, "detpar"'] \arrow[d, dashed, "{no, \{lapp, llet, cp\}, weak, *}"] \\
& a \in \{llet, lbeta, nd\} & t_2'' \arrow[ul, dashed, "detpar"] \arrow[d, dashed, "{no, a, middle}"] \\
t_1' \arrow[r, dashed, "detpar"'] & s' & t_2' \arrow[l, dashed, "detpar"]
\end{tikzcd}$$

Beweis. Der Beweis erfolgt zunächst nach demselben Schema wie für einen stark nichtdeterministischen Redex beim Lemma zuvor:
Es wird zunächst von t_2 aus solange reduziert, wie der no-Redex deterministisch bzw. schwach nichtdeterministisch und vom Typ *lapp*, *llet* oder *cp* ist. Für den so erreichten Ausdruck t_2'' wird dann mittels einer Reihe von Widerspruchsüberlegungen gezeigt, daß der no-Redex von t_2'' vom gleichen Typ wie der no-Redex von t_1 sowie beschränkt nichtdeterministisch ist.

Als t'_2 wählen wir den mittels einer $(no, middle)$-Reduktion ausgehend von t''_2 erreichbaren Ausdruck. Die Hypothese ergibt sich dann direkt aus den Gabeldiagrammen von Lemma 4.3.1 zusammen mit der Eindeutigkeit des no-Redex kombiniert gemäß des Reduktionsdiagramms aus Abbildung 4.6. Den gesuchten Ausdruck s' erhalten wir durch Wahl eines der syntaktisch äquivalenten Ausdrücke s_1 und s_2. Die syntaktische Äquivalenz beider Ausdrücke ergibt sich aus der Eindeutigkeit der *detpar*-Normalform. ☐

Wir bilden nun eine Erweiterung der drei zuvor gezeigten Lemmata, indem wir zu dem Fall übergehen, daß die no-Reduktion von beliebiger Länge sein darf.

Lemma 4.3.9. *Gegeben 4 Ausdrücke* t_1, t_2, t'_1, s *mit* $t_1 \xrightarrow{detpar} s$, $t_2 \xrightarrow{detpar} s$, $t_1 \xrightarrow{no,*} t'_1$. *Dann existieren Ausdrücke* s', t''_1, t''_2, *so daß die Reduktionszusammenhänge des folgenden Diagramms erfüllt werden:*

$$\begin{tikzcd}
t_1 \arrow[rr, "detpar"] \arrow[d, "{no,*}"] & & s & & t_2 \arrow[ll, "detpar"'] \arrow[dd, dashed, "{no,*}"] \\
t'_1 \arrow[d, dashed, "{no,lbeta,*}"] & & & & \\
t''_1 \arrow[r, dashed, "{i,lbeta,*}"'] & \circ \arrow[r, dashed, "detpar"'] & s' & \circ \arrow[l, dashed, "detpar"] & t''_2 \arrow[l, dashed, "{i,lbeta,*}"]
\end{tikzcd}$$

Beweis. Wir führen den Beweis durch Induktion über die Länge der no-Reduktion von t_1 zu t'_1.

Induktionsanfang
Die Reduktion von t_1 zu t'_1 umfaßt 0 Reduktionen, d.h. es gilt $t_1 \equiv t'_1$. Indem wir $t''_1 \equiv t_1$ und $t''_2 \equiv t_2$ sowie $s' \equiv s$ wählen, haben wir diesen trivialen Fall gezeigt.

Induktionsschritt
Wir zerlegen die no-Reduktion von t_1 zu t'_1 in $t_1 \xrightarrow{no} t'''_1 \xrightarrow{no,*} t''_1$, d.h t'''_1 wird in einem no-Schritt von t_1 aus erreicht, und führen eine Fallunterscheidung nach der Art des no-Redex von t_1 durch:

Fall 1) Der no-Redex von t_1 ist *stark oder beschränkt nichtdeterministisch.*

Abbildung 4.7 enthält ein Reduktionsdiagramm, daß den Beweis dieses Falls grafisch darstellt. Wir beginnen mit Lemma 4.3.7 bzw. Lemma 4.3.8, dies ist abhängig davon, ob t'''_1 einen stark oder beschränkt nichtdeterministischen no-Redex besitzt, und bilden auf diesem Wege die beiden *detpar*-Reduktionen "nach unten" ab. Da die no-Reduktion von t'''_1 zu t'_1 um eine Reduktion gegenüber der von t_1 ausgehenden verkürzt ist, können wir anschließend die Induktionshypothese benutzen und haben diesen Fall gezeigt.

$$\begin{tikzcd}
t_1 \arrow[rr, "detpar"] \arrow[d, "{no, a}"'] & & s \arrow[d, phantom, "\text{Lemma 4.3.7 falls } a = strong \quad \text{Lemma 4.3.8 falls } a = middle"] & & t_2 \arrow[ll, "detpar"'] \arrow[d, dashed, "{no, *}"] & \\
t_1''' \arrow[rr, Rightarrow, dashed, "detpar"'] \arrow[d, "{no, *}"'] & & \circ & & \circ \arrow[ll, Rightarrow, dashed, "detpar"] \arrow[dd, dashed, "{no, *}"] & a \in \{strong, middle\} \\
t_1' \arrow[d, dashed, "{no, lbeta, *}"'] & & \text{Induktionsvoraussetzung} & & & \\
t_1'' \arrow[r, dashed, "{i, lbeta, *}"'] & \circ \arrow[r, dashed, "detpar"'] & s' & \circ \arrow[l, dashed, "detpar"'] & t_2'' \arrow[l, dashed, "{i, lbeta, *}"'] &
\end{tikzcd}$$

Abbildung 4.7:

$$\begin{tikzcd}
t_1 \arrow[rr, "detpar"] \arrow[d, "{no, \{lapp, llet, cp\}, weak}"'] & & s & & t_2 \arrow[ddd, dashed, "{no, *}"] \\
t_1''' \arrow[urr, Rightarrow, dashed, "\text{Lemma 4.3.1}", "detpar"'] \arrow[d, "{no, *}"'] & & & & \\
t_1' \arrow[d, dashed, "{no, lbeta, *}"'] & & \text{Induktionsvoraussetzung} & & \\
t_1'' \arrow[r, dashed, "{i, lbeta, *}"'] & \circ \arrow[r, dashed, "detpar"'] & s' & \circ \arrow[l, dashed, "detpar"'] & t_2'' \arrow[l, dashed, "{i, lbeta, *}"']
\end{tikzcd}$$

Abbildung 4.8:

Fall 2) Der no-Redex von t_1 ist *schwach nichtdeterministisch oder deterministisch.*
Wir differenzieren in einer zweiten Ebene nach dem Typ des no-Redex:

Fall 2.1) Der no-Redex von t_1 ist vom Typ *llet*, *lapp*, *cp*.
Einen Beweis dieses Falls auf Basis eines Reduktionsdiagramms beinhaltet Abbildung 4.8. Aus Lemma 4.3.1 folgt $t_1''' \xrightarrow{detpar} s$. Die no-Reduktion ausgehend von t_1''' zu t_1' ist um eine Reduktion gegenüber der von t_1 ausgehenden verkürzt. Somit können wir die Induktionsvoraussetzung anwenden und haben diesen Fall gezeigt.

Fall 2.2) Der no-Redex von t_1 ist vom Typ *lbeta*.
Den Beweis dieses Falls beinhaltet das Reduktionsdiagramm aus Abbildung 4.9. Die in Abbildung 4.9 enthaltenen Zahlen zeigen an, in welcher Reihenfolge die einzelnen Lemmata anzuwenden sind, damit der Gesamtzusammenhang bewiesen wird. Die Beweisablauf soll kurz erläutert werden:

$$\begin{tikzcd}
t_1 \arrow[rrr, "detpar"] \arrow[d, "{no, lbeta, det}"] & & & s & & & t_2 \arrow[lll, "detpar"'] \arrow[d, dashed, "{no, \{lapp, llet, cp\}, weak, *}"'] \\
t_1''' \arrow[d, dashed, "{no, lbeta, *}"] & & & \text{① Lemma 4.3.6} & & & \circ \arrow[d, dashed, "{no, lbeta, +}"'] \\
\circ \arrow[r, dashed, "{i, lbeta, *}"'] \arrow[d, "{no, m}"'] & t_{1.a} \arrow[rr, "detpar"'] \arrow[dd, dashed, "{no, n \text{ wobei } n \leq m}"] & & s_a & & t_{2.a} \arrow[ll, "detpar"] \arrow[ddd, dashed, "{no, *}"] & \circ \arrow[l, dashed, "{i, lbeta, *}"] \arrow[ddd, dashed, "{no, *}"'] \\
t_1' \; \text{② Lemma 4.2.4} \arrow[d, dashed, "{no, lbeta, *}"'] & & & \text{③ Induktionsvoraussetzung} & \text{⑤ Lemma 4.2.1} & & \\
\circ \arrow[r, dashed, "{i, lbeta, *}"'] \arrow[d, dashed, "{no, lbeta, *}"'] & \circ \arrow[d, dashed, "{no, lbeta, *}"] & \text{④ Lemma 4.2.1} & & & & \\
t_1'' \arrow[r, dashed, "{i, lbeta, *}"'] & \circ \arrow[r, dashed, "{i, lbeta, *}"'] & t_{1.b} \arrow[r, dashed, "detpar"'] & s' & t_{2.b} \arrow[l, dashed, "detpar"] & \circ \arrow[l, dashed, "{i, lbeta, *}"] & t_2'' \arrow[l, dashed, "{i, lbeta, *}"]
\end{tikzcd}$$

Abbildung 4.9:

Wir beginnen mit Lemma 4.3.6 und erhalten auf diesem Wege die Existenz der Ausdrücke $t_{1.a}$, $t_{2.a}$ sowie s_a. Falls die Reduktion von t_1 zu t_1' dabei vollständig erfaßt wird, haben wir die Hypothese für diesen Fall bereits gezeigt. Ansonsten fahren wir mit Lemma 4.2.4 fort, um die zu t_1' führende (no, m)-Reduktion über die $(i, lbeta, *)$-Reduktionen hinweg zu $t_{1.a}$ hin zu übertragen. Die Bedingung $n \leq m$ sichert uns, daß wir im Anschluß die Induktionsvoraussetzung anwenden können, und wir erhalten die Existenz der Ausdrücke $t_{1.b}$, $t_{2.b}$ sowie s'. Um die beiden Ausdrücke t_1'' sowie t_2'' erhalten zu können, ist es erforderlich, die beiden "freibleibenden Ecken" links und rechts zu schließen. Dazu benutzen wir die in Lemma 4.2.1 vorgestellten Vertauschungsregeln für eine $(i, lbeta, *)$-Reduktion. Insgesamt haben wir damit die Hypothese für diesen Fall gezeigt. □

Das obige Lemma enthält eine Existenz-quantifizierte Aussage im Bezug auf die Übertragbarkeit einer no-Reduktion zwischen zwei Ausdrücken mit derselben *detpar*-Normalform. Wir werden obiges Lemma nun als Grundlage benutzen um eine all-quantifizierte Aussage im Bezug auf die *detpar*-Reduktion zu erhalten.

Lemma 4.3.10. *Gegeben drei Ausdrücke* t_1, t_2, s *mit* $t_1 \xrightarrow{detpar} s$ *sowie* $t_2 \xrightarrow{detpar} s$. *Dann gilt* $t_1 \not\Uparrow \Leftrightarrow t_2 \not\Uparrow$.

Beweis. Es reicht die Aussage für eine Richtung zu zeigen. Wegen der Eindeutigkeit der *detpar*-Normalform, sowie der in der Hypothese enthaltenen Symmetrie, ist die Aussage damit auch für die andere Richtung bewiesen.

Wir zeigen die Aussage für die Richtung $\Rightarrow$.
$t_1 \not\Uparrow$ bedeutet, daß alle von t_1 ausgehenden no-Reduktionen endlich sind. Wir beweisen die Hypothese indem wir eine Induktion über die Länge der längsten von t_1 ausgehenden no-Reduktionen führen.

Induktionsanfang
Die längste von t_1 ausgehende no-Reduktion hat die Länge 0, dies ist gleichbedeutend damit, daß sich t_1 in WHNF befindet. Gemäß Lemma 4.3.2 folgt daraus, daß auch der Ausdruck s, die *detpar*-Normalform von t_1, in WHNF ist. Aus s in WHNF wiederum folgt, daß ausgehend von t_2, einem Ausdruck, der die selbe *detpar*-Normalform wie t_1 aufweist, eine no-Reduktion zu einem Ausdruck $t_{2.WHNF}$ in WHNF existiert, wobei $t_2 \xrightarrow{no,\{lllet,lapp,cp\}} t_{2.WHNF}$ gilt. Da die von t_2 ausgehende no-Reduktion keine *nd*-Reduktionen beinhaltet, ist diese eindeutig und die Hypothese für diesen Fall gezeigt.

Induktionsschritt
Wir führen eine Argumentation ähnlich wie beim Beweis von Lemma 4.3.9. Dazu differenzieren wir nach dem Typ des no-Redex des Ausdrucks t_1:

Fall 1) Der no-Redex von t_1 ist *stark oder beschränkt nichtdeterministisch* jedoch **nicht** vom Typ *nd*.
Es entstehen die in Diagramm 1 aus Abbildung 4.10 gezeigten Reduktionszusammenhänge. Die längste von t_1' ausgehende no-Reduktion ist einen no-Schritt kürzer als die Längste von t_1 ausgehende. Dies erlaubt die Anwendung der Induktionsvoraussetzung auf t_1', woraus folgt, daß alle von t_2' ausgehenden no-Reduktionen endlich sind. Da die no-Reduktion von t_2 zu t_2' über keinerlei *nd*-Reduktionen verfügt, sind somit auch alle von t_2 ausgehenden no-Reduktionen endlich.

Fall 2) Der no-Redex von t_1 ist *stark oder beschränkt nichtdeterministisch* und vom Typ *nd*.
Dieser Fall ist eine erweiterte Form des zuvor betrachteten, es entstehen die in Diagramm 2 aus Abbildung 4.10 gezeigten Reduktionszusammenhänge. Auf beide von t_1 ausgehenden (no, nd)-Reduktionen kann dieselbe Argumentation wie bei Fall 1 angewendet werden. Da alle von $t_{2.l}$ wie $t_{2.r}$ ausgehenden no-Reduktionen endlich sind und die no-Reduktion von t_2 zu t_2' über keine *nd*-Reduktionen verfügt sind, folglich auch alle von t_2 ausgehenden no-Reduktionen endlich.

Fall 3) Der no-Redex von t_1 ist *schwach nichtdeterministisch oder deterministisch* und **nicht** vom Typ *lbeta*.
Abbildung zeigt 4.11 zeigt die entstehenden Reduktionszusammenhänge. Die längste von t_1' ausgehende no-Reduktion ist einen no-Schritt kürzer als die Läng-

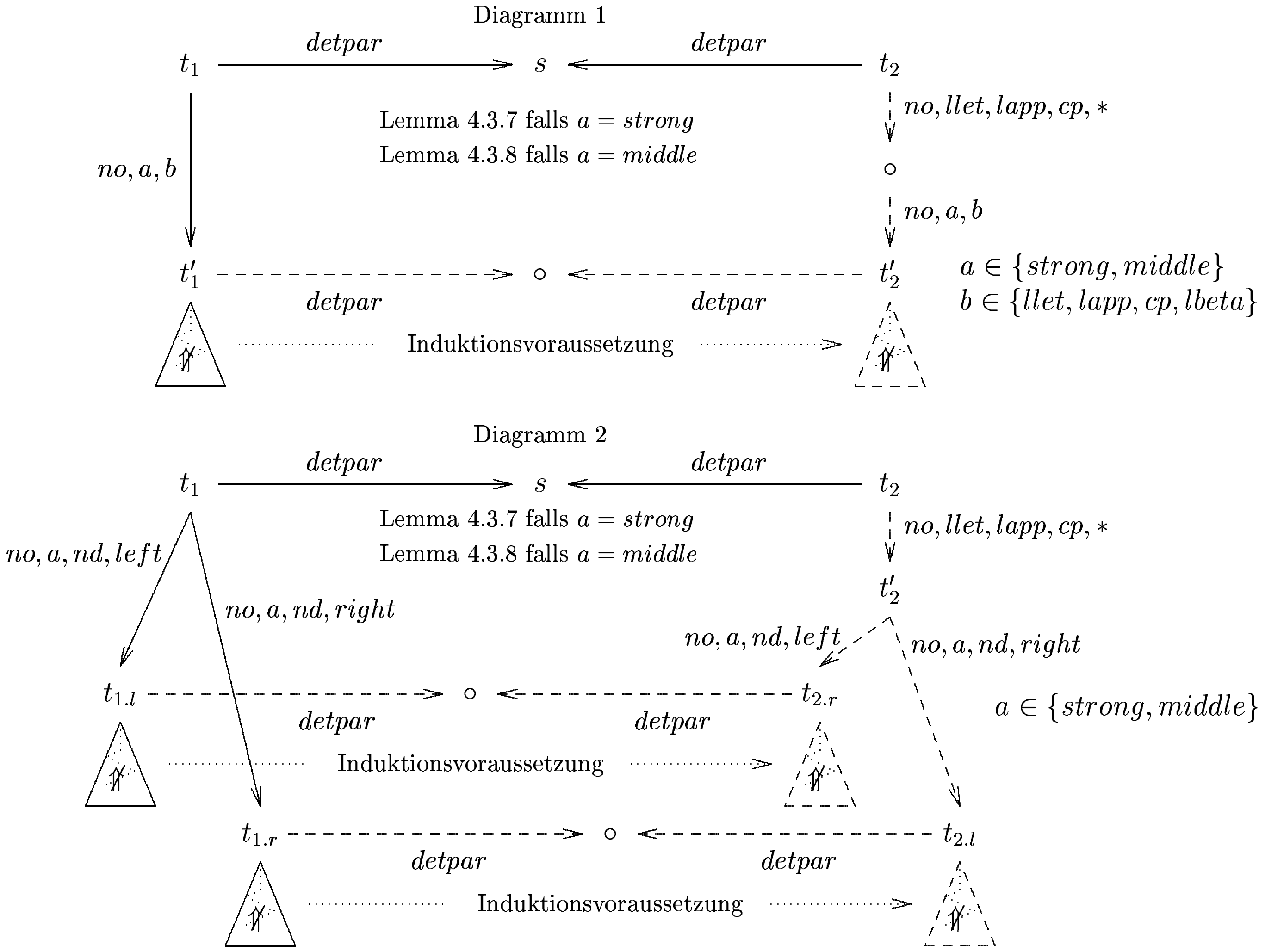


Abbildung 4.10:

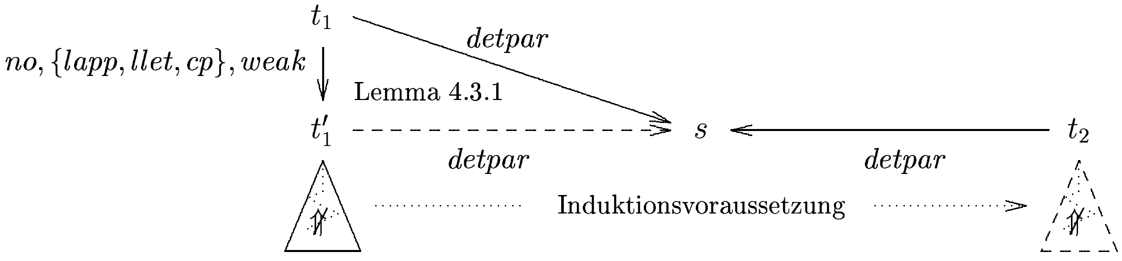


Abbildung 4.11:

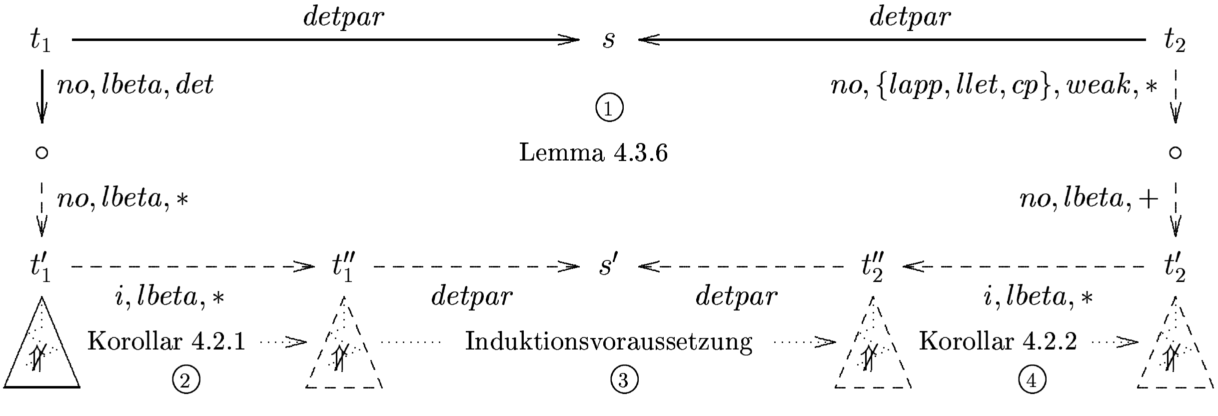


Abbildung 4.12:

ste von t_1 ausgehende. Wegen der Induktionsvoraussetzung sind damit alle von t_2 ausgehenden no-Reduktionen endlich.

Fall 4) Der no-Redex von t_1 ist *schwach nichtdeterministisch oder deterministisch* und vom Typ *lbeta.*
Die entstehenden Reduktionszusammenhänge zeigt Abbildung 4.12. Lemma 4.3.6 sowie die Korollare 4.2.1 und 4.2.2 sind zusammen mit der Induktionsvoraussetzung entsprechend der angefügten Nummerierung anzuwenden. Die längste von t_1' ausgehende no-Reduktion ist mindestens einen no-Schritt kürzer als die längste von t_1 ausgehende no-Reduktion. Gemäß Korollar 4.2.1 folgt daraus, daß alle von t_1'' ausgehenden no-Reduktionen endlich sind und die längste von t_1'' ausgehende no-Reduktion kürzer oder genauso lang wie die längste von t_1' ausgehende no-Reduktion ist. Somit können wir auf t_1'' die Induktionsvoraussetzung anwenden und erhalten $t_2''\not\Uparrow$. Wir können nun Korollar 4.2.2 anwenden und erhalten $t_2'\not\Uparrow$. Da die no-Reduktion von t_2 zu t_2' über keinerlei *nd*-Reduktionen verfügt, folgt daraus direkt, daß alle von t_2 ausgehenden no-Reduktionen endlich sind. □

Wir können nun zum Beweis des Schlüsseltheorems dieses Abschnitts übergehen, das besagt, daß zwei Ausdrücke, die mittels *detpar* in dieselbe *detpar*-Normalform überführt werden können, kontextuell äquivalent sind. Zunächst beweisen wir in einer Vorstufe die wechselseitige Übertragbarkeit einer no-Reduktion zu WHNF bzw. einer unendlichen no-Reduktion.

Satz 4.3.1. *Gegeben drei geschlossene Ausdrücke* t_1, t_2, s. *Wenn* $t_1 \xrightarrow{detpar} s$ *und* $t_2 \xrightarrow{detpar} s$, *dann*

$$t_1\Downarrow \iff t_2\Downarrow \quad \textit{und} \quad t_1\Uparrow \iff t_2\Uparrow.$$

Beweis. Es reicht wegen der Symmetrie der Aussage bezüglich t_1 und t_2 jeweils nur eine Richtung zu zeigen. Wir zeigen stellvertretend die Richtung von t_1 zu t_2.

$$\begin{tikzcd}
t_1 \arrow[rr, "detpar"] \arrow[dd, "{no,*}"'] & & s & & t_2 \arrow[ll, "detpar"'] \arrow[dd, dashed, "{no,*}"] \\
& & \text{Lemma 4.3.9} \; \text{①} & & \\
t_{1.WF} \arrow[r, dashed, "{i,lbeta,*}"'] & t'_{1.WF} \arrow[r, dashed, "detpar"'] & s_{WF} & t'_2 \arrow[l, dashed, "detpar"'] \arrow[dd, dashed] \arrow[dd, "{no,*}"] & t''_2 \arrow[l, dashed, "{i,lbeta,*}"'] \arrow[dd, dashed, "{no,*}"] \\
\text{Lemma 2.3.4} \; \text{②} & \text{Lemma 4.3.2} \; \text{③} & \text{Lemma 4.3.3} \; \text{④} & \text{Lemma 4.2.1} \; \text{⑤} & \\
& & & t'_{2.WF} & t''_{2.WF} \arrow[l, dashed, "{i,lbeta,*}"'] \\
& & & \text{Korollar 2.3.2} \; \text{⑥} &
\end{tikzcd}$$

Abbildung 4.13:

Konvergenz
Wir betrachten dazu das in Abbildung 4.13 enthaltene Reduktionsdiagramm und zeigen die Übertragbarkeit der Konvergenz indem wir der Nummerierung folgen. $t_{1.WF}$ sei ein Ausdruck in WHNF und die no-Reduktion von t_1 zu $t_{1.WF}$ gegeben.

1. Gemäß Lemma 4.3.9 existieren die 4 Ausdrücke $t'_{1.WF}, s_{WF}, t'_2, t''_2$, so daß die Zusammenhänge des Reduktionsdiagramms aus Abbildung 4.13 erfüllt werden.

2. Lemma 2.3.4 besagt, daß eine *lbeta*-Reduktion einen Ausdruck niemals aus WHNF herausbewegt. Daraus folgt, daß sich $t'_{1.WF}$ in WHNF befindet.

3. Lemma 4.3.2 impliziert, daß sich auch der Ausdruck s_{WF}, die *detpar*-Normalform von $t'_{1.WF}$, in WHNF befindet.

4. Daraus folgt gemäß Lemma 4.3.3, daß ausgehend von t'_2 eine no-Reduktion zu einem Ausdruck $t'_{2.WF}$ existiert, der sich WHNF befindet.

5. Wir benutzen die Vertauschungsregeln einer $(i, lbeta, *)$-Reduktion aus Lemma 4.2.1 und erhalten in der Folge die Existenz des Ausdrucks $t''_{2.WF}$.

6. Korollar 2.3.2 besagt, daß aus $t'_{2.WF}$ in WHNF folgt, daß auch der Ausdruck $t''_{2.WF}$ in WHNF ist, womit wir die Übertragbarkeit der Konvergenz gezeigt haben.

Divergenz
Die wechselseitige Übertragbarkeit der Divergenz folgt direkt aus Lemma 4.3.10 in Verbindung mit einer einfachen Widerspruchsüberlegung. □

Mit Hilfe des obigen Lemmas können wir nun abschließend das *detpar*-Lemma zeigen.

Satz 4.3.2. *(detpar-Lemma) Gegeben drei geschlossene Ausdrücke* t_1, t_2, s *mit* $t_1 \xrightarrow{detpar} s$ *und* $t_2 \xrightarrow{detpar} s$. *Dann* $t_1 \sim_c t_2$.

Beweis. Für alle Kontexte C existiert gemäß Lemma 4.1.1 ein eindeutiger Ausdruck s', so daß $C[t_1] \xrightarrow{detpar} s'$ und $C[t_2] \xrightarrow{detpar} s'$ gilt. Aus Satz 4.3.1 folgt wegen der Existenz von s'

$$C[t_1]\Downarrow \iff C[t_2]\Downarrow \text{ und } C[t_1]\Uparrow \iff C[t_2]\Uparrow$$

womit die Hypothese gezeigt ist. □

4.4 Die Reduktionen *dcp* und *lsh*

Wir werden nun zwei Reduktionen kennenlernen, deren Definition auf deterministische Subausdrücke zurückgreift.

Definition 4.4.1. *Wir definieren folgende Reduktion:*

(*dcp*) $C[(\texttt{let}\ x = s\ \texttt{in}\ D[x])] \xrightarrow{dcp} C[(\texttt{let}\ x = s\ \texttt{in}\ D[s])]$
falls s ein deterministischer Subausdruck

(*lsh*) $C[\lambda y.(\texttt{let}\ x = s\ \texttt{in}\ t)] \xrightarrow{lsh} C[\texttt{let}\ x = s\ \texttt{in}\ \lambda y.t]$
falls s ein deterministischer Subausdruck

Das Kürzel *dcp* steht für "deterministisches Kopieren". Es ist offensichtlich, daß die *dcp*-Reduktion eine Vorstufe der *detpar*-Reduktion darstellt, da jede *detpar*-Reduktion mittels einer Folge von *dcp*- und *ldel*-Reduktionen simulierbar ist. Für die später vorkommende Untersuchung des "fully lazy lambda-lifting" werden wir jedoch diese 'reduzierte Form' benötigen.
lsh steht für "let-shift", da diese Reduktion eine let-Bindung aus einer Abstraktion herausbewegt. Wir werden diese Reduktion nur kurz vorstellen; sie ist interessant im Zusammenhang mit der Optimierung funktionaler Programme, steht jedoch für sich alleine und wird in keinem späteren Beweis verwendet.

Der Beweis der kontextuellen Äquivalenz bei den beiden zuvor eingeführten Reduktionen erfolgt nach einem einfachen Schema:

> Es wird zunächst gezeigt, daß eine Anwendung der Reduktion keinen Einfluß auf die *detpar*-Normalform hat, die jeweils durch die Anwendung von *detpar* erhalten wird. Das dadurch entstehende Bild zeigt Abbildung. Dann wird Lemma 4.3.2 angewendet, woraus direkt die kontextuelle Äquivalenz folgt.

Um die kontextuelle Äquivalenz von *dcp* zu zeigen benutzen wir *detpar* und zeigen, daß eine *dcp*-Reduktion zu keiner Änderung der *detpar*-Normalform führt.

Lemma 4.4.1. *Gegeben zwei geschlossene Ausdrücke t und t' mit $t \xrightarrow{dcp} t'$. Dann existiert ein Ausdruck s, so daß folgende Reduktionszusammenhänge gelten:*

$$\begin{tikzcd} t \arrow[rr, "dcp"] \arrow[dr, dashed, "detpar"'] & & t' \arrow[dl, dashed, "detpar"] \\ & s & \end{tikzcd}$$

Beweis. Die *dcp*-Reduktion habe o.b.d.A. die Form $C[\texttt{let}\ x = t_x\ \texttt{in}\ D[x,\dots,x]] \xrightarrow{dcp} C[\texttt{let}\ x = t_x\ \texttt{in}\ D[t_x,\dots,x]]$, d.h. sie kopiere in das erste Loch des Multikontext D. Dabei komme die Variable x im Multikontext D nicht weiter vor, ansonsten ist ein anderer Multikontext mit entsprechend mehr Löchern zu wählen. Es reicht die Hypothese isoliert im Bezug auf den Ausdruck $\texttt{let}\ x = t_x\ \texttt{in}\ D[x,\dots,x]$ zu zeigen, die Aussage des Lemmas ergibt sich dann durch Ausdehnung auf den umgebenden Kontext C mittels Lemma 4.1.1.

Dann gilt $\texttt{let}\ x = t_x\ \texttt{in}\ D[x,\dots,x] \xrightarrow{detpar} D'[t'_x/x]$, wobei D' derjenige Kontext sei, der nach der Anwendung von *detpar* auf D erhalten wird und t'_x die *detpar*-Normalform von t_x sei. D' kann durchaus mehr Löcher als D enthalten, dies ist jedoch nicht weiter störend. Es ist offensichtlich, daß von $\texttt{let}\ x = t_x\ \texttt{in}\ D[t_x,\dots,x]$ dieselbe *detpar*-Normalform $D'[t'_x/x]$ erreicht wird, da in t_x keine Variablen vorkommen können, die in D gebunden sind. Wegen der zuvor aufgeführten Ausdehnbarkeit ist die Hypothese damit gezeigt. □

Lemma 4.4.2. *Gegeben zwei geschlossene Ausdrücke t und t' mit $t \xrightarrow{lsh} t'$. Dann existiert ein Ausdruck s, so daß folgende Reduktionszusammenhänge gelten:*

$$\begin{tikzcd} t \arrow[rr, "lsh"] \arrow[dr, dashed, "detpar"'] & & t' \arrow[dl, dashed, "detpar"] \\ & s & \end{tikzcd}$$

Beweis. Die *lsh*-Reduktion habe die Form $C[\lambda y.(\texttt{let}\ x = s\ \texttt{in}\ t)] \xrightarrow{detpar} C[\lambda y.(t'[s'/x])]$. Dann gilt $\lambda y.(\texttt{let}\ x = s\ \texttt{in}\ t) \xrightarrow{detpar} \lambda y.(t'[s'/x])$ und $\texttt{let}\ x = s\ \texttt{in}\ \lambda y.t \xrightarrow{detpar} (\lambda y.t')[s'/x]$ mit $s \xrightarrow{detpar} s'$ und $t \xrightarrow{detpar} t'$. Aus der Definition der Substitution folgt $(\lambda y.t')[s'/x] \equiv \lambda y.(t'[s'/x])$. Die Hypothese folgt dann direkt aus der in Lemma 4.1.1 gezeigten Ausdehnbarkeit auf den umgebenden Kontext C. □

Wir zeigen zum Abschluß die Erhaltung der kontextuelle Äquivalenz durch eine *dcp*- wie lsh-Reduktion.

Satz 4.4.1. *Gegeben zwei Ausdrücke t, s. Wenn $t \xrightarrow{dcp} s$ oder $t \xrightarrow{lsh} s$, dann $t \sim_c s$.*

Beweis. Gemäß Lemma 4.4.1 bzw. 4.4.2 folgt aus $t \xrightarrow{a} s$ für $a \in \{dcp, lsh\}$, daß ein eindeutiger Ausdruck s' existiert, so daß $t \xrightarrow{detpar} s'$ und $s \xrightarrow{detpar} s'$ gilt. Die kontextuelle Äquivalenz von t und s folgt dann direkt aus Satz 4.3.2. □

4.5 fully-lazy Lambda-Lifting

Der Begriff "fully-lazy Lambda-Lifting" steht für eine Optimierung, die die Evaluation funktionaler Programme mittels einer abstrakten Maschine wie der G-Maschine beschleunigt, indem verstecktes Sharing nutzbar gemacht wird. Für eine genauere Beschreibung der Wirkungsweise von fully-lazy Lambda-Lifting sei auf [PJ87] verwiesen. Dort wird der erhaltene Vorteil anhand von Beispielen ausführlich dargelegt. Wir werden in diesem Abschnitt zeigen, daß fully-lazy Lambda-Lifting im Kontext des Λ_{let}-Kalkül in einer eingeschränkten Form zulässig ist. Die Einschränkungen sind eine Folge des im Λ_{let}-Kalkül vorhandenen Nichtdeterminismus und werden am Ende des Abschnitts dargelegt.

Um fully-lazy Lambda-Lifting im Kontext des Λ_{let} betrachten zu können, führen wir den Begriff des "reproduzierbaren Subausdrucks" ein:

Definition 4.5.1. *t sei ein Ausdruck. Ein Subausdruck t_s von t heißt* reproduzierbar, *falls t_s*

- *eine Abstraktion oder die Konstante* `choice`
- *eine Variable*
- *ein deterministischer Subausdruck*

Wie der Begriff "reproduzierbar" bereits nahelegt, sind alle Ausdrucksformen, die in der Definition eines reproduzierbaren Ausdrucks aufgeführt werden, kopierbar. Wir definieren nun eine Reduktion, die dem liften maximal freier Ausdrücke, wie in [PJ87] vorgestellt, gleichkommt:

Definition 4.5.2. *$D[t_1, \ldots, t_n]$ sei ein Ausdruck, bei dem alle t_i reproduzierbare Subausdrücke darstellen. Wir definieren die folgende Reduktion, die einer verallgemeinerten Form des Liftens maximal freier Ausdrücke entspricht:*

$$(ll - dmfe) \quad C[D[t_1, \ldots, t_n]] \xrightarrow{ll-dmfe} C[(\lambda z_1, \ldots, z_n.D[z_1, \ldots, z_n])t_1 \ldots t_n]$$

Wir zeigen nun, daß eine $ll-dmfe$-Reduktion die kontextuelle Äquivalenz zweier Ausdrücke bewahrt.

Satz 4.5.1. *t, s seien zwei Ausdrücke mit $t \xrightarrow{ll-dmfe} s$. Dann $t \sim_c s$.*

Beweis. Für alle Kontexte C ist folgende Entwicklung möglich:

$$\begin{array}{rl} & C[(\lambda z_1, \ldots, z_n.D[z_1, \ldots, z_n])t_1, \ldots, t_n] \\ \xrightarrow{lbeta} & C[(\texttt{let } z_1 = t_1 \texttt{ in } \lambda z_2, \ldots, z_n.D[z_1, \ldots, z_n])t_2, \ldots, t_n] \\ \xrightarrow{a} & C[(\texttt{let } z_1 = t_1 \texttt{ in } \lambda z_2, \ldots, z_n.D[t_1, \ldots, z_n])t_2, \ldots, t_n] \\ \xrightarrow{ldel} & C[(\lambda z_2, \ldots, z_n.D[t_1, \ldots, z_n])t_2, \ldots, t_n] \end{array}$$

$$\text{wobei } a = \begin{cases} cp & \text{falls } t_1 \text{ eine Abstraktion oder die Konstante } \texttt{choice} \\ lcv & \text{falls } t_1 \text{ eine Variable} \\ dcp & \text{falls } t_1 \text{ ein deterministischer Subausdruck von } t \end{cases}$$

Alle oben benutzten Reduktionen erhalten die kontextuelle Äquivalenz. Durch Induktion über die Anzahl n von Abstraktionen und wiederholte Anwendung der obigen Entwicklung erhalten wir

$$C[(\lambda z_1, \ldots, z_n.D[z_1, \ldots, z_n])t_1, \ldots, t_n] \sim_c C[D[t_1, \ldots, t_n]]$$

und damit die Hypothese. □

An dieser Stelle sei angemerkt, daß die *ucp*-Reduktion im Kontext des full-lazy Lambda-Lifting nicht eingesetzt werden kann, da das Kopierziel bei der in obigem Beweis vorkommenden Entwicklung stets unterhalb einer Abstraktion liegt, was bei einer *ucp*-Reduktion nicht zulässig ist.

Das folgende Beispiel zeigt die Anwendung von fully-lazy Lambda-Lifting für einen gegebenen Ausdruck.

Beispiel 4.5.1. Gegeben sei

$t_1 \equiv \texttt{let } x = t_x \texttt{ in } (\texttt{let } y = t_y \texttt{ in } \lambda z.z(xy))$ mit t_x und t_y als deterministischen Ausdrücken.

(xy) ist reproduzierbar sowie gemäß [PJ87] ein maximal freier Subausdruck und wird durch das Liften aus der Abstraktion herausbewegt. Als Ergebnis des fully-lazy-Liftens erhalten wir somit:

$$t_2 \equiv \texttt{let } x = t_x \texttt{ in } (\texttt{let } y = t_y \texttt{ in } (\lambda b.\lambda z.zb)(xy))$$

Wir überführen mittels einer Folge von Reduktionen, wie in obigem Beweis gezeigt, den Ausdruck t_2 in den Ausdruck t_1:

$$\begin{aligned}
&\texttt{let}\ x = t_x\ \texttt{in}\ (\texttt{let}\ y = t_y\ \texttt{in}\ (\lambda b.\lambda z.zb)(xy)\,)\\
\xrightarrow{lbeta}\ &\texttt{let}\ x = t_x\ \texttt{in}\ (\texttt{let}\ y = t_y\ \texttt{in}\ \texttt{let}\ b = xy\ \texttt{in}\ \lambda z.zb\,)\\
\xrightarrow{dcp}\ &\texttt{let}\ x = t_x\ \texttt{in}\ (\texttt{let}\ y = t_y\ \texttt{in}\ \texttt{let}\ b = xy\ \texttt{in}\ \lambda z.z(xy)\,)\\
\xrightarrow{ldel}\ &\texttt{let}\ x = t_x\ \texttt{in}\ (\texttt{let}\ y = t_y\ \texttt{in}\ \lambda z.z(xy)\,)
\end{aligned}$$

4.5.1 Grenzen des fully-lazy Lambda-Lifting beim Λ_{let}-Kalkül

Das in [PJ87] vorgestellte fully-lazy Lambda-Lifting ist beim Λ_{let}-Kalkül nicht in allen Fällen zulässig. Dies ist darin begründet, daß der Begriff eines "maximal freien Ausdrucks ", der in [PJ87] als Grundlage des "fully-lazy Lambda-Lifting" dient, nur partiell mit dem eines reproduzierbaren Ausdrucks vereinbar ist. Die in [PJ87] getroffene Definition lautet:

Definition 4.5.3. *Ein Subausdruck t_s einer Abstraktion r ist* frei in r*, wenn alle Variablen in t_s frei in r vorkommen. Eine* maximal freier Subausdruck *von r ist ein freier Subausdruck, der nicht ein echter Subausdruck eines anderen freien Ausdrucks in r ist. (s ist ein echter Subausdruck von t gdw. s ein Subausdruck von t ist und $t \not\equiv s$ gilt.)*

Im obigen Beispiel ist (xy) ein maximal freier Ausdruck, wohingegen die einzelnen Variablen x und y nur frei vorkommen. $(z(xy))$ hingegen ist weder frei noch maximal frei, da die Variable z durch eine Abstraktion gebunden wird. Es ist offensichtlich, daß nicht jeder maximal frei vorkommender Subausdruck zugleich reproduzierbar ist, da die freien Variablen Ausdrücke binden können, die nichtdeterministisch sind. Die Einschränkung auf deterministische Subausdrücke ist jedoch zwingend erforderlich, wie das folgende Beispiel zeigt.

Beispiel 4.5.2. Gegeben sei der Ausdruck

$$t \equiv \texttt{let}\ y = \lambda x.((\texttt{choice}\ I)x)\ \texttt{in}\ (y\,K)(y\,K)$$

mit den üblichen Kombinatoren $I \equiv \lambda x.x$ und $K \equiv \lambda x.\lambda y.x$. Der Subausdruck $\texttt{choice}\ I$ ist innerhalb von $\lambda x.((\texttt{choice}\ I)x)$ ein maximal freier Ausdruck und könnte somit bei [PJ87] geliftet werden. Der erhaltene Ausdruck wäre

$$t_l \equiv \texttt{let}\ y = (\lambda z.\lambda x.zx)(\texttt{choice}\ I)\ \texttt{in}\ (y\,K)(y\,K).$$

t und t_l sind jedoch nicht kontextuell äquivalent, da $t \xrightarrow{no,*} t_K$ mit $t_K \sim_c K$ jedoch ausgehend von t_l keine adäquate no-Reduktion zu einem Ausdruck existiert, der kontextuell äquivalent zu K wäre.

Eine genauere Betrachtung der Ursache des Fehlverhaltens bei obigem Beispiel zeigt zugleich die Wirkungsweise des Liftens maximal freier Ausdrücke. Bei t_l wurde effektiv mehr Sharing erzeugt, zuviel im Kontext des Nichtdeterminismus.

4.6 Bezug zum klassischen Lambda-Kalkül

In diesem Abschnitt wird der Bezug des Λ_{let}-Kalküls zum klassischen Lambda-Kalkül untersucht. Jeder Λ_{let}-Ausdruck, in dem weder die Konstante `choice` noch ein let-Ausdruck als Subausdruck vorkommt, ist zugleich ein Lambda-Ausdruck des klassischen Lambda-Kalkül. Wir werden zeigen, daß der Λ_{let}-Kalkül und der klassische Lambda-Kalkül bezogen auf geschlossene Ausdrücke ein gleichartiges Terminierungsverhalten zeigen. Zunächst definieren wir die Menge aller Λ-Ausdrücke, sowie die β-Reduktion entsprechend [Bar84].

Definition 4.6.1. *Auf den syntaktischen Elementen aus Definition 2.1.1 wird die Menge von Ausdrücken Λ wie folgt induktiv definiert:*

$$\begin{array}{lll} x \in \Lambda & & \\ s \in \Lambda & \Rightarrow & (\lambda x.s) \in \Lambda \\ s,t \in \Lambda & \Rightarrow & (st) \in \Lambda \end{array}$$

Dabei ist x eine beliebige Variable.

Die Gültigkeit von $\Lambda \subset \Lambda_{let}$ ist offensichtlich.

Definition 4.6.2. *Die β-Reduktion wird auf den Menge der Λ_{let}-Ausdrücke wie folgt definiert:*

$$(\beta) \quad C[(\lambda x.s)t] \xrightarrow{\beta} C[s[t/x]]$$

Das folgende Lemma zeigt, daß die β-Reduktion bei ihrer Anwendung zu einem kontextuell äquivalenten Ausdruck führt.

Lemma 4.6.1. *t sei ein geschlossener Λ-Ausdruck. Wenn $t \xrightarrow{\beta} t'$, dann $t \sim_c t'$.*

Beweis. Wir zeigen, daß jede β-Reduktion durch eine Folge bestehend aus *lbeta*-, *dcp*- und *ldel*-Reduktionen nachgeahmt werden kann. Daraus, daß jede dieser Reduktionen die kontextuelle Äquivalenz bewahrt, folgt die Hypothese:

$$\begin{array}{ll} & C[(\lambda x.s)t_x] \\ \xrightarrow{lbeta} & C[\texttt{let } x = t_x \texttt{ in } s] \\ \xrightarrow{dcp,*} & C[\texttt{let } x = t_x \texttt{ in } s[t_x/x]], \text{ wobei } (*) \\ \xrightarrow{ldel} & C[s[t_x/x]] \end{array}$$

$(*)$ Die $(dcp,*)$-Reduktion ersetze alle Vorkommen von x in s durch t_x. Da t gemäß Voraussetzung geschlossen ist und die Konstante `choice` nicht vorkommt, ist jeder Subausdruck von t deterministisch. □

Korollar 4.6.1. *t sei ein geschlossener Λ-Ausdruck. Wenn $t \xrightarrow{\beta,*} t'$, dann $t \sim_c t'$.*

Beweis. Folgt aus der Transitivität von $\sim_c$. □

Wir wollen nun den Beweis des umgekehrten Falls vorbereiten. Dazu formulieren wir zunächst ein Lemma, das einer leicht variierten Form von Lemma 4.3.5 entspricht. Die Variation betrifft die Beschränkung auf vollständig deterministische Ausdrücke und das Herausstellen einer versteckten Folge von β-Reduktionen.

Lemma 4.6.2. *Gegeben ein Λ_{let}-Ausdruck t, in dem die Konstante* `choice` *nicht vorkommt, sowie zwei Ausdrücke s, t' mit $t \xrightarrow{detpar} s$ und $t \xrightarrow{no,lbeta} t'$. Dann existiert eine Λ-Ausdruck s', so daß die folgenden Reduktionszusammenhänge erfüllt werden:*

$$\begin{tikzcd}
t \arrow[rr, "detpar"] \arrow[d, "{no,lbeta}"'] & & s \arrow[dd, dashed, "{\beta,*}"] \\
t' \arrow[d, dashed, "{no,lbeta,*}"'] & & \\
\circ \arrow[r, dashed, "{i,lbeta,*}"'] & \circ \arrow[r, dashed, "detpar"'] & s'
\end{tikzcd}$$

Beweis. Der Beweis dieses Lemma kann auf einfache Weise aus dem Beweis von Lemma 4.3.5 abgeleitet werden. Wir benötigen zunächst folgende Überlegung:
Da t geschlossen ist und über keine Vorkommen der Konstanten `choice` verfügt, sind alle Subausdrücke von t deterministisch. Daraus wiederum folgt, daß bei der Anwendung von *detpar* auf t alle let-Bindungen verschwinden und s ein Ausdruck aus Λ ist.

Wenn wir nun zeigen können, daß die Reduktion von s zu s' aus der zum Beweis von Lemma 4.3.5 gehörenden Abbildung 4.4 bei dem hier betrachteten speziellen Fall einer $(\beta, *)$-Reduktion entspricht, haben wir die Hypothese gezeigt. Dies zeigen wir wie folgt:
C sei ein Multikontext, so daß $s \equiv C[(\lambda x_1.s_1)t_1, \ldots, (\lambda x_n.s_n)t_n]$ gelte und sich in den Löchern des Kontexts C genau alle markierten *lbeta*-Redizes befinden. Wir verfolgen nun die Anwendung der *lbeta*- und *detpar*-Reduktionen in Abbildung 4.4 auf dem Weg von s zu s'.
Wir wenden die *lbeta*-Reduktionen an und erhalten $C[\texttt{let}\ x_1 = t_1\ \texttt{in}\ s_1, \ldots, \texttt{let}\ x_n = t_n\ \texttt{in}\ s_n]$. Die folgende *detpar*-Reduktion löst nun alle let-Ausdrücke auf, da alle Subausdrücke von s deterministisch sind, und wir erhalten $C[s_1[t_1/x_1], \ldots, s_n[t_n/x_n]]$. Offensichtlich entspricht das Resultat genau der Anwendung einer β-Reduktion auf jeden markierten *lbeta*-Redex in s. □

Lemma 4.6.3. *Gegeben ein geschlossener Λ_{let}-Ausdruck t, in dem die Konstante* `choice` *nicht vorkommt, sowie ein Ausdruck s mit $t \xrightarrow{detpar} s$ und ein Ausdruck*

Diagramm 1

$$\begin{tikzcd}
t \arrow[rr, "detpar"] \arrow[d, "{no, lbeta}"] \arrow[ddd, two heads, bend right] & & s \arrow[dd, dashed, "{\beta, *}"] \\
\circ \arrow[d, dashed, "{no, lbeta, *}"] \arrow[rr, phantom, "\text{Lemma 4.6.2} \; ①"] & & \\
t' \arrow[r, dashed, Rightarrow, "{i, lbeta, *}"'] \arrow[d, two heads, "{no, n}"'] \arrow[dr, phantom, "\text{Lemma 4.2.4} \; ②"] & t_d \arrow[r, dashed, Rightarrow, "detpar"'] \arrow[d, two heads, "{no, m}"'] \arrow[dr, phantom, "\text{Ind. Voraus.} \; ③"] & s' \arrow[d, dashed, "{\beta, *}"] \\
t_{WF} \arrow[r, dashed, "{i, lbeta, *}"'] & t'_{WF} \arrow[r, dotted, no head] & s_{WF}
\end{tikzcd}$$

wobei $n \geq m$

Diagramm 2

$$\begin{tikzcd}
t \arrow[rd, "detpar"] \arrow[d, "{no, \{llet, lapp, cp\}, +}"'] \arrow[rd, phantom, "①"'] & \text{Korollar 4.3.2} \\
t' \arrow[r, dashed, "detpar"] \arrow[d, "{no, *}"'] \arrow[rd, phantom, "\text{Ind. Voraus.} \; ②"] & s \arrow[d, dashed, "{\beta, *}"] \\
t_{WF} \arrow[r, dotted, no head] & s_{WF}
\end{tikzcd}$$

Abbildung 4.14:

t_{WF} mit $t \xrightarrow{no,*} t_{WF}$. Da existiert ein Λ-Ausdruck s_{WF}, so daß die folgenden Reduktionszusammenhänge erfüllt werden:

$$\begin{tikzcd}
t \arrow[rr, "detpar"] \arrow[d, "{no, *}"'] & & s \arrow[d, dashed, "{\beta, *}"] \\
t_{WF} \arrow[r, dashed, "{i, lbeta, *}"'] & \circ \arrow[r, dashed, "detpar"'] & s_{WF}
\end{tikzcd}$$

Beweis. Wir führen eine Induktion über die Länge der no-Reduktion von t zu t_{WF}.

Induktionsanfang $t \equiv t_{WHNF}$
Wir wählen $s' \equiv s$, gemäß Lemma 4.3.2 befindet sich s in WHNF.

Induktionsschritt
Wir differenzieren nach dem Typ des no-Redex von t:

1. Fall: Der no-Redex ist vom Typ *llet*, *lapp* oder *cp*:
Den Beweis der Hypothese zeigt Diagramm 2 aus Abbildung 4.14. Wir reduzieren ausgehend von t bis wir einen Ausdruck t' erreichen, der entweder in WHNF ist oder aber über einen no-Redex vom Typ *lbeta* verfügt. Gemäß Korollar 4.3.2 sind die *detpar*-Normalformen von t und t' identisch. s muß ein Ausdruck aus Λ sein, da alle Subausdrücke von t deterministisch sind. Die von t' ausgehende no-Reduktion zu t_{WF} ist kürzer als die von t ausgehende, somit können wir auf t' die Induktionshypothese anwenden.

2. Fall: Der no-Redex ist vom Typ *lbeta*:
Den Beweis der Hypothese zeigt Diagramm 1 aus Abbildung 4.14. Wir benutzen

zunächst Lemma 4.6.2 und erhalten die Existenz der drei Ausdrücke t', t_d, s', wobei s' ein Λ-Ausdruck ist. Wir wenden Lemma 4.2.4 auf die durch t', t_{WF}, t_d gebildete Gabel an und erhalten den Ausdruck t'_{WF}, der sich gemäß Lemma 2.3.4 wie t_{WF} in WHNF befinden muß. Ferner muß die no-Reduktion von t_d zu t'_{WF} kürzer oder gleich lang sein, wie die von t' zu t_{WF}. Daraus folgt, daß wir auf t_d die Induktionshypothese anwenden können und wir erhalten die Existenz des Λ-Ausdrucks s_{WF}.

Ein no-Redex vom Typ *nd* ist nicht möglich, da t gemäß Voraussetzung über keine Vorkommen der Konstanten `choice` verfügt. □

Das nun folgende Lemma bildet eine Erweiterung des zuvor gezeigten.

Lemma 4.6.4. *t sei ein geschlossener Λ-Ausdruck. Wenn $t\Downarrow$, dann existiert eine Abstraktion $\lambda x.s$, so daß $t \xrightarrow{\beta,*} \lambda x.s$.*

Beweis. Wir zeigen zunächst in einem ersten Schritt, daß jeder Λ-Ausdruck in WHNF die Form $\lambda x.s$ besitzen muß:

Jeder Ausdruck in WHNF hat die Form $L_R^*[r]$ mit r eine Abstraktion oder die Konstante `choice` . Da ein Λ-Ausdruck weder einen L_R-Kontext noch die Konstante `choice` beinhalten kann, muß dieser die Form $\lambda x.s$ besitzen, wenn er sich in WHNF befindet.

Da t ein Λ-Ausdruck ist und *detpar* auf Λ-Ausdrücken reflexiv wirkt, gilt $t \xrightarrow{detpar} t$. Daraus folgt unter Anwendung von Lemma 4.6.3, daß ein Λ-Ausdruck s' existiert, mit $t \xrightarrow{\beta,*} s'$ und s' in WHNF. Die zuvor aufgeführte Überlegung zeigt, daß s' die Form $\lambda x.s$ besitzen muß. □

Damit sind wir in der Lage zu zeigen, daß der call-by-name Evaluator des klassischen Lambda-Kalküls mit der no-Reduktion des Λ_{let}-Kalküls austauschbar ist.

Satz 4.6.1. *t sei ein geschlossener Λ-Ausdruck. folgendes gilt:*

$$t\Downarrow \iff \exists \text{ eine Abstraktion } \lambda x.s : t \xrightarrow{\beta,*} \lambda x.s$$

Beweis.

$\Rightarrow$
Zeigt Lemma 4.6.4.

$\Leftarrow$
Aus Korollar 4.6.1 folgt $t \sim_c \lambda x.s$. Daraus folgt wegen der Definition der kontextuellen Äquivalenz direkt $t\Downarrow$. □

4.7 Abschließende Bemerkungen

In diesem Kapitel wurde der Begriff des deterministischen Subausdrucks eingeführt und gezeigt, daß das Liften deterministischer Subausdrücke eine zulässige Transformation auf Ausdrucksebene ist. Damit wurde für eine Optimierung, die im Zusammenhang mit der Implementation verzögernd auswertender funktionaler Sprachen von zentraler Bedeutung ist, in einer beschränkten Form die Zulässigkeit gezeigt. Ferner wurde die Beziehung zwischen dem klassischen Lambda-Kalkül und dem Λ_{let}-Kalkül untersucht. Letztere Beziehung soll abschließend noch einmal aufgegriffen werden.

Beim klassischen Lambda-Kalkül entspricht die wiederholte Reduktion des am weitesten links stehenden Redex einer normalisierenden Reduktionsstrategie, d.h. sie führt zu einem Ausdruck in Normalform, sofern irgendwie zu einem Ausdruck in Normalform reduziert werden kann([Bar84], Lemma 13.2.2). Der Begriff Normalform ([Bar84], Definition 3.1.8) ist dabei stärker als der der WHNF, da Normalform komplette Redexfreiheit impliziert. Jedoch kann dieser Unterschied im Kontext der nun folgenden Aussagen vernachlässigt werden.
Die Reduktionsstrategie, der wiederholten Reduktion des am weitesten links stehenden Redex ist beim klassischen Lambda-Kalkül mit call-by-name Auswertung identisch. Satz 4.6.1 läßt damit die Aussage zu, daß der call-by-name Evaluator des klassischen Lambda-Kalküls mit der no-Reduktion des Λ_{let}-Kalküls bei Λ-Ausdrücken austauschbar ist. Damit steht der Λ_{let}-Kalkül bei einer rein deterministischen Sicht in der Tradition des klassischen Lambda-Kalküls, was eine wünschenswerte Eigenschaft ist.

Kapitel 5

Rekursion und Fixpunkte

Ein bisher nicht betrachteter Aspekt des Λ_{let}-Kalküls ist Rekursion. Der Λ_{let}-Kalkül erlaubt keine `letrec`-Ausdrücke, d.h. rekursive let-Konstrukte, und verfügt damit über keine Möglichkeit, Rekursion auf syntaktischer Ebene auszudrücken.
Beim klassischen Lambda-Kalkül - auch dieser erlaubt keine rekursiven Konstrukte auf syntaktischer Ebene - sind Betrachtungen dieses Problemkreises bereits aus der Frühphase bekannt [Tur37] (siehe dazu auch die Einleitung von Abschnitt 6.3 in [Bar84]). In diesem Zusammenhang erscheint dort der Begriff des Fixpunktes. Ein Fixpunkt eines Ausdrucks t ist beim klassischen Lambda-Kalkül ein Ausdruck p, mit der Eigenschaft $t\,p =_\beta p$, wobei $=_\beta$ die zur Äquivalenzrelation erweiterte β-Reduktion ist. Mittels Fixpunkten ist es auf einfache Art und Weise möglich, Rekursion nachzuahmen. Wie dies geschieht, wird in [PJ87] anhand der Fakultätsfunktion vorgestellt.

In diesem Abschnitt wird gezeigt werden, daß auch beim Λ_{let}-Kalkül die Bildung von Fixpunkten möglich ist, und zwar auf der Grundlage der kontextuellen Äquivalenz. Jedoch ist die Fixpunktbildung hier, wie auch im Lambda-Kalkül, mit Tücken behaftet. So kann ein Ausdruck mehrere Fixpunkte potentiell sogar unendlich viele Fixpunkte besitzen, die, wie wir gleich an einem Beispiel sehen werden, sehr unterschiedlich sein können.
Ein übliches Beispiel für einen Ausdruck mit unendlich vielen Fixpunkten ist die Identität $\lambda x.x$. Tatsächlich ist jeder Λ_{let}-Ausdruck Fixpunkt der Identität; mittels der im Abschnitt über kontextuelle Äquivalenz betrachteten *ucp*-Reduktion ist leicht verifizierbar, daß für einen beliebigen Λ_{let}-Ausdruck p die Äquivalenz $(\lambda x.x)p \sim_c p$ gilt. Abhängig von der Wahl von p kann ausgehend von $(\lambda x.x)p$ eine no-Reduktion zu WHNF existieren oder auch nicht, somit können die verschiedenen Fixpunkte offensichtlich ein unterschiedliches Terminierungsverhalten bewirken.
Diese Beobachtung führt zu der Fragestellung, welches der korrekte Fixpunkt ist, d.h. derjenige Fixpunkt ist, der unsere Vorstellung von Rekursion modelliert. Um

diese Fragestellung zu beantworten, ist es erforderlich, die Fixpunkte gemäß eines sinnvollen Kriteriums zu bewerten. Im klassischen Lambda-Kalkül wird diese Bewertung üblicherweise mittels einer denotationalen Semantik erzielt. Eine gute Darstellung dieses Themenkreises findet sich in [Sto77]. Von Bedeutung für unsere weiteren Überlegungen ist in diesem Zusammenhang, daß Fixpunkte dort zunächst rein semantisch auf Basis einer cpo (complete partial order) gebildet werden. Eine Motivation für den Rückgriff auf cpo's ist, Lambda-Ausdrücke mittels einer Abbildung auf Elemente einer cpo nach ihrem Informationsgehalt zu ordnen. Das schwächste Element einer cpo wird im allgemeinen mit $\perp$, gesprochen 'Bot', bezeichnet, und steht für "keine Information". Der gedankliche operationale Bezug zu "keine Information" ist dabei Nichtterminierung.

Komplexere denotationale Betrachtungen, wie die Bildung einer cpo für den Λ_{let}-Kalkül seien außerhalb dieser Arbeit und an die weitere Forschung übergeben. Einen Eindruck, wie schwierig es ist, eine cpo für einen nichtdeterministischen Kalkül zu bilden, gibt [Plo76]. Wir werden jedoch in diesem Abschnitt sehen, daß auch Wege existieren, ohne Rückgriff auf eine cpo interessante semantische Aussagen treffen zu können sowie einen Korrektheitsbegriff für Fixpunkte zu formulieren. Dazu greifen wir auf die in Abschnitt 3.1 definierte Präkongruenz $\leq_c$ zurück. Mittels letzterer ist es möglich, Ausdrücke nach ihrem Verhalten in allen umgebenden Kontexten zu ordnen.
Wir werden zu Beginn dieses Abschnitts zeigen, daß alle Ausdrücke ohne eine Reduktion zu einer WHNF oder VWHNF bezüglich der Präkongruenz $\leq_c$ kleinste Elemente sind. Wir betrachten alle diese kleinsten Ausdrücke als eine Klasse und assoziieren diese mit $\perp$. Danach führen wir die Begriffe *lub* und *club* auf Basis von $\leq_c$ ein; diese Vereinbarungen geschehen in Anlehnung an die Bildung von kleinsten oberen Schranken bei cpo's. Um einen Eindruck von der Vorstellung zu geben, die dem Begriff Rekursion beim Λ_{let}-Kalkül in der vorliegenden Arbeit zugrundeliegt, erscheint im Anschluß ein Abschnitt, der anhand der Fakultätsfunktion zeigt, was sich hinter den gesuchten Fixpunkten verbirgt und wie diese motiviert sind. Danach wird der Begriff des Fixpunktkombinators eingeführt und ein spezifischer Fixpunktkombinator Y_T vorgestellt. Abschließend wird für Y_T ein *club*-Beweis geführt, d.h. gezeigt, daß Y_T die Eigenschaften eines als korrekt angesehenen Fixpunktkombinators besitzt.

5.1 Ausdrücke ohne WHNF und VWHNF

In diesem Abschnitt wird gezeigt werden, daß Ausdrücke, die über keine no-Reduktion zu WHNF bzw. VWHNF verfügen, bezüglich der Präkongruenz $\leq_c$ kleinste Elemente sind. Dazu werden zunächst drei Paare aus jeweils einem Lemma und einem Korollar gebildet, in denen Aussagen bezüglich der Übertragbarkeit von Terminierungseigenschaften über Kontexte hinweg gezeigt werden.

Lemma 5.1.1. *Für alle Λ_{let}-Ausdrücke t_x, t:*

- $\texttt{let}\ x = t_x\ \texttt{in}\ t{\Downarrow}_{WHNF} \implies t{\Downarrow}_{WHNF} \vee t{\Downarrow}_{VWHNF}$
- $\texttt{let}\ x = t_x\ \texttt{in}\ t{\Downarrow}_{VWHNF} \implies t{\Downarrow}_{VWHNF}$
- $\texttt{let}\ x = t_x\ \texttt{in}\ t\not\Uparrow \implies t\not\Uparrow$

Beweis. Auf jedem no-Pfad ausgehend von $\texttt{let}\ x = t_x\ \texttt{in}\ t$ wird danach differenziert, ob t_x irgendwann an einem no-Redex beteiligt ist oder aber über den ganzen no-Pfad hinweg unberührt bleibt.
Ist t_x auf einem terminierenden no-Pfad niemals in einen no-Redex involviert, dann terminiert derselbe no-Pfad beschränkt auf t, wie durch Induktion über die Anzahl der no-Schritte zu WHNF bzw. VWHNF auf dem no-Pfad leicht gezeigt werden kann.
Ist t_x auf einem no-Pfad $\mathcal{P}$ in einen no-Redex involviert, so muß t dafür zunächst die Form $W[x]$ erreichen, was ein Abgleich mit den in 2.3.1 vorgestellten Kontextformen der verschiedenen Arten von no-Redizes sofort zeigt. Bei Beschränkung auf den Ausdruck t führt der no-Pfad $\mathcal{P}$ dann zu $W[x]$, wie durch Induktion über die Anzahl der no-Schritte bis zum Erreichen von $\texttt{let}\ x = t_x\ \texttt{in}\ W[x]$ leicht gezeigt werden kann. Ein Ausdruck $W[x]$ mit x als frei vorkommender Variable befindet sich jedoch gemäß Satz 2.3.1 stets in VWHNF.
Da die obige Argumentation Pfad-bezogen erfolgt, werden alle Implikationen, in Kombination mit König's Lemma auch die all-quantifizierte dritte Implikation, der Hypothese bewiesen. □

Das obige Lemma läßt verschiedene Schlußfolgerungen zu, die in dem untenstehenden Korollar zusammengefaßt sind.

Korollar 5.1.1.

1. $L_R^*[t]{\Downarrow}_{WHNF \vee VWHNF} \implies t{\Downarrow}_{WHNF \vee VWHNF}$
2. $L_R^*[t]\not\Uparrow \implies t\not\Uparrow$
3. $t\not\Downarrow \implies L_R^*[t]\not\Downarrow$
4. $t\Uparrow \implies L_R^*[t]\Uparrow$

Beweis. Die ersten beiden Implikationen können einfach durch Induktion über die Tiefe des L_R^*-Kontext gezeigt werden. Die beiden letzten Implikationen sind die logischen Umkehrungen der beiden ersten. □

Wir führen nun dieselbe Untersuchung für Applikationen.

Lemma 5.1.2. *Für alle Λ_{let}-Ausdrücke s, t und $n \geq 0$:*

- $L^*_R[(A^n_L[s])\,t]{\Downarrow}_{WHNF} \ \Rightarrow\ L^*_R[(A^n_L[s])]{\Downarrow}_{WHNF}$
- $L^*_R[(A^n_L[s])\,t]{\Downarrow}_{VWHNF} \ \Rightarrow\ L^*_R[(A^n_L[s])]{\Downarrow}_{VWHNF} \vee L^*_R[(A^n_L[s])]{\Downarrow}_{WHNF}$
- $L^*_R[(A^n_L[s])\,t]{\not\Uparrow} \ \Rightarrow\ L^*_R[(A^n_L[s])]{\not\Uparrow}$

Beweis. Wir werden folgende Gegenüberstellungen von no-Entwicklungen benötigen:

$$\begin{array}{lclc}
L1.1 & L^*_R[A^n_L[x]\,t] & R1.1 & L^*_R[A^n_L[x]] \\
 & \xrightarrow{no,\omega} & & \xrightarrow{no,\omega} \\
L1.2 & L^*_R[A^n_L[\lambda z.s]\,t] & R1.2 & L^*_R[A^n_L[\lambda z.s]] \\
 & \xrightarrow{no,lbeta} & & \xrightarrow{no,lbeta}\ (*) \\
L1.3 & L^*_R[A^{n-1}_L[\texttt{let}\ x = t_x\ \texttt{in}\ s]\,t] & R1.3 & L^*_R[A^{n-1}_L[\texttt{let}\ x = t_x\ \texttt{in}\ s]] \\
 & \xrightarrow{no,lapp,n} & & \xrightarrow{no,lapp,n-1} \\
L1.4 & L^*_R[\texttt{let}\ x = t_x\ \texttt{in}\ (A^{n-1}_L[s]\,t)] & R1.4 & L^*_R[\texttt{let}\ x = t_x\ \texttt{in}\ A^{n-1}_L[s]] \\
 & \equiv & & \equiv \\
 & L^*_R[A^{n-1}_L[s]\,t] & & L^*_R[A^{n-1}_L[s]]
\end{array}$$

$(*)$ falls $n > 0$ sonst WHNF

$$\begin{array}{lclc}
L2.1 & L^*_R[A^n_L[x]\,t] & R2.1 & L^*_R[A^n_L[x]] \\
 & \xrightarrow{no,\omega} & & \xrightarrow{no,\omega} \\
L2.2 & L^*_R[A^n_L[\texttt{choice}\ ]\,t] & R2.2 & L^*_R[A^n_L[\texttt{choice}\ ]] \\
 & \xrightarrow{no,nd,x}\ (*) & & \xrightarrow{no,nd,x} \\
L2.3 & L^*_R[A^n_L[\lambda x.\lambda y.x]\,t] & R2.3 & L^*_R[A^n_L[\lambda x.\lambda y.x]] \\
 & \xrightarrow{no,3}\ (**) & & \xrightarrow{no,3}\ (***) \\
L2.4 & L^*_R[(A^{n-2}_L[\texttt{let}\ x = t_x\ \texttt{in}\ \texttt{let}\ y = t_y\ \texttt{in}\ x])\,t] & R2.4 & L^*_R[A^{n-2}_L[\texttt{let}\ x = t_x\ \texttt{in}\ \texttt{let}\ y = t_y\ \texttt{in}\ x]] \\
 & \xrightarrow{lapp,2*n-2} & & \xrightarrow{lapp,2*n-2} \\
L2.5 & L^*_R[\texttt{let}\ x = t_x\ \texttt{in}\ \texttt{let}\ y = t_y\ \texttt{in}\ A^{n-2}_L[x]\,t] & R2.5 & L^*_R[\texttt{let}\ x = t_x\ \texttt{in}\ \texttt{let}\ y = t_y\ \texttt{in}\ A^{n-2}_L[x]] \\
 & \equiv & & \equiv \\
 & L^*_R[A^{n-2}_L[x]\,t] & & L^*_R[A^{n-2}_L[x]]
\end{array}$$

$(*)$ $x \in \{left, right\}$; es wird nur der Linkszweig weiterentwickelt; die Entwicklung des Rechtszweigs erfolgt simultan.
$(**)$ falls $n > 1$ sonst WHNF
$(***)$ falls $n > 2$ sonst WHNF

$$\begin{array}{lclc} L3.1 & L_R^*[A_L^n[x]\,t] & R3.1 & L_R^*[A_L^n[x]] \\ & \xrightarrow{no,\omega} & & \xrightarrow{no,\omega} \\ L3.2 & L_R^*[\texttt{let}\ x = W[z]\ \texttt{in}\ L_R^*[A_L^n[x]\,t]] & R3.2 & L_R^*[\texttt{let}\ x = W[z]\ \texttt{in}\ L_R^*[A_L^n[x]]] \\ & \equiv & & \equiv \end{array}$$

z komme frei vor, daher VWHNF auf beiden Seiten

Wir führen den Beweis durch Induktion über die Menge aller Ausdrücke der Form $L_R^*[A_L^n[L_R^m[s]]\,t]$, wobei $n, m \geq 0$ und s kein Let-Ausdruck, die wir mittels der folgenden Bewertung ordnen:

> Das Bewertungsmaß besteht aus zwei Komponenten die lexikographisch geordnet sind. Die erste Komponente ist die Tiefe n des A_L^n-Kontexts, die zweite Komponente ist die Tiefe m des L_R^m-Kontext.

Als Induktionshypothese benutzen wir die Hypothese des Lemmas.

Induktionsbeginn: $(n, m = 0)$
s kann nur eine Variable, die Konstante `choice` oder eine Abstraktion sein, alle anderen Möglichkeiten ergeben einen Widerspruch zur Minimalität.
Ist s eine Variable so ist eine bei $L1.1$ oder $L2.1$ oder $L3.1$ startenden no-Entwicklungen möglich. Entspricht die no-Entwicklung derjenigen ausgehend von $L1.1$ oder $L2.1$ so erscheint bei $L1.2$ bzw. $L2.2$ bei der rechten no-Entwicklung ein Ausdruck der Form $L_R^*[r]$ mit r eine Abstraktion oder die Konstante `choice` . Damit ist die Hypothese für diese Fälle gezeigt, da sich $L_R^*[r]$ gemäß Satz 2.3.1 in WHNF befindet. Entspricht die no-Entwicklung derjenigen ausgehend von $L3.1$, so ist die Gültigkeit der Hypothese offensichtlich.

Entspricht s einer Abstraktion oder der Konstante `choice` , so ist eine bei $L1.2$ oder $L2.2$ startende no-Entwicklung möglich. In beiden Fälle wird nach Entfernung des Ausdrucks t direkt eine WHNF erhalten, womit die Hypothese auch für diese Fälle gezeigt ist.

Induktionsschritt: $(n, m \geq 0)$ und $n \neq 0$ oder $m \neq 0$
Wir betrachten zunächst alle Ausdrücke mit $m > 0$ und n beliebig. Die Situation bei der no-Entwicklung mit bzw. ohne t gibt das Paar $L1.3$, $R1.3$ wieder. Da bei $L1.4$ der Wert m um eins erniedrigt und n gleichgeblieben ist, können wir die Induktionsvoraussetzung anwenden und erhalten in Kombination mit den beiden no-Entwicklungen die Hypothese.

Wir betrachten nun alle Ausdrücke mit $n > 0$ und $m = 0$. s kann in diesem Fall nur einer Variable, der Konstante `choice` oder einer Abstraktion entsprechen. Die Annahme, daß s einem Let-Ausdruck entspricht, ergibt einer Widerspruch im Zusammenhang mit der Voraussetzung $m = 0$. Entspricht s einer Applikation, dann können wir wegen der Voraussetzung $m = 0$ einen A_L^{n+1}-Kontext bilden und diesen Fall auf einen anderen zurückführen.
Entspricht s einer Variablen, dann sind drei unterschiedliche Paare von no-Entwicklungen, nämlich ausgehend von $L1.1$, $R1.1$ oder $L2.1$, $R2.1$ oder $L3.1$, $R3.1$

möglich. In den beiden ersten Fällen kann bei $L1.4$, $R1.4$ bzw. $L2.4$, $R2.4$ die Induktionsvoraussetzung angewendet werden. In Kombination mit den paarweisen no-Entwicklungen folgt daraus in beiden Fällen die Hypothese. Beim letzten Fall ist die Gültigkeit der Hypothese offensichtlich, da bei beiden no-Entwicklungen eine VWHNF erreicht wird.
Entspricht s einer Abstraktion oder der Konstante `choice` , so ist eine paarweise no-Entwicklung ausgehend von $L1.2$, $R1.2$ bzw. $L2.2$, $R2.2$ möglich. Bei $L1.4$, $R1.4$ bzw. $L2.4$, $R2.4$ kann die Induktionsvoraussetzung angewendet, woraus in Kombination mit den no-Entwicklungen die Hypothese folgt. Im Falle der Konstante `choice` ist für den Links- und Rechtszweig der no-Reduktion dieselbe Argumentation anwendbar.
Die paarweisen no-Entwicklungen zeigen, daß durch Entfernung des Subausdrucks t keine (no, nd)-Reduktion hinzukommt. Da im Falle einer (no, nd)-Reduktion die Induktionsvoraussetzung beim Links- wie beim Rechtszweig angewendet werden kann, gilt wegen König's Lemma auch die allquantifizierte dritte Implikation der Hypothese. □

Das obige Lemma läßt die folgenden Schlußfolgerungen zu:

Korollar 5.1.2. *Für beliebige Ausdrücke s, t:*

1. $s\,t{\Downarrow}_{WHNF} \implies s{\Downarrow}_{WHNF}$

2. $s\,t{\not\Uparrow} \implies s{\not\Uparrow}$

3. $s{\not\Downarrow} \implies s\,t{\not\Downarrow}$

4. $s{\Uparrow} \implies s\,t{\Uparrow}$

Beweis. Die ersten beiden Implikationen werden bei Wahl des leeren Kontext $[\cdot]$ für L_R^* und $n = 0$ erhalten. Die beiden letzten Implikationen sind die logischen Umkehrungen der beiden ersten. □

Lemma 5.1.3. *Für alle Λ_{let}-Ausdrücke s gilt*

- $L_R^*[\texttt{let}\ x = A_L^n[s]\ \texttt{in}\ W[x]]{\Downarrow}_{WHNF} \Rightarrow L_R^*[A_L^n[s]]{\Downarrow}_{WHNF}$

- $L_R^*[\texttt{let}\ x = A_L^n[s]\ \texttt{in}\ W[x]]{\Downarrow}_{VWHNF} \Rightarrow L_R^*[A_L^n[s]]{\Downarrow}_{VWHNF \vee WHNF}$

- $L_R^*[\texttt{let}\ x = A_L^n[s]\ \texttt{in}\ W[x]]{\not\Uparrow} \Rightarrow L_R^*[A_L^n[s]]{\not\Uparrow}$

Beweis. Der Beweis ähnelt methodisch und inhaltlich stark dem des Lemmas zuvor, soll jedoch der Vollständigkeit halber komplett ausgearbeitet werden. Wir werden die folgenden 4 paarweisen no-Entwicklungen benötigen:

Entwicklung 1:

$$\begin{array}{llll}
L1.1 & L_R^*[\texttt{let } x = A_L^n[L_R[s]] \texttt{ in } W[x]] & R1.1 & L_R^*[A_L^n[L_R[s]]] \\
 & \xrightarrow{no,lapp,n} & & \xrightarrow{no,lapp,n} \\
L1.2 & L_R^*[\texttt{let } x = L_R[A_L^n[s]] \texttt{ in } W[x]] & R1.2 & L_R^*[L_R[A_L^n[s]]] \\
 & \xrightarrow{no,llet} & & \\
L1.3 & L_R^*[L_R[\texttt{let } x = A_L^n[s] \texttt{ in } W[x]]] & &
\end{array}$$

Entwicklung 2:

$$\begin{array}{llll}
L2.1 & L_R^*[\texttt{let } x = A_L^n[y] \texttt{ in } W[x]] & R2.1 & L_R^*[A_L^n[y]] \\
 & \xrightarrow{no,\omega} & & \xrightarrow{no,\omega} \\
L2.2 & L_R^*[\texttt{let } x = A_L^n[\lambda z.s] \texttt{ in } W[x]] & R2.2 & L_R^*[A_L^n[\lambda z.s]] \\
 & \xrightarrow{no,lbeta}(*) & & \xrightarrow{no,lbeta}(*) \\
L2.3 & L_R^*[\texttt{let } x = A_L^n[L_R[s]] \texttt{ in } W[x]] & R2.3 & L_R^*[A_L^n[L_R[s]]]
\end{array}$$

$(*)$ falls $n > 1$ sonst WHNF

Entwicklung 3:

$$\begin{array}{llll}
L3.1 & L_R^*[\texttt{let } x = A_L^n[y] \texttt{ in } W[x]] & R3.1 & L_R^*[A_L^n[y]] \\
 & \xrightarrow{no,\omega} & & \xrightarrow{no,\omega} \\
L3.2 & L_R^*[\texttt{let } x = A_L^n[\texttt{choice }] \texttt{ in } W[x]] & R3.2 & L_R^*[A_L^n[\texttt{choice }]] \\
 & \xrightarrow{no,nd,x}(*) & & \xrightarrow{no,nd,x} \\
L3.3 & L_R^*[\texttt{let } x = A_L^n[\lambda x.\lambda y.x] \texttt{ in } W[x]] & R3.3 & L_R^*[A_L^n[\lambda x.\lambda y.x]] \\
 & \xrightarrow{no,lbeta,lapp,lbeta}(**) & & \xrightarrow{no,lbeta,lapp,lbeta}(**) \\
L3.4 & L_R^*[\texttt{let } x = A_L^{n-2}[L_R^2[x]] \texttt{ in } W[x]] & R3.4 & L_R^*[A_L^{n-2}[L_R^2[x]]]
\end{array}$$

$(*)$ $x \in \{left, right\}$ die Entwicklung beider Zweige erfolgt simultan.

$(**)$ falls $n > 2$ sonst WHNF

Entwicklung 4:

$$\begin{array}{llll}
L4.1 & L_R^*[\texttt{let } x = A_L^n[y] \texttt{ in } W[x]] & R4.1 & L_R^*[A_L^n[y]] \\
 & \xrightarrow{no,\omega} & & \xrightarrow{no,\omega} \\
L4.2 & L_R^*[\texttt{let } y = W[z] \texttt{ in } L_R^*[\texttt{let } x = A_L^n[y] \texttt{ in } W[x]]] & R4.2 & L_R^*[\texttt{let } y = W[z] \texttt{ in } L_R^*[A_L^n[y]]] \\
 & \equiv & & \equiv
\end{array}$$

z komme frei vor, daher VWHNF auf beiden Seiten

Wir führen den Beweis durch Induktion über die Menge aller Ausdrücke der Form $L_R^*[\texttt{let } x = A_L^n[L_R^m[s]] \texttt{ in } W[x]]$, wobei $n, m \geq 0$ und s minimal, die wir mittels der folgenden Bewertung ordnen:

Das Bewertungsmaß besteht aus zwei Komponenten die lexikographisch geordnet sind. Die erste Komponente ist die Tiefe n des A_L^n-Kontexts, die zweite Komponente ist die Tiefe m des L_R^m-Kontext.

Als Induktionshypothese benutzen wir die Hypothese des Lemmas.

Induktionsbeginn: $(n, m = 0)$
s kann nur eine Variable, die Konstante `choice` oder eine Abstraktion sein, da alle anderen Möglichkeiten einen Widerspruch zur Minimalität ergeben.
Ist s eine Variable so ist eine der paarweisen no-Entwicklung 2,3,4 möglich. Bei der zweiten bzw. dritten Entwicklung wird bei $R2.2$ bzw $R3.2$ ein Ausdruck der Form $L_R^*[r]$ mit r eine Abstraktion oder die Konstante `choice` erreicht. Damit ist die Hypothese für diese Fälle gezeigt, da sich $L_R^*[r]$ gemäß Satz 2.3.1 in WHNF befindet. Bei der vierten no-Entwicklung ist die Gültigkeit der Hypothese offensichtlich, da bei $L4.2$ wie $R4.2$ eine VWHNF erreicht wird.
Entspricht s einer Abstraktion oder der Konstante `choice` , so liegt eine bei $L2.2$ und $R2.2$ bzw. $L3.2$ und $R3.2$ startend paarweise no-Entwicklung vor. Da sich $R2.2$ wie $R3.2$ in WHNF befindet, ist die Hypothese für beide Fälle gezeigt.

Induktionsschritt: $(n, m \geq 0)$ und $n \neq 0$ oder $m \neq 0$
Wir betrachten zunächst alle Ausdrücke mit $m > 0$ und n beliebig. Die Situation bei der no-Entwicklung gibt das Paar $L1.1$, $R1.1$ wieder. Da bei $L1.3$ der Wert m um eins erniedrigt und n gleichgeblieben ist, können wir die Induktionsvoraussetzung auf $L1.3$, $R1.2$ anwenden und erhalten in Kombination mit den beiden no-Entwicklungen die Hypothese.

Wir betrachten nun alle Ausdrücke mit $n > 0$ und $m = 0$. s kann in diesem Fall nur einer Variable, der Konstante `choice` oder einer Abstraktion entsprechen. Die Annahme, daß s einem Let-Ausdruck entspricht, ergibt einer Widerspruch im Zusammenhang mit der Voraussetzung $m = 0$. Entspricht s einer Applikation, dann können wir wegen der Voraussetzung $m = 0$ einen A_L^{n+1}-Kontext bilden und diesen Fall auf einen anderen zurückführen.
Entspricht s einer Variablen, dann sind drei unterschiedliche Paare von no-Entwicklungen, nämlich ausgehend von $L2.1$, $R2.1$ oder $L3.1$, $R3.1$ oder $L4.1$, $R4.1$ möglich. In den beiden ersten Fällen kann bei $L2.3$, $R2.3$ bzw. $L3.4$, $R3.4$ die Induktionsvoraussetzung angewendet werden. In Kombination mit den paarweisen no-Entwicklungen folgt daraus in beiden Fällen die Hypothese. Beim letzten Fall ist die Gültigkeit der Hypothese offensichtlich, da bei beiden no-Entwicklungen eine VWHNF erreicht wird.
Entspricht s einer Abstraktion oder der Konstante `choice` , so ist eine paarweise no-Entwicklung ausgehend von $L2.2$, $R2.2$ bzw. $L3.2$, $R3.2$ möglich. Bei $L2.3$, $R2.3$ bzw. $L3.4$, $R3.4$ kann die Induktionsvoraussetzung angewendet werden, woraus in Kombination mit den no-Entwicklungen die Hypothese folgt. Im Falle der Konstante `choice` ist für den Links- und Rechtszweig der no-Reduktion dieselbe Argumentation anwendbar.
Die paarweisen no-Entwicklungen zeigen, daß durch Entfernung des Subausdrucks t keine (no, nd)-Reduktion hinzukommt. Da im Falle einer (no, nd)-Reduktion die Induktionsvoraussetzung beim Links- wie beim Rechtszweig angewendet werden kann, gilt wegen König's Lemma auch die allquantifizierte dritte Implikation der

Hypothese. ☐

Das obige Lemma erlaubt die folgenden Schlußfolgerungen:

Korollar 5.1.3. *Für einen beliebigen Ausdruck t sowie W-Kontext gilt*

1. $\texttt{let}\ x = t\ \texttt{in}\ W[x]{\Downarrow}_{WHNF \vee VWHNF} \implies t{\Downarrow}_{WHNF \vee VWHNF}$
2. $\texttt{let}\ x = t\ \texttt{in}\ W[x]{\not\Uparrow} \implies t{\not\Uparrow}$
3. $t{\not\Downarrow} \implies \texttt{let}\ x = t\ \texttt{in}\ W[x]{\not\Downarrow}$
4. $t{\Uparrow} \implies \texttt{let}\ x = t\ \texttt{in}\ W[x]{\Uparrow}$

Beweis. Die ersten beiden Implikationen werden bei Wahl des leeren Kontext $[\cdot]$ für L_R^* und $n = 0$ erhalten. Die beiden letzten Implikationen sind die logischen Umkehrungen der beiden ersten. ☐

Die Aussagen der zuvor bewiesenen Korollare erlauben den Beweis des folgenden Lemmas:

Lemma 5.1.4. *Für alle Λ_{let}-Ausdrücke t und W-Kontexte gilt*

1. $W[t]{\Downarrow}_{WHNF \vee VWHNF} \implies t{\Downarrow}_{WHNF \vee VWHNF}$
2. $W[t]{\not\Uparrow} \implies t{\not\Uparrow}$
3. $t{\not\Downarrow} \implies W[t]{\not\Downarrow}$
4. $t{\Uparrow} \implies W[t]{\Uparrow}$

Beweis. Die zweite letzten Implikationen sind die logische Umkehrung der beiden ersten. Wir zeigen die Korrektheit der ersten Implikation für alle W-Kontexte:

nicht rekursiver W-Kontext

$$\begin{array}{lll} & L_R^*[A_L^*[t]]{\Downarrow}_{WHNF \vee VWHNF} & \\ \Rightarrow & A_L^*[t]{\Downarrow}_{WHNF \vee VWHNF} & \text{Korollar 5.1.1} \\ \Rightarrow & t{\Downarrow}_{WHNF \vee VWHNF} & \text{Korollar 5.1.2,*} \end{array}$$

rekursiver W-Kontext

$$\begin{array}{lll} & L_R^*[\texttt{let}\ x = A_L^*[t]\ \texttt{in}\ W[x]]{\Downarrow}_{WHNF \vee VWHNF} & \\ \Rightarrow & \texttt{let}\ x = A_L^*[t]\ \texttt{in}\ W[x]{\Downarrow}_{WHNF \vee VWHNF} & \text{Korollar 5.1.1} \\ \Rightarrow & A_L^*[t]{\Downarrow}_{WHNF \vee VWHNF} & \text{Korollar 5.1.3} \\ \Rightarrow & t{\Downarrow}_{WHNF \vee VWHNF} & \text{Korollar 5.1.2,*} \end{array}$$

Genauso wird die Korrektheit der zweiten Implikation gezeigt. ☐

Nun sind wir in der Lage zu beweisen, daß alle Ausdrücke ohne WHNF bzw. VWHNF bezüglich der Präordnung $\leq_c$ kleinste Elemente bilden.

Satz 5.1.1. *Für alle Λ_{let}-Ausdrücke s, t gilt: $s\not\Downarrow \implies s \leq_c t$*

Beweis. Wegen des Kontextlemmas 3.2.2 reicht es zu zeigen, daß $s \leq_W t$ gilt. Aus $s\not\Downarrow$ folgt gemäß Lemma 5.1.4 $W[s]\not\Downarrow$ für alle Kontexte W. Aus $W[s]\not\Downarrow$ für alle Kontexte W folgt, daß $W[s]\Downarrow \Rightarrow W[t]\Downarrow$ für beliebige t und Kontexte W wahr ist. Genauso gilt die andere Richtung $W[t]\Uparrow \Rightarrow W[s]\Uparrow$, wegen $W[s]\not\Downarrow \Rightarrow W[s]\Uparrow$. □

Korollar 5.1.4. *s, t seien zwei Λ_{let}-Ausdrücke mit $s\not\Downarrow$ und $t\not\Downarrow$. Dann gilt $s \sim_c t$.*

Notation 5.1.1. *Die Äquivalenzklasse aller kleinsten Ausdrücke bezüglich $\leq_c$ wird mit dem Symbol $\bot$ bezeichnet.*

Die Existenz der Klasse aller $\bot$-Ausdrücke ist eine Voraussetzung für die nun folgende Betrachtung von Rekursion.

5.2 Rekursion und Fixpunktkombinatoren

In diesem Abschnitt soll die Fragestellung der Rekursion beim Λ_{let}-Kalkül untersucht werden. Der Λ_{let}-Kalkül verfügt über keine rekursiven Let-Konstrukte und erlaubt daher keine Spezifikation von rekursiven Strukturen auf syntaktischer Ebene. Nach einem kurzen Einschub mit grundlegenden Definitionen wird zunächst eine informale Einführung in Rekursion beim Λ_{let}-Kalkül gegeben. Rekursion wird dabei unter Zuhilfenahme der kontextuelle Präordnung $\leq_c$ erschlossen. Im Anschluß werden wir Fixpunktkombinatoren kennenlernen.

Definition 5.2.1. *$S = s_1 \leq_c s_2 \leq_c \ldots$ sei eine potentiell unendliche aufsteigende Folge von Λ_{let}-Ausdrücken. Wir definieren:*

- *t ist eine obere Schranke von S, gdw. für alle $s_i \in S : s_i \leq_c t$.*
- *t ist eine kleinste obere Schranke von S, bezeichnet durch $t = lub(S)$, gdw. $\forall$ oberen Schranken u von $S : t \leq_c u$.*
- *t ist eine kleinste kontextuelle obere Schranke von S, bezeichnet durch $t = club(S)$, gdw. $\forall$ Kontexte $C : lub(C[S]) = C[t]$. Dabei entspricht $C[S]$ der aufsteigenden Folge $C[s_1] \leq_c C[s_2] \leq_c \ldots$.*

Eine wichtige Differenz gegenüber der klassischen Modellbildung beim Lambda-Kalkül auf Basis einer cpo ist, daß die kleinste obere Schranke hier kein eindeutiges semantisches Objekt ist. So können mehrere Ausdrücke die kleinste obere Schranke einer aufsteigenden Folge von Ausdrücken bilden. Es gilt jedoch, daß die *lub* bezüglich der kontextuellen Äquivalenz eindeutig ist, d.h. bilden zwei Ausdrücke t und t' die *lub* einer aufsteigenden Folge von Ausdrücken, so gilt $t \sim_c t'$. Somit kann für einen Ausdruck t und eine bezüglich $\leq_c$ aufsteigende Folge von Ausdrücken S gezeigt werden, daß $t = lub(S)$ gilt, indem $t \sim_c lub(S)$ gezeigt wird. Genau dasselbe gilt auch für die *club*. Dieser Rückgriff auf $\sim_c$ wird später beim Beweis von Satz 5.2.2 Verwendung finden.
Es ist ferner offensichtlich, daß jede *club* eine *lub* ist. Jedoch ist die Umkehrung der letzteren Aussage eine offene Fragestellung, die in dieser Arbeit nicht beantwortet werden wird.

Eine bleibende Fragestellung ist, warum neben der *lub* noch die *club* eingeführt wurde. Wir werden dazu auf eine Schwäche aufmerksam machen, die die *lub*-Definition beinhaltet. Angenommen s ist die *lub* einer aufsteigenden Folge von Ausdrücken $F = s_1 \leq_c s_2 \leq_c s_3 \leq_c \cdots$. Wir bilden nun die Folge $F' = s_1 t \leq_c s_2 t \leq_c s_3 t \leq_c \cdots$, die erhalten wird, indem jeder Ausdruck s_i auf einen Ausdruck t angewendet wird. Selbstverständlich ist die Folge F' wegen der Präkongruenz von $\leq_c$ aufsteigend und st eine obere Schranke von t, da $s_i t \leq_c st$ für alle i gilt. Jedoch ist nicht klar, ob st auch die kleinste obere Schranke, d.h. eine *lub* von F' ist. Beispielsweise könnte gelten, daß alle s_i die Eigenschaft $s_i \not\sim_c \bot$ und alle $s_i t$ die $s_i t \sim_c \bot$ Eigenschaft erfüllen. Daraus folgt zum einen, daß $s \not\sim_c \bot$ gilt, und zum anderen, daß die *lub* von F' $\bot$ wäre. Wir können jedoch aus der *lub*-Definition heraus nicht folgern, daß $st \sim_c \bot$ gilt. Die Ursache dieser Schwäche ist, daß die *lub*-Definition nicht die Eigenschaft

$$lub(C[S]) = C[lub(S)]$$

impliziert, wobei S eine bezüglich $\leq_c$ aufsteigende Folge von Ausdrücken sei. Die *club*-Definition ist so gewählt, daß die obige Eigenschaft zusätzlich erfüllt wird. Mit anderen Worten gesprochen, sind die Kontexte bezüglich der *club* stetig.
Auf die oben beschriebene Schwäche der *lub* wird am Ende dieses Abschnitts noch einmal eingegangen.

Notation 5.2.1. *s, t seien zwei beliebige Λ_{let}-Ausdrücke. Mit $s^i(t)$ kürzen wir die i-fache rekursive Schachtelung der Form* $\underbrace{s(s \ldots (s}_{i \text{ mal}}\ t) \ldots)$ *ab.*

Wird die Schreibweise $s^i(t)$ in Verbindung mit dem Wort Folge benutzt, so wird die unendliche Folge t, $s(t)$, $s^2(t)$, $s^3(t), \ldots$ bezeichnet.

Lemma 5.2.1. *Für jeden Λ_{let}-Ausdruck t gilt $\bot \leq_c t(\bot) \leq_c t^2(\bot) \leq_c t^3(\bot) \ldots$.*

Beweis. Gemäß Satz 5.1.1 gilt $\perp \leq_c t(\perp)$ für alle Ausdrücke t. (Das Symbol $\perp$ ist wie vereinbart Stellvertreter für einen beliebige Ausdruck s mit $s \not\Downarrow$). Die Präkongruenz von $\leq_c$ liefert $\perp \leq_c t(\perp) \;\Rightarrow\; t^i(\perp) \leq_c t^i(t(\perp))$ für alle $i \in \mathbb{N}$. Die Hypothese ergibt sich dann durch eine einfache transitive Entwicklung. □

Es sei an dieser Stelle explizit darauf hingewiesen, daß die obige aufsteigende Folge nicht entstehen braucht, wenn statt $\perp$ ein anderer Ausdruck als Startpunkt gewählt wird. Dies ist eine grundlegende Beobachtung im Bezug auf die nun folgende Einführung in Rekursion beim Λ_{let}-Kalkül.

5.2.1 Rekursion im Λ_{let}-Kalkül - eine Einführung

Am Beispiel der Fakultätsfunktion soll nun aufgezeigt werden, wie rekursive Funktionen im Λ_{let}-Kalkül als nicht rekursive Ausdrücke dargestellt werden können.
Im weiteren gehen wir davon aus, der Λ_{let}-Kalkül sei um Church-Zahlen, die mathematischen Operationen Multiplikation und Subtraktion, sowie eine Kombinator `ifzero` erweitert. `ifzero` entspreche einem Test auf Null und gehorche folgender Reduktionsvorschrift:

$$\texttt{ifzero}\; c\, s\, t \rightarrow \begin{cases} s & \text{falls } c \text{ dem Wert 0 entspricht} \\ t & \text{sonst} \end{cases}$$

Wie derartige Erweiterungen im Lambda-Kalkül eingebettet werden können, beschreibt [Bar84]. Die dort vorgestellten Methodiken sind prinzipiell auf den Λ_{let}-Kalkül übertragbar. Aus Gründen der Vereinfachung trennen wir im weiteren nicht explizit zwischen Zahlen und Ausdrücken, die Zahlwerte repräsentieren. Zudem bezeichnen wir die no-Reduktion, die mit den zusätzlichen Konstrukten umgehen kann, als noa-Reduktion.
Unser Ziel sei nun, die Fakultätsfunktion zu modellieren. Wir definieren uns dazu einen Ausdruck f wie folgt:

$$f \equiv_{def} \lambda f'.\lambda n.\texttt{ifzero}\; n\; 1\; (*\; n\; f'(-\; n\; 1))$$

Wir betrachten nun das Reduzierungsverhalten der $f^i(\perp)$ für alle $i > 1$, angewendet auf alle natürlichen Zahlen:

$$\begin{array}{llll}
(f^1\,(\perp))\,0 \xrightarrow{noa,*} 1 & & & \text{für } n > 0 \text{ gilt } f^1\,(\perp) = \perp \\
(f^2\,(\perp))\,0 \xrightarrow{noa,*} 1 & (f^2\,(\perp))\,1 \xrightarrow{noa,*} 1 & & \text{für } n > 1 \text{ gilt } f^2\,(\perp) = \perp \\
(f^3\,(\perp))\,0 \xrightarrow{noa,*} 1 & (f^3\,(\perp))\,1 \xrightarrow{noa,*} 1 & (f^3\,(\perp))\,2 \xrightarrow{noa,*} 2 & \text{für } n > 2 \text{ gilt } f^3\,(\perp) = \perp \\
\vdots & \vdots & \vdots & \vdots
\end{array}$$

Offensichtlich wird durch eine immer tiefere Schachtelung des Ausdrucks f eine immer bessere Approximation der Fakultätsfunktion erzielt. Eine andere Sichtweise erlaubt die Aussage, daß die Tiefe der Schachtelung in Korrelation mit der Anzahl rekursiver Aufrufe der Fakultät steht. Die Spezifikation eines Ausdrucks, der das Abbildungsverhalten der Fakultätsfunktion total wiederspiegelt, ist aber offenbar mittels der $f^i(\bot)$ nicht möglich, da der dazu erforderliche Ausdruck nicht endlich spezifiziert wäre. Jedoch scheint ein Ausdruck, der das Abbildungsverhalten aller $f^i(\bot)$ approximiert, zugleich eine Approximation der Fakultätsfunktion zu sein. Wir werden nun sehen, wie die Folge $f^i(\bot)$ selbst dazu benutzt werden kann, um die Eigenschaften eines Ausdrucks zu beschreiben, der eine möglichst genaue Approximation aller $f^i(\bot)$ darstellt. Dazu ordnen wir die Menge aller Λ_{let}-Ausdrücke mit Zahlen und der `ifzero`-Konstante wie folgt:

Wir bilden eine Bewertung

$$N(t) := \{n \in \mathbb{N} \ : \ t\,n \xrightarrow{noa,*} n' \ \wedge \ n' \text{ ist eine Zahl mit } n' = fac(n)\},$$

die jedem Ausdruck eine Menge natürlicher Zahlen zuordnet, dabei sei fac die Funktion, die die Fakultät berechnet. Wir vereinbaren:

$$t \leq t' \text{ gdw. } N(t) \subseteq N(t').$$

Die zuletzt definierte Ordnung auf der Menge der Λ_{let}-Ausdrücke besitzt folgende Eigenschaften:

- Die Ausdrücke der unendlichen Folge $f^i(\bot)$ werden durch die definierte Ordnung aufsteigend sortiert.
- Die Ausdrücke werden nach ihrem Verhalten im Bezug auf ihre Argumente und damit nach ihrem Verhalten im Bezug auf ihren umgebenden Kontext bewertet.
- Ein Ausdruck ist um so größer, für je mehr Argumentwerte die Reduzierung der Berechnung der Fakultät entspricht.

Zugleich ermöglicht die zuletzt definierten Ordnung in Kombination mit der Folge $f^i(\bot)$ folgende Charakterisierung:

- Ein Ausdruck t approximiert das Abbildungsverhalten aller $f^i(\bot)$ sehr genau, wenn alle $f^i(\bot)$ kleiner als t sind und kein Ausdruck t' existiert, der kleiner als t und größer als alle $f^i(\bot)$ ist.

Wir werden uns nun von dem Beispiel lösen und zum allgemeinen Fall übergehen. Dabei entstehen folgende Problemstellungen:

- Für einen beliebigen Ausdruck t haben wir keine Vorstellung davon, welche Abbildung sich hinter der Folge $t^i(\bot)$ verbirgt.
- Wir benötigen eine Ordnung, die für alle t die Ausdrücke der Folge $t^i(\bot)$ aufsteigend ordnet, um die obige Approximationsvorstellung beibehalten zu können.

Eine Lösung stellt die Wahl der Präkongruenz $\leq_c$ als Ordnung der Ausdrücke dar. Wir haben in Satz 5.1.1 gezeigt, daß $\bot$ bezüglich $\leq_c$ die schwächste Ausdrucksklasse darstellt. Daraus folgt direkt $s(\bot) \geq_c \bot$ und wegen der Präkongruenz von $\leq_c$ auch $s^i(\bot) \geq_c s^j(\bot)$ für alle Ausdrücke s mit $i \geq j$. Somit erfüllt $\leq_c$ die geforderten Eigenschaften (siehe auch Lemma 5.2.1). Anhand einer Gegenüberstellung der Ordnung aus dem Fakultätsbeispiel und der zuletzt definierten, soll gezeigt werden, warum die Wahl von $\leq_c$ auch intuitiv gesehen korrekt ist:

- Desto größer der Index i gewählt wurde, desto weniger Nichtterminierung bezogen auf alle möglichen Argumente entstand bei den $f^i(\bot)$ aus obigem Beispiel. Ähnliches gilt bei $\leq_c$. Desto größer der Index i bei einer Folge $t^i(\bot)$ gewählt wird, desto weniger Nichtterminierung bezogen auf alle umgebenden Kontexte entsteht.

- Durch einen immer größeren Index i wurde das in dem fiktiven Ausdruck $f^\infty(\bot)$ gekapselte Verhalten der Fakultät immer besser approximiert. Ähnliches gilt bei $\leq_c$. Durch einen immer größeren Index i wird das in dem fiktiven Ausdruck $t^\infty(\bot)$ gekapselte Terminierungsverhalten immer besser approximiert.

Damit sind wir in der Lage auch den Approximationsgedanken zu übertragen:

> Ein Ausdruck s approximiert das Terminierungsverhalten aller $t^i(\bot)$ sehr genau, wenn alle $t^i(\bot)$ bezüglich $\leq_c$ kleiner als s sind und kein Ausdruck s' existiert, der kleiner als s und größer als alle $t^i(\bot)$ ist.

Formal entspricht dies $s = lub(t^i(\bot))$. Somit können wir s auch als kleinste obere Schranke bezeichnen. In dem folgenden Abschnitt werden wir einen Λ_{let}-Ausdruck kennenlernen, der in Kombination mit Abstraktionen sowie der Konstante `choice` geeignet ist, solch eine kleinste obere Schranke zu bilden.

5.2.2 Fixpunktkombinatoren

Zum Zweck der Entwicklung der oben beschriebenen kleinsten oberen Schranke führen wir den Begriff des Fixpunktkombinators ein.

Definition 5.2.2. *Ein Fixpunktkombinator eines Ausdrucks t ist ein Ausdruck Y, der die Eigenschaft $Y\,t \sim_c t(Y\,t)$ besitzt.*

Der Begriff Fixpunktkombinator resultiert aus der Eigenschaft des Ausdrucks Y einen Kombinator zu bilden sowie der Eigenschaft von $Y\,t$, einen Fixpunkt des

Ausdrucks t bezüglich der kontextuellen Äquivalenz zu bilden. Es sei noch einmal explizit erwähnt, daß die Definition des Fixpunktkombinators hier an der kontextuellen Äquivalenz verankert ist, im klassischen Lambda-Kalkül geschieht diese Verankerung üblicherweise an der Kongruenzrelation $=_\beta$ (siehe [HP86], Kapitel 3). Auf eine Darstellung der operationalen Funktionsweise von Fixpunktkombinatoren soll hier verzichtet und auf die Literatur verwiesen werden. In [PJ87] werden die operationalen Aspekte der Realisation von Rekursion mit Hilfe eines Fixpunktkombinators anhand der Fakultätsfunktion ausführlich erläutert.

Das folgende Lemma stellt nun eine Verbindung zwischen dem Fixpunkt eines Ausdrucks s bezüglich $\sim_c$ und der im letzten Kapitel vorgestellten oberen Schranken der Folge $s^i(\bot)$ her.

Lemma 5.2.2. *s sei ein Ausdruck und u ein Fixpunkt von s, d.h. es gilt $s\,u \sim_c u$. Dann gilt $C[s^i(\bot)] \leq_c C[u]$ für alle Kontexte C und $i \in \mathbb{N}$.*

Beweis. Aus $s\,u \sim_c u$ kann durch einfache transitive Entwicklung $s^i(u) \sim_c u$ erhalten werden. Da $\sim_c$ eine Kongruenzrelation ist, gilt $C[s^i(u)] \sim_c C[u]$ für alle Kontexte C und damit insbesondere $C[s^i(u)] \leq_c C[u]$. Da $\leq_c$ eine Präkongruenzrelation ist und $\bot$ einem kleinsten Element bezüglich $\leq_c$ entspricht, gilt $C[s^i(\bot)] \leq_c C[s^i(u)]$ für alle $i \in \mathbb{N}$ und Kontexte C. Zusammengesetzt ergibt sich daraus die Hypothese. □

Für jeden Fixpunktkombinator Y eines Ausdrucks s gilt somit, daß $Y\,s$ eine obere Schranke der Folge $s^i(\bot)$ bildet, und diese Aussage auf alle Kontexte erweiterbar ist. Nun stellt sich die Frage nach der Existenz des Fixpunktkombinators. Ferner stellt sich die Frage, ob ein Fixpunktkombinator existiert, der die kleinste obere Schranke liefert.

Wir werden im Anschluß einen Fixpunktkombinator kennenlernen, der im Fall einer Abstraktion bzw. der Konstante `choice` einen Fixpunkt liefert, und zeigen, daß die gebildeten Fixpunkte der gesuchten *club* entsprechen.

Definition 5.2.3. *Wir definieren folgende Kombinatoren:*

$$\begin{aligned} Z &\equiv \lambda z.\lambda x.x(zzx) \\ Y_T &\equiv ZZ \end{aligned}$$

Y_T ist ein von Turing entwickelter Superkombinator, der im klassischen Lambda-Kalkül einen kleinsten Fixpunktkombinator auf der Kongruenzrelation $=_\beta$ bildet. Zur weiteren Untersuchung von Y_T werden wir ein spezielle Variante der no-Reduktion benötigen, die nun vorgestellt werden soll.

$$\begin{tikzcd}
\circ \arrow[r, "{no, lbeta}"] \arrow[ddrr, "{no, lbs}"'] & \circ \arrow[r, dashed, "{no, cp, *}"] & \circ \arrow[r, "{no, *}"] \arrow[d, dashed, "{i, cppar, 0/1}"] & t_{WHNF} \arrow[dd, dashed, "{(i, cppar), ldel, *}"] \\
 & & \circ \arrow[d, dashed, "ldel"] & \\
 & & \circ \arrow[r, dashed, "{no, *}"] & t'_{WHNF}
\end{tikzcd}$$

Abbildung 5.1:

5.2.3 no$_\beta$-Reduktionen

Definition 5.2.4. *Wir definieren die Reduktion* $\xrightarrow{lbs}$ *als:*

$$C[(\lambda x.t)r] \xrightarrow{lbs} C[t[r/x]]$$

wobei r *eine Abstraktion oder die Konstante* `choice` *.*

Eine no$_\beta$-Reduktion ist eine no-Reduktion, bei der jede $(no, lbeta)$*-Reduktion durch eine lbs-Reduktion ersetzt wird, sofern dies zulässig ist, d.h. der lbeta-Redex als rechten Subausdruck eine Abstraktion oder die Konstante* `choice` *besitzt.*

lbs-Reduktionen entsprechen β-Reduktionen des klassischen Lambda-Kalküls mit der Einschränkung, daß der zu substituierende Ausdruck einer Abstraktion oder der Konstante `choice` entspricht. Wir werden nun zeigen, daß die Begriffe no$_\beta$-Reduktion und no-Reduktion austauschbar sind.

Lemma 5.2.3. *s sei ein beliebiger Λ_{let}-Ausdruck. Folgende Aussagen gelten:*

1. *s besitzt eine no-Reduktion zu WHNF gdw. s eine no$_\beta$-Reduktion zu WHNF besitzt.*
2. *s besitzt eine unendliche no-Reduktion gdw. s eine unendliche no$_\beta$-Reduktion besitzt.*

Beweis. Jede (no, lbs)-Reduktion kann wie folgt nachgebildet werden:

$$(no, lbeta) \circ (no, cp)^n \circ (i, cppar)^{0/1} \circ ldel \quad \text{wobei } n \geq 0$$

$\Rightarrow$

Wir zeigen durch Induktion über die Anzahl der $(no, lbeta)$-Reduktionen und mit Hilfe der Gabeldiagramme der *cppar*- sowie *ldel*-Reduktion, daß der Austausch von $(no, lbeta)$-Reduktionen durch *lbs*-Reduktionen nichts an der Reduktion zu WHNF ändert. Dabei nutzen wir die Eigenschaft einer *cppar*- sowie *ldel*-Reduktion aus, die Anzahl der $(no, lbeta)$-Reduktionen nicht zu verändern. Das bei jedem Induktionsschritt entstehende Schema zeigt Abbildung 5.1. Die Übertragbarkeit einer unendlichen no-Reduktion zeigen wir wie in früheren Abschnitten gezeigt durch Zählung der $(no, lbeta)$-Reduktionen und Anwendung von König's Lemma.

$\Leftarrow$

Sei r die Reduktion, die erhalten werde, indem jede (no, lbs)-Reduktion der no_β-Reduktion gemäß der obigen Nachbildung ersetzt wurde. Wie in Abschnitt 2.4.4 gezeigt, können $(i, cppar)$- und *ldel*-Reduktionen durch Vertauschen stets an das Ende der Reduktion r bewegt werden. Ferner können *cp*- und *ldel*-Reduktionen einen Ausdruck nicht aus einer WHNF herausführen. Die Bewegung der $(i, cppar)$- und *ldel*-Reduktionen verändert die Zahl der $(no, lbeta)$-Reduktionen nicht. Damit bleibt von jeder (no, lbs)-Reduktion stets eine $(no, lbeta)$-Reduktion erhalten. Durch Zählung der $(no, lbeta)$-Reduktionen und Anwendung von König's Lemma kann somit die Übertragbarkeit einer unendlich no_β-Reduktion gezeigt. □

Das Einführen der no_β-Reduktion erbringt beweistechnische Vorteile. Bei einer no_β-Reduktion werden durch die Reduktion des Y_T-Kombinators keine Let-Bindungen eingeführt. Stattdessen zeigt der Kombinator in zwei no_β-Reduktionen direkt das gewünschte Verhalten.

5.2.4 *club*-Beweis für Y_T

Wir zeigen zunächst, daß Y_T einen Fixpunktkombinator bildet.

Satz 5.2.1. *r sei eine (offene oder geschlossene) Abstraktion oder die Konstante* `choice` *. Dann gilt $(Y_T\, r) \sim_c r\, (Y_T\, r)$.*

Beweis. Dies zeigt folgende einfache Entwicklung:

$$\begin{array}{ll}
 & (ZZ)r \\
\equiv & (\lambda z\, x.x(zzx)Z)r \\
\xrightarrow{lbeta} & (\texttt{let}\ z = Z\ \texttt{in}\ \lambda x.x(zzx))r \\
\xrightarrow{lapp} & \texttt{let}\ z = Z\ \texttt{in}\ (\lambda x.x(zzx))r \\
\xrightarrow{lbeta} & \texttt{let}\ z = Z\ \texttt{in}\ \texttt{let}\ x = r\ \texttt{in}\ x(zzx) \\
\xrightarrow{cp,4} & \texttt{let}\ z = Z\ \texttt{in}\ \texttt{let}\ x = r\ \texttt{in}\ r(ZZr) \\
\xrightarrow{ldel,2} & r(ZZr)
\end{array}$$

Alle benutzen Reduktionen erhalten gemäß Lemma 3.3.2 die kontextuelle Äquivalenz. □

Die obige Entwicklung gilt nur, wenn r eine Abstraktion oder die Konstante `choice` ist, ansonsten ist die Kopierbarkeit mittels der *cp*-Reduktionen nicht gegeben.
Der obige Satz zeigt in Verbindung mit Lemma 5.2.2, daß Y_T eine obere Schranke bildet. Wir werden nun zusätzlich zeigen, daß Y_T eine kleinste obere Schranke bildet.

Satz 5.2.2. *r sei eine (offene oder geschlossene) Abstraktion oder die Konstante* `choice` *. Dann gilt $(Y_T\, r) = club(r^i(\bot))$.*

Beweis. Satz 5.2.1 zeigt, daß $Y_T\, r$ ein Fixpunkt von r bildet, d.h. $Y_T\, r \sim_c r(Y_T\, r)$ gilt. In Kombination mit Lemma 5.2.2 folgt daraus $C[r^i(\bot)] \leq_c C[Y_T\, r]$ für alle $i \in \mathbb{N}$ und Kontexte C. Daraus folgt wiederum gemäß der Definition der *club*, daß $club(r^i(\bot)) \leq_c Y_T\, r$ gilt.

Wir zeigen nun umgekehrt $Y_T\, r \ \leq_c \ club(r^i(\bot))$. Dazu beweisen wir zunächst folgende zwei Implikationen:

1. für alle Kontexte R mit $R[Y_T\, r]$ geschlossen gilt: falls $R[Y_T\, r]\Downarrow$, dann existiert eine Zahl n, so daß für alle $i > n$ $R[r^i(\bot)]\Downarrow$ gilt.
2. für alle Kontexte R mit $R[Y_T\, r]$ geschlossen gilt: falls eine Zahl n existiert, so daß für alle $i > n$ $R[r^i(\bot)]\Uparrow$, dann gilt auch $R[Y_T\, r]\Uparrow$.

Aus den letzteren beiden Implikationen folgt $Y_T\, r \leq_R club(r^i(\bot))$. Daraus wiederum folgt wegen des Kontextlemmas 3.2.2 $Y_T\, r \leq_c club(r^i(\bot))$. Wir beweisen nun die beiden obigen Implikationen:

1. Konvergenz
Wir zeigen durch Induktion über die Anzahl der no$_\beta$-Reduktionen bis WHNF

$$R[(ZZ)r]\Downarrow^n \ \Rightarrow \ \forall i > n : \ R[r^i(\bot)]\Downarrow$$

Es sind folgende drei Situationen zu unterscheiden:

1) $R[(ZZ)r]$ befindet sich in WHNF. $R[(ZZ)r]$ hat dann die Form $L_R^*[r']$ mit r' eine Abstraktion oder die Konstante `choice` . Dann muß $R[r^i(\bot)]$ für alle $i \in \mathbb{N}$ die Form $L_R^*[r']$ mit geändertem L_R^*-Kontext besitzen, woraus folgt, daß sich $R[r^i(\bot)]$ in WHNF befindet.

2) Der no-Redex befindet sich innerhalb von R. Zwar kann $(ZZ)r$ einen Teil des no-Redex bilden jedoch kann es nicht kopiert werden, da es sich dazu unterhalb einer Abstraktion befinden müßte. Diese Möglichkeit wird jedoch durch den R-Kontext ausgeschlossen. Wir wenden auf beiden Seiten einen no$_\beta$-Schritt an und verwenden anschließend die Induktionsvoraussetzung.

3) Als dritte Möglichkeit bleibt, daß $(ZZ)r$ selbst den no-Redex bildet. In diesem Fall erhalten wir folgende no$_\beta$-Entwicklung:

$$\begin{array}{ll} & W[(ZZ)r] \\ \equiv & W[(\lambda z\, x.x(zzx)Z)r] \\ \xrightarrow{no,lbs} & W[(\lambda x.x(ZZx))r] \\ \xrightarrow{no,lbs} & W[r(ZZr)] \end{array}$$

Die no$_\beta$-Reduktion zu WHNF ausgehend von $W[r(ZZr)]$ ist um zwei Reduktionen verkürzt, d.h. $W[r(ZZr)]{\Downarrow}^{n-2}$. Wir zerlegen $W[r^i(\bot)]$ in $W[r(r^{i-1}(\bot))]$ und wenden anschließend die Induktionsvoraussetzung an.

2. Divergenz
Die Übertragbarkeit der Divergenz zeigen wir mittels der logischen Umkehrung

$$R[(ZZ)r]{\not\Uparrow} \;\Rightarrow\; \forall i > n : R[r^i(\bot)]{\not\Uparrow}$$

wobei n die Anzahl einzelner no$_\beta$-Reduktionen sei, die der längste no$_\beta$-Pfad ausgehend von $R[(ZZ)r]$ zu WHNF besitzt. Wir führen eine Induktion über die Anzahl der no$_\beta$-Reduktionen bis zum Erreichen einer WHNF auf dem längsten no$_\beta$-Pfad ausgehend von $R[(ZZ)r]$. Außer im Falle einer (no, nd)-Reduktion benutzen wir die bei der zuvor gezeigten Konvergenz verwendete Argumentation. Beim Erreichen einer (no, nd)-Reduktion wenden wir die Induktionsvoraussetzung auf beide Alternativen simultan an und benutzen anschließend König's Lemma, um auf die Endlichkeit aller no$_\beta$-Pfade ausgehend von $R[r^i(\bot)]$ zu schließen. □

5.2.5 *lub* versus *club*

Wie bereits angekündigt, sollen die Schwächen der *lub* gegenüber der *club* am Ende dieses Abschnitts noch einmal verdeutlicht werden. Dazu betrachten wir, welche Einschränkungen die Folge gewesen wären, wenn Satz 5.2.2 alleinig für die *lub* jedoch nicht für die *club* gezeigt worden wäre.
t sei ein Ausdruck für den sowohl eine Zerlegung in $C[Y_T\, s]$ wie auch in $D[Y_T\, r]$ existiere, wobei C und D unterschiedliche Kontexte seien. Somit kommt in t an zwei Stellen eine Rekursion vor, dargestellt mittels des Fixpunktkombinators Y_T. Wir wissen aus Satz 5.2.2, daß $Y_T\, s = club(s^i(\bot))$ und $Y_T\, r = club(r^i(\bot))$ gilt. Nun erfolgt eine entscheidende Ausdehnung, die im Falle der *lub* **nicht** möglich wäre. Folgende zwei Implikationen sind wegen der Definition der *club* zulässig:

- $Y_T\, s = club(s^i(\bot)) \quad\Rightarrow\quad C[Y_T\, s] = club(C[s^i(\bot)])$
- $Y_T\, r = club(r^i(\bot)) \quad\Rightarrow\quad D[Y_T\, r] = club(D[r^i(\bot)])$

Somit ist t eine *club* zweier aufsteigender Folgen von Ausdrücken gleichzeitig, wie wir es auch erwarten würden. Hätten wir Satz 5.2.2 alleinig für die *lub* gezeigt, wäre diese Ausdehnung nicht möglich gewesen, d.h. all unsere Beweise hätten nur einen sehr beschränkten Wert besessen, da wir das gleichzeitige mehrfache Vorkommen von Rekursion in einem Ausdruck nicht betrachtet hätten.

5.3 Abschließende Bemerkungen

Dieser Abschnitt ist weit davon entfernt, eine vollständige Betrachtung der Thematik Rekursion und Fixpunkte beim Λ_{let}-Kalkül zu geben. Eine Vielzahl von Fragestellungen bleibt unbeantwortet. Eine Auswahl dieser Fragestellungen lautet:

- Existiert ein Fixpunktkombinator, der für alle Λ_{let}-Ausdrücke einen Fixpunkt bildet? Im klassischen Lambda-Kalkül existieren solche Fixpunktkombinatoren, beim Λ_{let}-Kalkül scheint die Frage jedoch mit nein beantwortet werden zu müssen. Ein Prototyp des zugehörigen Beweises wurde entwickelt, jedoch nicht in die vorliegende Arbeit aufgenommen.
- Wie wird "verschränkte Rekursion" im Λ_{let}-Kalkül repräsentiert? Unter verschränkter Rekursion verstehen wir die Situation bei einander wechselseitig referenzierenden Subausdrücken eines `letrec`-Ausdrucks:

  $$\texttt{letrec}\ x = C[y],\ y = D[x]\ \texttt{in}\ \ldots$$

  Um diese Frage zu beantworten, ist es erforderlich, etwas vergleichbares zu dem "multiple fixed point theorem" (6.5.2 in [Bar84]) zu entwickeln.
- Wie sieht eine cpo (complete partial order) aus, die geeignet ist, eine denotationale Semantik im klassischen Sinne zu entwickeln, d.h. ein Modell für den Λ_{let}-Kalkül anzugeben? Diese Fragestellung scheint eine besonders schwierige Problemstellung zu bilden, zumindestens hat die im Kontext dieser Arbeit erfolgte Forschung in diesem Bereich trotz hohen Zeitaufwands nur "frustrierende Ergebnisse" geliefert.

All diese Fragestellungen seien an die weitere Forschung übergeben.

Kapitel 6

Maschinelle Nachahmung der no-Reduktion des Λ_{let}-Kalküls

Zur Ausführung eines funktionalen Programms wird nach einem Übersetzungsprozess üblicherweise auf eine abstrakte Maschine zurückgegriffen. Ein bekanntes Beispiel für eine solche abstrakte Maschine ist die G-Maschine, deren Arbeitsweise in [PJ87] vorgestellt wird.
Alle Ausführungen über den Λ_{let}-Kalkül in den letzten Abschnitten ließen die Frage offen, wie die no-Reduktion des Λ_{let}-Kalküls maschinell nachgeahmt werden kann. In diesem Abschnitt wird eine abstrakte Maschine vorgestellt, die eine solche maschinelle Nachahmung leistet. Die Motivation für diesen Abschnitt geht auf [MSC99] zurück, wo eine ähnliche abstrakte Maschine vorgestellt wird.

Das Kapitel beginnt mit der Definition der abstrakten Maschine $M_{\Lambda_{let}}$ und einer Einführung in deren Arbeitsweise. Eine Gegenüberstellung zu Sestofts abstrakter Maschine Mark 1 aus [Ses97] sowie der darauf aufbauenden abstrakten Maschine aus [MSC99], soll die Herkunft von $M_{\Lambda_{let}}$ darlegen. Im Anschluß wird die Austauschbarkeit der no-Reduktion des Λ_{let}-Kalküls und der abstrakten Maschine $M_{\Lambda_{let}}$ bewiesen. Die Maschine $M_{\Lambda_{let}}$ verfügt über keinen "echten" Heap. Der Heap beinhaltet bei $M_{\Lambda_{let}}$ eine Ordnungseigenschaft, deren Einführung beweistechnische Gründe hat. Deshalb wird zusätzlich die abstrakte Maschine $M^{\oplus}_{\Lambda_{let}}$ eingeführt, die über einen "echten" Heap verfügt. Für $M^{\oplus}_{\Lambda_{let}}$ wird gezeigt, daß sie mit $M_{\Lambda_{let}}$ austauschbar ist.

6.1 Die abstrakte Maschine $M_{\Lambda_{let}}$

Wir benötigen zunächst eine gegenüber Λ_{let} eingeschränkte Klasse von Ausdrücken, da die später definierte abstrakte Maschine $M_{\Lambda_{let}}$ mit allgemeinen Λ_{let} -Ausdrücken nicht umgehen kann:

Definition 6.1.1. *Die Menge der $\Lambda_{let}^{\ominus}$-Ausdrücke entspricht der Menge aller Ausdrücke, die folgender Syntax gehorchen:*

$$t, s \; := \; x \mid \texttt{choice} \mid \lambda x.t \mid t\,x \mid \texttt{let}\ x = s\ \texttt{in}\ t$$

wobei x eine Variable bezeichnet.

Ein $\Lambda_{let}^{\ominus}$-Ausdruck ist somit ein Λ_{let}-Ausdruck, der die Nebenbedingung erfüllt, daß der rechte Subausdruck jeder Applikation eine Variable ist.

Konvention 6.1.1. *Für $\Lambda_{let}^{\ominus}$-Ausdrücke gelten alle Vereinbarungen, die wir in Abschnitt 2.1.3 für Λ_{let}-Ausdrücke getroffen haben.*

Die abstrakte Maschine $M_{\Lambda_{let}}$ wird durch eine Menge von Zustandsübergängen spezifiziert, die über Zustandsbeschreibungen bestehend aus einem Heap, einem auszuwertenden Ausdruck sowie einem Stack formuliert sind. Der Heap ist dabei eine partielle Funktion von Variablen auf $\Lambda_{let}^{\ominus}$-Ausdrücke bzw. spezielle Update-Markierungen; er hält Bindungen zwischen Variablen und $\Lambda_{let}^{\ominus}$-Ausdrücken fest. In Erweiterung zu der allgemein mit einem Heap assoziierten Struktur, von einem Container zu dem Objekte hinzugefügt bzw. aus dem Objekte gelöscht werden können, besitzen die Elemente auf dem Heap stets eine festgelegte Reihenfolge. Der Heap darf keinerlei zyklische Bindungen formulieren und kann daher stets als rechtsgeschachtelter Let-Ausdruck interpretiert werden. Der auszuwertende Ausdruck bestimmt in Kombination mit dem Heap und dem Stack einen möglichen Zustandsübergang. Der Stack besteht aus Variablen und Update-Markierungen und dient zur Speicherung von Funktionsargumenten. Update-Markierungen erfüllen den Zweck, bei der Bereitstellung eines Ausdrucks aus dem Heap zur weiteren Auswertung Informationen über dessen Herkunft festzuhalten, bzw. umgekehrt das Ergebnis einer Auswertung an der korrekten Stelle im Heap abzulegen. Die abstrakte Maschine $M_{\Lambda_{let}}$ soll nun formal definiert werden.

Die Definition der abstrakte Maschine $M_{\Lambda_{let}}$ setzt sich aus zwei Komponenten zusammen; der Definition einer Menge von Zustandsbeschreibungen und der Definition einer partiellen Funktion, die eine Überführung zwischen den Zustandsbeschreibungen festlegt.

Definition 6.1.2. *Eine Zustandsbeschreibung der abstrakten Maschine $M_{\Lambda_{let}}$ hat die Form $\langle \Gamma, t, S \rangle$ wobei Γ einen Heap, t den zu evaluierenden Ausdruck und S einen Stack bezeichnet. Dabei gilt im einzelnen:*

1. *Der Heap Γ besteht aus einer Folge von Bindungen $(x_1 = t_1, \dots, x_n = t_n)$, bei der für alle $i \in \{1, \dots, n\}$ gilt, daß x_i eine Variable und t_i ein $\Lambda_{let}^{\ominus}$-Ausdruck oder eine Update-Markierung $\#x$ ist. Der leere Heap wird durch die leere Folge $()$ dargestellt. Die Schreibweise $\overrightarrow{x = t}$ repräsentiert*

nichts oder eine beliebig lange Folge einzelner Bindungen. Die Menge der Variablen, für die der Heap eine Bindung formuliert, wird durch $\mathrm{dom}(\Gamma) := \{x_1, \ldots, x_n\}$ *bezeichnet.*

2. *Der zu evaluierende Ausdruck t ist aus $\Lambda_{let}^{\ominus}$.*
3. *Der Stack S besteht aus Variablen und Update-Markierungen. Die Schreibweise $x : S$ bezeichnet den Stack S mit der Variable x als Element an der Spitze. Entsprechend bezeichnet die Schreibweise $\#x : S$ den Stack S mit der Update-Markierung $\#x$ an der Spitze. Der leere Stack wird durch ε bezeichnet. Die Komposition zwei Stacks S und T wird durch ST bezeichnet.*
4. *Update-Markierungen haben die Form $\#x$, wobei x eine Variable bezeichnen muß.*

Eine Zustandsbeschreibung $\langle \Gamma, t, S\rangle$ mit $n \geq 0$ ist gültig, *wenn sie zusätzlich folgenden Eigenschaften erfüllt:*

1. *Zu jeder $\#x$-Markierung in S existiert genau eine $\#x$-Markierung auf dem Heap und umgekehrt. Alle Update-Markierungen in S sind verschieden. Alle Bindungen von Update-Markierungen auf dem Heap Γ haben die Form $x = \#x$ für eine Variable x.*
2. *Alle Bindungen auf dem Heap werden durch verschiedene Variablen hergestellt. Dasselbe gilt für alle Ausdrücke, die auf dem Heap vorkommen, sowie bei dem zu evaluierenden Ausdruck.*
3. *Der Heap Γ habe die Form $(x_1 = t_1, \ldots, x_n = t_n)$. Dann muß für alle $i \in \{1, \ldots, n\}$ gelten, daß in t_i keine Variable x_j mit $j \geq i$ vorkommt. Existiert eine Zerlegung des Stacks $S = T' \, \#x : T$, so darf die Variable x weder in T' noch in dem zu evaluierenden Ausdruck t vorkommen.*

Eine Zustandsbeschreibung $\langle \Gamma, t, S\rangle$ mit $\Gamma = (x_1 = t_1, \ldots, x_n = t_n)$ ist geschlossen, *wenn*

$$\left(\bigcup_{i \in I} V_{free}(t_i) \cup V_{free}(t) \cup V_S\right) \subseteq \mathrm{dom}(\Gamma)$$

wobei I die Menge aller Indizes i ist, für die x_i keine Update-Markierung in Γ bindet, und V_S der Menge aller Variablen auf dem Stack S entspricht.

Zustandsbeschreibungen der abstrakten Maschine $M_{\Lambda_{let}}$ werden durch das Symbol Z zuzüglich möglicher Indizes bezeichnet.

Die letzte der drei für gültige Zustandsbeschreibungen formulierten Eigenschaften gewährleistet, daß die direkten sowie versteckten Bindungen einer gültigen Zustandsbeschreibung azyklisch sind. Wenn wir im weiteren von einer Zustandsbeschreibung sprechen, so wird implizit immer angenommen, daß diese gültig ist. Wir definieren nun die Semantik der abstrakten Maschine.

$$
\begin{array}{lrcl}
\textit{(Lookup)} & \big\langle(\overrightarrow{y=t_y}, x=t_x, \overrightarrow{z=t_z}),\ x,\ S\big\rangle & \rightarrow & \big\langle(\overrightarrow{y=t_y}, x=\#x, \overrightarrow{z=t_z}),\ t_x,\ \#x : S\big\rangle \\
\textit{(Update)} & \big\langle(\overrightarrow{y=t_y}, x=\#x, \overrightarrow{z=t_z}),\ r,\ \#x : S\big\rangle & \rightarrow & \big\langle(\overrightarrow{y=t_y}, x=r', \overrightarrow{z=t_z}),\ r,\ S\big\rangle \text{ wobei } (*) \\
\textit{(Unwind)} & \langle\Gamma,\ t\,x,\ S\rangle & \rightarrow & \langle\Gamma,\ t,\ x : S\rangle \\
\textit{(Share)} & \langle\Gamma,\ \lambda x.t,\ y : S\rangle & \rightarrow & \langle(\Gamma,\ \texttt{let}\ x = y\ \texttt{in}\ t,\ S\rangle
\end{array}
$$

$$
\textit{(Let)} \quad \begin{cases}
\Big\langle(\overrightarrow{z=t_z}, x=\#x, \overrightarrow{z'=t_z}),\ \texttt{let}\ y=t_y\ \texttt{in}\ t,\ S\Big\rangle & \text{falls } S = T\,\#x{:}S' \\
\rightarrow \Big\langle(\overrightarrow{z=t_z}, y=t_y, x=\#x, \overrightarrow{z'=t_z}),\ t,\ S\Big\rangle & \\
 & \\
\big\langle(\overrightarrow{z=t_z}),\ \texttt{let}\ y=t_y\ \texttt{in}\ t,\ S\big\rangle & \text{sonst } (**) \\
\rightarrow \big\langle(\overrightarrow{z=t_z}, y=t_y),\ t,\ S\big\rangle &
\end{cases}
$$

wobei T alleinig aus Variablen besteht oder $T = \varepsilon$ gilt

$$
\begin{array}{lrcl}
\textit{(Left)} & \langle\Gamma,\ \texttt{choice}\ ,\ x : S\rangle & \rightarrow & \langle\Gamma,\ (\lambda y.\lambda z.y),\ x : S\rangle \\
\textit{(Right)} & \langle\Gamma,\ \texttt{choice}\ ,\ x : S\rangle & \rightarrow & \langle\Gamma,\ (\lambda y.\lambda z.z),\ x : S\rangle
\end{array}
$$

$(*)$ r ist eine Abstraktion oder die Konstante `choice` . r' ist eine Kopie von r bei der alle Bindungen mittels frischer Variablen umbenannt wurden.

$(**)$ S besteht alleinig aus Variablen oder $S = \varepsilon$.

Abbildung 6.1: Die Überführungsregeln der abstrakten Maschine $M_{\Lambda_{let}}$.

Definition 6.1.3. *Die Regeln, gemäß der die abstrakten Maschine* $M_{\Lambda_{let}}$ *eine Zustandsbeschreibung* Z *in eine Zustandsbeschreibung* Z' *überführen kann, sind in Abbildung 6.1 aufgeführt.*

Wird durch die Anwendung einer Überführungsregel eine Zustandsbeschreibung Z *in eine Zustandsbeschreibung* Z' *überführt, so wird dies als* Zustandsübergang von Z nach Z' *bezeichnet. Wie der Begriff Reduktion beim* Λ_{let}*-Kalkül kann auch der Begriff Zustandsübergang transitiv erweitert benutzt werden.*
Für Zustandsübergänge wird die Notation für Reduktionen des Λ_{let}*-Kalküls übernommen; der Name der angewendeten Überführungsregel übernimmt dabei die Funktion des Namens der angewendeten Reduktion. Wir führen dabei folgende zusätzliche Schreibweise in Verbindung mit der Überführungsregel Let ein:*

$$
\big\langle(\overrightarrow{z=t_z}),\ \texttt{let}\ y=t_y\ \texttt{in}\ t,\ \varepsilon\big\rangle \xrightarrow{Let_\varepsilon} \big\langle(\overrightarrow{z=t_z}, y=t_y),\ t,\ \varepsilon\big\rangle
$$

Einige Zustandsbeschreibungen der abstrakten Maschine $M_{\Lambda_{let}}$ werden nun besonders ausgezeichnet, da sie Sonderfunktionen übernehmen.

Definition 6.1.4. *Eine Zustandsbeschreibung* $\langle (), t, \varepsilon \rangle$ *bezeichnen wir als* Startzustand. *Eine Zustandsbeschreibung* $\langle \Gamma, r, \varepsilon \rangle$ *mit* r *eine Abstraktion oder die Konstante* `choice` *bezeichnen wir als* Endzustand.

Korollar 6.1.1. *Befindet sich die abstrakte Maschine in einem Endzustand, können keine weiteren Zustandsübergänge durchgeführt werden.*

Beweis. Dies zeigt ein einfacher Abgleich mit den in Definition 6.1.3 spezifizierten Zustandsübergängen der abstrakten Maschine. □

Ein einfacher Abgleich zwischen den in Abbildung 6.1 enthaltenen Überführungsregeln und der Definition der Menge aller gültigen Zustandsbeschreibungen zeigt, daß die Definition der abstrakten Maschine $M_{\Lambda_{let}}$ sinnvoll ist. Keine Regel kann eine gültige Zustandsbeschreibung in eine ungültige überführen. Zudem zeigt die Betrachtung der Überführungsregeln, daß die anzuwendende Regel bis auf die beiden Sonderfälle *Left* und *Right* immer eindeutig ist. Diese Beobachtung führt zu folgender Definition:

Definition 6.1.5. *Sei* $Z_1 \to Z_2 \to \ldots \to Z_n$ *eine Folge einzelner Zustandsübergänge, bei der für alle* Z_i *gilt, daß die anzuwendende Überführungsregel weder Left noch Right ist. Die Folge von Zustandsübergängen von* Z_1 *nach* Z_n *wird dann als* singulär *bezeichnet.*

6.1.1 Erläuterung der Arbeitsweise von $M_{\Lambda_{let}}$

Zur Evaluation eines Ausdrucks t durch die Maschine $M_{\Lambda_{let}}$ wird ein Startzustand bestehend aus einem leeren Heap und Stack sowie t als zu evaluierendem Ausdruck gebildet. Der Ausdruck t muß dabei gemäß Voraussetzung geschlossen sein, da ansonsten das Vorkommen von freien Variablen in t zu einer Zustandsbeschreibung Z_{err} führen kann, bei der der zu evaluierende Ausdruck aus einer einzelnen Variablen x besteht, zu der jedoch keine Bindung $x = t_x$ auf dem Heap existiert. Auf Z_{err} kann keine Überführungsregel angewendet werden, da die Anwendung von *Lookup* eine Bindung erfordert. Die abstrakte Maschine befindet sich somit in einem undefinierten Zustand, da Z_{err} keinem Endzustand entspricht. Das Vorkommen dieser Situation wird durch die Voraussetzung der Geschlossenheit von t verhindert.
Wird bei dem Evaluationsprozess eine Zustandsbeschreibung Z_f mit leerem Stack und einer Abstraktion als zu evaluierenden Ausdruck erreicht, so kann kein weiterer Zustandsübergang ausgehend von Z_f erfolgen und die Maschine $M_{\Lambda_{let}}$ befindet sich gemäß Definition in einem Endzustand.

Wir betrachten nun die Wirkungsweise der einzelnen Überführungsregeln aus Abbildung 6.1 beim Evaluationsprozess. Die Überführungregeln zerfallen in vier

Gruppen, die jeweils als eine Einheit angesehen werden können. Die erste Gruppe bilden die beiden Regeln *Lookup* und *Update*. Besteht der zu evaluierende Ausdruck aus einer einzelnen Variable x und besitzt diese die zugehörige Bindung $x = t_x$ auf dem Heap, so bewegt *Lookup* den Ausdruck t_x in die Position des zu evaluierenden Ausdrucks. Zusätzlich füllt *Lookup* die frei gewordene Stelle auf dem Heap mit einer Update-Markierung auf und legt dieselbe Update-Markierung auch an die Spitze des Stacks. Die Update-Markierung wird, darin ist auch der Name begründet, später von der Regel *Update* benutzt, um einen bis zu einer Abstraktion oder einem `choice` ausgewerteten Ausdruck wieder an der korrekten Stelle und unter dem korrekten Namen auf dem Heap abzulegen. Dieser Vorgang ist vergleichbar dem Erstellen einer Schablonen-Kopie wie es bei der "Template-Instantiation-Machine" geschieht.

Unwind erfüllt die Aufgabe die Argumente des gerade evaluierten Ausdrucks auf den Stack zu bewegen. Sind alle Argumente auf den Stack bewegt, bleibt als zu evaluierender Ausdruck ein Let-Ausdruck, eine Abstraktion, die einzelne Konstante `choice` oder eine einzelne Variable zurück. Ist der verbleibende Ausdruck eine Abstraktion und befindet sich an der Spitze des Stacks eine Variable, so bildet *Share* eine Let-Bindung zwischen der abstrahierten Variable der Abstraktion und dem an der Stackspitze befindlichen Argument in Form einer anderen Variablen.

Ist der zu evaluierende Ausdruck ein Let-Ausdruck ist alleinig die *Let*-Regel anwendbar. Diese übernimmt die Aufgabe die Let-Bindung auf den Heap zu bewegen. Dabei differenziert *Let* danach, ob sich auf dem Stack eine Update-Markierung befindet oder nicht. Anhand der Update-Markierung stellt *Let* fest, welches die korrekte Position auf dem Heap ist, an der die neue Bindung eingefügt werden muß. Kommt keine Update-Markierung auf dem Stack vor, wird die Bindung am Ende des Heaps angefügt. Zwischen der zuletzt beschriebenen Differenzierung der *Let*-Regel und dem Vorkommen von L_L-Subkontexten innerhalb von W-Kontexten bei der no-Reduktion des Λ_{let}-Kalküls besteht ein direkter Bezug. Dieser Bezug wird in den Beweisen der folgenden Abschnitte deutlich und soll deshalb hier nicht weiter vertieft werden; an dieser Stelle sei nur auf diesen Zusammenhang hingewiesen, um die Ursachen für die aufwendige Form der *Let*-Regel zu verdeutlichen.

Die *Left*- und *Right*-Regel bilden den einzigen Sonderfall, bei dem die anzuwendende Regel nicht eindeutig von der Zustandsbeschreibung bestimmt wird. Sie korrespondieren mit der *nd*-Reduktion des Λ_{let}-Kalküls und ermöglichen, wie dort zwischen zwei Berechnungsalternativen auszuwählen.

6.1.2 Gegenüberstellung mit den in [MS99] und [Ses97] vorgestellten abstrakten Maschinen

Die hier vorgestellte abstrakte Maschine $M_{\Lambda_{let}}$ wurde durch die Veröffentlichung [MS99] motiviert und aus einer dort vorgestellten abstrakten Maschine hergeleitet. Die abstrakte Maschine aus [MS99] wiederum ist eine erweiterte Form der in [Ses97] von Sestoft vorgestellten abstrakten Maschine "mark 1" mit Konstruktoren. Die Erweiterung besteht dabei in der Hinzunahme von Regeln zur Modellierung von erratic Nichtdeterminismus.

Beim Design der Maschine $M_{\Lambda_{let}}$ wurden zwei wesentliche Merkmale gegenüber der Maschine aus [MS99] fortgelassen. Diese beiden Merkmale sind die Fähigkeiten, zum einen mit Konstruktoren, zum anderen mit rekursiven Bindungen umgehen zu können. Die abstrakte Maschine von [MS99] verfügt damit über zwei Eigenschaften, die viele von Compilern verwendete abstrakte Maschinen wie die G-Maschine aufweisen. Im Kontext des Λ_{let}-Kalküls werden diese Fähigkeiten jedoch nicht benötigt, da der Λ_{let}-Kalkül weder über rekursive Let-Strukturen noch über Konstruktoren verfügt und wurden daher fortgelassen. Das Ausschließen rekursiver Bindungen erlaubte das Fortlassen der *Black Hole*-Regel, da Ausdrücke der Form $\texttt{let}\ x = x\ \texttt{in}\ x$ als unzulässig angesehen werden. Der Heap kann bei der abstrakten Maschine $M_{\Lambda_{let}}$ jederzeit als azyklischer gerichteter Graph interpretiert werden.

Die als Ersatz der *LetRec*-Regel eingeführte *Let*-Regel besitzt eine auffällig aufwendigere Definition als ihr rekursives Vorbild. Die aufwendige Definition der *Let*-Regel ist im Kontext der Gegenüberstellung mit dem Λ_{let}-Kalkül erforderlich geworden. Aus demselben Grund wurden auch die *Lookup*- und *Update*-Regel verändert, das Vorkommen von Update-Markierungen auf dem Heap zugelassen und der Heap als Folge anstatt als Menge von Bindungen repräsentiert. All diese Änderungen bewirken in ihrer Gemeinsamkeit, daß jeder Zustandsbeschreibung eindeutig ein $\Lambda_{let}^{\ominus}$-Ausdruck zugeordnet werden kann. Wird die Repräsentation des Heaps als Folge zugunsten einer Menge fallengelassen, so geht diese Eigenschaft verloren. Die Zustandsbeschreibung $\langle\{x = t_x, y = t_y\}, s, \varepsilon\rangle$ kann dann beispielsweise sowohl als $\texttt{let}\ x = t_x\ \texttt{in let}\ y = t_y\ \texttt{in}\ s$ wie auch als $\texttt{let}\ y = t_y\ \texttt{in let}\ x = t_x\ \texttt{in}\ s$ interpretiert werden, was in den Beweisen der folgenden Abschnitte ausgesprochen störend wäre.

Einen weiteren Unterschied zwischen beiden abstrakten Maschinen betrifft die nichtdeterministischen Zustandsübergänge *Left* und *Right*. Die in [MS99] benutzte Sprache $\Lambda_{\text{need}}^{\oplus}$[1] benutzt einen Operator $\oplus$ mit zwei $\Lambda_{\text{need}}^{\oplus}$-Ausdrücken als Argumente zur Spezifikation der nichtdeterministischen Wahlmöglichkeit während der Auswertung. Die abstrakte Maschine $M_{\Lambda_{let}}$ wurde jedoch in Anlehnung an den Λ_{let}-Kalkül spezifiziert, der für diesen Zweck die Konstante `choice` kennt.

---

[1] Die Definition der Sprache $\Lambda_{\text{need}}^{\oplus}$, sowie der Semantik des Operators $\oplus$ erfolgt in [MS99]

Daher wurde eine entsprechende Änderung der Überführungsregeln *Left* und *Right* notwendig. Da die Konstante `choice` im Λ_{let}-Kalkül als kopierbar angesehen wird, war auch ein Angleich der *Update*-Regel erforderlich, der diesem Tatbestand gerecht wurde.

Die Überführungsregel *Subst* der abstrakten Maschine aus [MS99] wurde durch die Regel *Share* ersetzt. Die *Subst*-Regel entspricht aus der Sichtweise des Λ_{let}-Kalküls einer Kombination aus einer Folge von *lbeta*, *lcv* und *ldel*-Reduktionen. Die *Share*-Regel entspricht einer auf die *lbeta*-Reduktion reduzierten Form der *Subst*-Regel. Dies erleichtert die spätere Gegenüberstellung mit dem Λ_{let}-Kalkül, obwohl die Evaluation eines Ausdrucks durch die abstrakte Maschine $M_{\Lambda_{let}}$ dadurch häufig verlängert wird.

6.2 Austauschbarkeit der no-Reduktion des Λ_{let}-Kalküls und der abstrakten Maschine $M_{\Lambda_{let}}$

Wir führen nun eine Interpretationsfunktion ein, die eine Zustandsbeschreibung $\langle \Gamma, t, S\rangle$ auf einen Λ_{let}-Ausdruck abbildet.

Definition 6.2.1. *Wir definieren wie folgt eine Relation $\lessdot$ auf der Menge aller Zustandsbeschreibungen:*

$$Z \lessdot Z' \textit{ gdw. } Z \xrightarrow{Lookup} Z' \textit{ oder } Z \xrightarrow{Unwind} Z' \textit{ oder } Z \xrightarrow{Let_\varepsilon} Z'$$

Mit $<$ wird der transitive Abschluß von $\lessdot$ bezeichnet. Eine Zustandsbeschreibung Z heißt minimal*, wenn keine Zustandsbeschreibung Z' mit $Z' < Z$ existiert.*

Wir werden nun einige Eigenschaften kennenlernen, die die oben definierte Relation erfüllt.

Lemma 6.2.1.

(1) Jede minimale Zustandsbeschreibung hat die Form $\langle (), t, \varepsilon\rangle$, mit t als einem $\Lambda_{let}^{\ominus}$-Ausdruck.

(2) Zu jeder Zustandsbeschreibung Z existiert genau eine minimale Zustandsbeschreibung Z', so daß $Z' \leq Z$ gilt.

Beweis. Die untenstehende Tabelle gibt die Wirkungsweise der einzelnen Reduktionen wieder, auf denen die Relation $\lessdot$ basiert:

$$\begin{array}{rcll}
\langle((\ldots, x = \#x, \ldots), t_x, \#x : S\rangle & \gtrdot & \langle((\ldots, x = t_x, \ldots), x, S\rangle & (\xleftarrow{LookUp}) \\
\big\langle(\overrightarrow{x = t_x}), t, y : S\big\rangle & \gtrdot & \big\langle(\overrightarrow{x = t_x}), t\, y, S\big\rangle & (\xleftarrow{Unwind}) \\
\big\langle(\overrightarrow{x = t_x}, y = t_y), t, \varepsilon\big\rangle & \gtrdot & \big\langle(\overrightarrow{x = t_x}), \texttt{let } y = t_y \texttt{ in } t, \varepsilon\big\rangle & (\xleftarrow{Let_\varepsilon})
\end{array}$$

(1) Angenommen es existiert eine minimale Zustandsbeschreibung, die nicht die Form $\langle (), t, \varepsilon \rangle$ besitzt. Dann ist entweder der Stack oder der Heap oder beides nicht leer. In allen Fällen existiert gemäß obiger Tabelle eine kleinere Zustandsbeschreibung und es entsteht ein Widerspruch zu der Annahme der Minimalität.

(2) Wir zeigen zunächst, daß zu jeder Zustandsbeschreibung eine minimale Zustandsbeschreibung existiert. Definieren wir als Bewertungsmaß von Zustandsbeschreibungen ein Tupel, dessen erste Komponente die Anzahl der Bindungen auf dem Heap und dessen zweite Komponente die Anzahl der Elemente auf dem Stack wiedergibt, dann wird dieses Maß durch die Relation $\gtrdot$ stets verkleinert. Daraus folgt direkt die Existenz der gesuchten minimalen Zustandsbeschreibung. Die Eindeutigkeit ergibt sich daraus, daß zu jeder nicht minimalen Zustandsbeschreibung genau ein kleineres Element existiert, wie ein einfacher Abgleich mit der obigen Tabelle zeigt. □

Das obige Lemma erlaubt folgende Definition.

Definition 6.2.2. *Als die zu einer Zustandsbeschreibung Z gehörige minimale Zustandsbeschreibung bezeichnen wir die eindeutige minimale Zustandsbeschreibung Z', für die $Z' \leq Z$ gilt.*

Wir benutzen die oben gewonnen Informationen über die Existenz eines kleinsten Elements nun, um jeder Zustandsbeschreibung einen $\Lambda^{\ominus}_{let}$-Ausdruck zuzuordnen.

Definition 6.2.3. *Wir definieren eine Interpretation $[\![\ ]\!]_\ell$, die jeder Zustandbeschreibung Z einen $\Lambda^{\ominus}_{let}$-Ausdruck zuordnet, wie folgt:*

Als $[\![Z]\!]_\ell$ wählen wir den Ausdruck t aus der zu Z gehörigen minimalen Zustandsbeschreibung $[\![\langle (), t, \varepsilon \rangle]\!]_\ell$.

Korollar 6.2.1. *Z und Z' seien zwei Zustandsbeschreibungen.*

(1) $Z < Z' \quad \Leftrightarrow \quad Z \xrightarrow{a,+} Z'$ *wobei* $a \in \{Lookup, Unwind, Let_\varepsilon\}$

(2) $[\![Z]\!]_\ell = [\![Z']\!]_\ell \quad \Leftrightarrow \quad Z < Z'$ *oder* $Z > Z'$ *oder* $Z = Z'$

(3) $Z \xrightarrow{a,+} Z'$ *wobei* $a \in \{Lookup, Unwind, Let_\varepsilon\} \quad \Rightarrow \quad [\![Z]\!]_\ell = [\![Z']\!]_\ell$

(4) *Für alle Zustandsbeschreibungen $Z = \langle \Gamma, t, S \rangle$ existiert ein Kontext C, so daß $[\![Z]\!]_\ell = C[t]$.*

Beweis. (1) folgt direkt aus dem transitiven Abschluß der Relation $\lessdot$.

(2) ergibt sich direkt aus der Eindeutigkeit der zu jeder Zustandsbeschreibung zugehörigen minimalen Zustandsbeschreibung.

(3) ist eine einfache logische Kombination von *(1)* und *(2)*.

(4) Die in obigem Lemma enthaltene Tabelle zeigt, daß inverse *Lookup*-, *Unwind*–, Let_ε-Zustandsübergänge den zu evaluierenden Ausdruck t niemals zerlegen oder verändern, sondern nur bewegen oder einkapseln.

□

Wie bei Anwendung der Interpretation $[\![\ ]\!]_\ell$ der zu einer Zustandsbeschreibung gehörige $\Lambda_{let}^{\ominus}$-Ausdruck hergeleitet wird, soll an einem Beispiel verdeutlicht werden:

Beispiel 6.2.1. Gegeben sei die Zustandbeschreibung:

$$Z_1 = \langle (x = \#x, y = t_y, z = \#z), t_x, \#x : y : x : \#z \rangle$$

Zu Z_1 gehört folgende absteigende Kette kleinerer Zustandsbeschreibungen:

$$\begin{array}{llllr}
 & \langle (x = \#x, y = t_y, z = \#z), & t_x, & \#x : y : x : \#z\rangle & (Z_1) \\
\gtrdot & \langle (x = t_x, y = t_y, z = \#z), & x, & y : x : \#z\rangle & (Z_2) \\
\gtrdot & \langle (x = t_x, y = t_y, z = \#z), & x\,y, & x : \#z\rangle & (Z_3) \\
\gtrdot & \langle (x = t_x, y = t_y, z = \#z), & (x\,y)\,x, & \#z\rangle & (Z_4) \\
\gtrdot & \langle (x = t_x, y = t_y, z = (x\,y)\,x), & z, & \varepsilon\rangle & (Z_5) \\
\gtrdot & \langle (x = t_x, y = t_y), & \multicolumn{1}{l}{(\texttt{let}\ z = (x\,y)\,x\ \texttt{in}\ z))} & \varepsilon\rangle & (Z_6) \\
\gtrdot & \langle (x = t_x), & \multicolumn{1}{l}{(\texttt{let}\ y = t_y\ \texttt{in}\ (\texttt{let}\ z = (x\,y)\,x\ \texttt{in}\ z))} & \varepsilon\rangle & (Z_7) \\
\gtrdot & \langle (), & \multicolumn{1}{l}{\texttt{let}\ x = t_x\ \texttt{in}\ (\texttt{let}\ y = t_y\ \texttt{in}\ (\texttt{let}\ z = (x\,y)\,x\ \texttt{in}\ z))} & \varepsilon\rangle & (Z_8)
\end{array}$$

Offensichtlich ist Z_8 minimal. Daraus folgt direkt, daß

$$[\![Z_1]\!]_\ell = \texttt{let}\ x = t_x\ \texttt{in}\ (\texttt{let}\ y = t_y\ \texttt{in}\ (\texttt{let}\ z = (x\,y)\,x\ \texttt{in}\ z))$$

gilt.

Wir zeigen nun eine spezielle Eigenschaft von Ausdrücken mit einem W-Kontext, in Verbindung mit der Anwendung einer Folge von Zustandsübergängen.

Lemma 6.2.2. *Jede Zustandsbeschreibung Z der Form $\langle \Gamma, W[t], \varepsilon\rangle$ kann durch eine eindeutige Folge von Zustandsübergängen in eine Zustandsbeschreibung Z' der Form $\langle \Gamma', t, S\rangle$ überführt werden, für die $[\![Z]\!]_\ell = [\![Z']\!]_\ell$ gilt.*

Beweis. Wir führen eine strukturelle Induktion über den Aufbau eines W-Kontexts. Als Induktionshypothese benutzen wir die Hypothese des Lemmas.

Induktionsbeginn $W \equiv L_R^*[A_L^*[\cdot]]$

$$\begin{array}{llr}
 & \big\langle (\overrightarrow{x = t_x}),\ L_R^*[A_L^*[t]],\ \varepsilon\big\rangle & (Z_1) \\
\xrightarrow{Let_\varepsilon,*} & \big\langle (\overrightarrow{x = t_x}, \overrightarrow{y = t_y}),\ A_L^*[t],\ \varepsilon\big\rangle & (Z_2) \\
\xrightarrow{Unwind,*} & \big\langle (\overrightarrow{x = t_x}, \overrightarrow{y = t_y}),\ t,\ S\big\rangle & (Z_3)
\end{array}$$

Die Gültigkeit von $[\![Z_1]\!]_\ell = [\![Z_2]\!]_\ell = [\![Z_3]\!]_\ell$ ergibt sich direkt aus Korollar 6.2.1.

Induktionsschritt $W \equiv L_R^*[L_L[A_L^*[\cdot]]]$ *wobei* $L_L \equiv \texttt{let}\ z = [\cdot]\ \texttt{in}\ W[z]$

$$\begin{array}{rll} & \left\langle(\overrightarrow{x = t_x}),\ L_R^*[\texttt{let}\ z = A_L^*[t]\ \texttt{in}\ W[z]],\ \varepsilon\right\rangle & (Z_1) \\ \xrightarrow{Let_\varepsilon,*} & \left\langle(\overrightarrow{x = t_x}, \overrightarrow{y = t_y}),\ \texttt{let}\ z = A_L^*[t]\ \texttt{in}\ W[z],\ \varepsilon\right\rangle & (Z_2) \\ \xrightarrow{Let_\varepsilon} & \left\langle(\overrightarrow{x = t_x}, \overrightarrow{y = t_y}, z = A_L^*[t]),\ W[z],\ \varepsilon\right\rangle & (Z_3) \\ \xrightarrow{*} & \left\langle(\overrightarrow{x = t_x}, \overrightarrow{y = t_y}, z = A_L^*[t], \overrightarrow{z' = t_z}),\ z,\ S\right\rangle & (Z_4) \\ \xrightarrow{Lookup} & \left\langle(\overrightarrow{x = t_x}, \overrightarrow{y = t_y}, z = \#z, \overrightarrow{z' = t_z}),\ A_L^*[t],\ S\right\rangle & (Z_5) \\ \xrightarrow{Unwind,*} & \left\langle(\overrightarrow{x = t_x}, \overrightarrow{y = t_y}, z = \#z, \overrightarrow{z' = t_z}),\ t,\ x_1 : \ldots x_n : S\right\rangle & (Z_6) \end{array}$$

Der Übergang von Z_3 nach Z_4 ergibt sich aus der Induktionsvoraussetzung, die zusätzlich aussagt, daß $[\![Z_3]\!]_\ell = [\![Z_4]\!]_\ell$ gilt. Die Gültigkeit von $[\![Z_1]\!]_\ell = [\![Z_2]\!]_\ell = [\![Z_3]\!]_\ell$ sowie $[\![Z_4]\!]_\ell = [\![Z_5]\!]_\ell = [\![Z_6]\!]_\ell$ ergibt sich wiederum aus Korollar 6.2.1. □

Es sei an dieser Stelle angemerkt, daß das obige Lemma seine Gültigkeit verliert, wenn die Bedingung eines leeren Stacks fallengelassen wird, d.h. Zustandsbeschreibungen $\langle \Gamma, W[t], S\rangle$ mit $S \neq \varepsilon$ zugelassen werden, da dann nicht notwendigerweise Let_ε-Zustandsübergänge vorkommen.

Abbildung 6.2 gibt die Berechnung der abstrakten Maschine angewendet auf den Ausdruck $\texttt{let}\ x = \texttt{let}\ y = \lambda z.z\ \texttt{in}\ y\ \texttt{in}\ x$ wieder. Auf der rechten Seite letzterer Abbildung wird für jede Zustandsbeschreibung der $\Lambda_{let}^{\ominus}$-Ausdruck angegeben, der bei der Anwendung der Interpretation $[\![\ ]\!]_\ell$ erhalten wird. Der Äquivalenzoperator bzw. die Reduktionen zeigen dabei an, in welcher Beziehung die erhaltenen $\Lambda_{let}^{\ominus}$-Ausdrücke stehen. Offensichtlich korrespondieren bei dem gewählten Beispiel die Zustandsübergänge der abstrakten Maschine mit no-Reduktionen im Λ_{let}-Kalkül. Wir werden zunächst zeigen, daß eine Beziehung zwischen dem no-Markierungsalgorithmus und einer bestimmten Klasse von Zustandsbeschreibungen existiert.

Lemma 6.2.3. *Gegeben eine Zustandsbeschreibung* $Z = \langle \Gamma, t_s, S\rangle$ *sowie ein Kontext* C*, so daß* $[\![Z]\!]_\ell = C[t_s]$ *gilt. Wenn* t_s *eine Applikation oder eine Variable ist, dann wird im Verlauf der Anwendung des no-Markierungsalgorithmus auf den Ausdruck* $C[t_s]$ *an den Subausdruck* t_s *ein* E*-Label angeheftet.*

Beweis. Wir führen eine Induktion nach der Höhe des Stacks S über die Menge aller Zustandsbeschreibungen. Als Induktionshypothese benutzen wir die Hypothese des Lemmas.

Induktionsbeginn: $S = \varepsilon$
Es gilt $[\![\langle \Gamma, t_s, \varepsilon\rangle]\!]_\ell = L_R^*[t_s]$. Bei der Anwendung des no-Markierungsalgorithmus

$$
\begin{array}{ccc}
\langle (), \texttt{let}\ x = \texttt{let}\ y = \lambda z.z\ \texttt{in}\ y\ \texttt{in}\ x, \varepsilon \rangle & =_{[\![\,]\!]_\ell} & \texttt{let}\ x = \texttt{let}\ y = \lambda z.z\ \texttt{in}\ y\ \texttt{in}\ x \\
\xrightarrow{\mathit{Let}} & & \equiv \\
\langle (x = \texttt{let}\ y = \lambda z.z\ \texttt{in}\ y), x, \varepsilon \rangle & =_{[\![\,]\!]_\ell} & \texttt{let}\ x = \texttt{let}\ y = \lambda z.z\ \texttt{in}\ y\ \texttt{in}\ x \\
\xrightarrow{\mathit{Lookup}} & & \equiv \\
\langle (x = \#x), \texttt{let}\ y = \lambda z.z\ \texttt{in}\ y, \#x \rangle & =_{[\![\,]\!]_\ell} & \texttt{let}\ x = \texttt{let}\ y = \lambda z.z\ \texttt{in}\ y\ \texttt{in}\ x \\
\xrightarrow{\mathit{Let}} & & \xrightarrow{\mathit{no,llet}} \\
\langle (y = \lambda z.z, x = \#x), y, \#x \rangle & =_{[\![\,]\!]_\ell} & \texttt{let}\ y = \lambda z.z\ \texttt{in}\ (\texttt{let}\ x = y\ \texttt{in}\ x) \\
\xrightarrow{\mathit{Lookup}} & & \equiv \\
\langle (y = \#y, x = \#x), \lambda z.z, \#y : \#x \rangle & =_{[\![\,]\!]_\ell} & \texttt{let}\ y = \lambda z.z\ \texttt{in}\ (\texttt{let}\ x = y\ \texttt{in}\ x) \\
\xrightarrow{\mathit{Update}} & & \xrightarrow{\mathit{no,cp}} \\
\langle (y = \lambda z.z, x = \#x), \lambda z.z, \#x \rangle & =_{[\![\,]\!]_\ell} & \texttt{let}\ y = \lambda z.z\ \texttt{in}\ (\texttt{let}\ x = \lambda z.z\ \texttt{in}\ x) \\
\xrightarrow{\mathit{Update}} & & \xrightarrow{\mathit{no,cp}} \\
\langle (y = \lambda z.z, x = \lambda z.z), \lambda z.z, \varepsilon \rangle & =_{[\![\,]\!]_\ell} & \texttt{let}\ y = \lambda z.z\ \texttt{in}\ (\texttt{let}\ x = \lambda z.z\ \texttt{in}\ \lambda z.z)
\end{array}
$$

Abbildung 6.2: Gegenüberstellung der Berechnung der abstrakten Maschine und der no-Reduktion für einen gegebenen Beispielausdruck

wird das E-Label über alle Let-Bindungen aus L_R^* hinweg nach innen bewegt und an den Subausdruck t_s angeheftet, wie sich durch Induktion über die Anzahl der Bindungen in Γ leicht zeigen läßt.

Induktionsschritt:
Fall $S = x : S'$
Es seien $Z = \langle \Gamma, t_s, x : S' \rangle$ und $Z' = \langle \Gamma, t_s\, x, S' \rangle$. Aus $Z' \xrightarrow{\mathit{Unwind}} Z$ folgt in Kombination mit Korollar 6.2.1 $[\![Z]\!]_\ell = [\![Z']\!]_\ell$. Aus Korollar 6.2.1 folgt ferner die Existenz eines Kontexts C für den $[\![Z']\!]_\ell = C[t_s\, x]$ gilt. In Kombination mit der Induktionsvoraussetzung folgt daraus, daß das E-Label im Verlauf der no-Markierung an den Subausdruck $(t_s\, x)$ angeheftet wird, d.h. $C[(t_s\, x)^E]$ erhalten wird. Falls t_s eine Applikation oder eine Variable ist, wird gemäß Regel *iv)* des no-Markierungsalgorithmus das E-Label im nächsten Schritt an den Subausdruck t_s geheftet und die Hypothese ist für diesen Fall gezeigt.

Fall $S = \#x : S'$
Es seien $Z = \langle (\ldots x = \#x \ldots), t_s, \#x : S' \rangle$ und $Z' = \langle (\ldots x = t_s \ldots), x, S' \rangle$. Aus $Z' \xrightarrow{\mathit{Lookup}} Z$ folgt in Kombination mit Korollar 6.2.1 $[\![Z]\!]_\ell = [\![Z']\!]_\ell$. Aus Korollar 6.2.1 folgt ferner die Existenz eines Kontexts C für den $[\![Z']\!]_\ell = C[x]$ gilt. In Kombination mit der Induktionsvoraussetzung folgt daraus, daß das E-Label im Verlauf der no-Markierung an die Variable x angeheftet wird, d.h. $C[x^E]$ erhalten wird. Eine einfache Analyse der Interpretation $[\![\ ]\!]_\ell$ zeigt, daß sich $C[x^E]$ stets in $L_R^*[\texttt{let}\ x = t_s\ \texttt{in}\ D[x^E]]$ zerlegen läßt. Gemäß Regel *ix)* des no-Markierungsalgorithmus wird das E-Label im nächsten Schritt an den Subausdruck t_s geheftet, womit die Hypothese auch für diesen Fall gezeigt. □

$$\begin{array}{rcl}
Z \xrightarrow{Unwind} Z' & \Rightarrow & t \equiv t' \\
Z \xrightarrow{Lookup} Z' & \Rightarrow & t \equiv t' \\
Z \xrightarrow{Update} Z' & \Rightarrow & t \xrightarrow{no,cp} t' \\
Z \xrightarrow{Share} Z' & \Rightarrow & t \xrightarrow{no,lbeta} t' \\
Z \xrightarrow{Let_\varepsilon} Z' & \Rightarrow & t \equiv t' \\
Z \xrightarrow{Let} Z' \text{ und } (*) & \Rightarrow & t \xrightarrow{no,lapp,+} t' \\
Z \xrightarrow{Let} Z' \text{ und } (**) & \Rightarrow & t \xrightarrow{no,lapp,*} \circ \xrightarrow{no,llet} t' \\
Z \xrightarrow{Left} Z' & \Rightarrow & t \xrightarrow{no,nd_{left}} t' \\
Z \xrightarrow{Right} Z' & \Rightarrow & t \xrightarrow{no,nd_{right}} t'
\end{array}$$

(*) der Stack von Z besteht nur aus Variablen

(**) der Stack von Z verfügt über mindestens eine Indirektion

Abbildung 6.3: Entsprechungen von Zustandsübergängen und no-Reduktionen aus Lemma 6.2.4 im Detail.

Lemma 6.2.4. *Gegeben zwei Zustandsbeschreibungen Z und Z'. t und t' seien zwei Ausdrücke mit $t = [\![Z]\!]_\ell$ und $t' = [\![Z']\!]_\ell$. Wenn $Z \stackrel{\cdot}{\longrightarrow} Z'$, dann gilt $t \xrightarrow{no,*} t'$. Im Detail gelten dabei die Entsprechungen aus Abbildung 6.3.*

Beweis. Wir untersuchen nacheinander alle möglichen Zustandsübergänge der abstrakten Maschine:

(*Lookup* und *Unwind*)
Gemäß Korollar 6.2.1 verändert ein *Lookup*- sowie ein *Unwind*-Zustandsübergang die Interpretation einer Zustandsbeschreibung nicht. Beide Arten von Zustandsübergängen entsprechen somit der syntaktischen Äquivalenz.

(*Update*)
Wir werden zeigen, daß jeder *Update*-Zustandsübergang einer (no, cp)-Reduktion im Λ_{let}-Kalkül entspricht.

Der *Update*-Zustandsübergang habe die Form $Z \xrightarrow{Update} Z'$ mit $Z = \langle(\dots x = \#x \dots), r, \#x : S\rangle$ und $Z' = \langle(\dots x = r \dots), r, S\rangle$. Wir wählen $Z'' = \langle(\dots x = r \dots), x, S\rangle$ derart, daß $Z \gtrdot Z''$ gilt. Aus Korollar 6.2.1 folgt nun $[\![Z]\!]_\ell = [\![Z'']\!]_\ell$ sowie die Existenz eines Kontexts C, daß $[\![Z'']\!]_\ell = C[x]$ gilt. Lemma 6.2.3 besagt, daß bei Anwendung des no-Markierungsalgorithmus auf $C[x]$ die Variable x mit einem E-Label versehen wird. Da für x eine Bindung auf dem Heap formuliert ist, muß eine Zerlegung $C[x^E] \equiv D[\texttt{let}\ x = r\ \texttt{in}\ E[x^E]]$ existieren. Da ferner r eine Abstraktion oder die Konstante `choice` ist, liegt offensichtlich ein (no, cp)-Redex vor. Die Reduktion des no-Redex liefert den Ausdruck $D[\texttt{let}\ x = r\ \texttt{in}\ E[r]]$, der genau $[\![\langle Z'\rangle]\!]_\ell$ entspricht, wie aus einer Gegenüberstellung von Z' und Z'' in Kombination mit 6.2.1 sofort ersichtlich ist.

(*Share*)
Wir werden zeigen, daß jeder *Share*-Zustandsübergang einer $(no, lbeta)$-Reduktion im Λ_{let}-Kalkül entspricht.

Die Argumentation erfolgt ähnlich wie beim *Update*-Zustandsübergang. Der *Share*-Zustandsübergang habe die Form $Z \xrightarrow{Update} Z'$ mit $Z = \langle \Gamma, \lambda x.t, y : S\rangle$ und $Z' = \langle \Gamma, \texttt{let}\ x = y\ \texttt{in}\ t, S\rangle$. Wir wählen $Z'' = \langle \Gamma, (\lambda x.t)y, S\rangle$, offensichtlich gilt $Z \gtrdot Z''$. Aus Korollar 6.2.1 folgt nun $[\![Z]\!]_\ell = [\![Z'']\!]_\ell$ sowie die Existenz eines Kontexts C, daß $[\![Z'']\!]_\ell = C[(\lambda x.t)y]$ gilt. Lemma 6.2.3 besagt, daß bei Anwendung des no-Markierungsalgorithmus eine Markierung $C[((\lambda x.t)y)^E]$ erhalten wird. Somit liegt offensichtlich ein $(no, lbeta)$-Redex vor. Die Reduktion des no-Redex liefert den Ausdruck $D[\texttt{let}\ x = y\ \texttt{in}\ t]$, der genau $[\![\langle Z'\rangle]\!]_\ell$ entspricht, wie aus einer Gegenüberstellung von Z' und Z'' in Kombination mit 6.2.1 sofort ersichtlich ist.

(*Let*)
Wir werden zeigen, daß jeder *Let*-Zustandsübergang entweder der syntaktischen Äquivalenz oder einer Folge von $(no, lapp)$-Reduktionen die mit einer optionalen $(no, llet)$-Reduktion abgeschlossen werden entspricht.

Sei $Z = \langle \Gamma, \texttt{let}\ y = t_y\ \texttt{in}\ t, S\rangle$ die Zustandsbeschreibung, auf die der *Let*-Zustandsübergang angewendet wird. Wir differenzieren dazu nach dem Aussehen des Stacks.

1. Fall $S = \varepsilon$
In diesem Fall entspricht der *Let*-Zustandsübergang einem Let_ε-Zustandsübergang, der gemäß Korollar 6.2.1 bezogen auf die Interpretation $[\![\ ]\!]_\ell$ der syntaktischen Äquivalenz entspricht.

2. Fall $S = x_1 : \cdots x_n : \#z : T$
An der Spitze des Stacks befindet sich eine Folge von Variablen gefolgt von einer Update-Markierung. Wegen der Gültigkeit der Zustandsbeschreibung Z muß der Heap Γ die Form $(\ldots z = \#z \ldots)$ besitzen. Wir betrachten die Herleitung der zu Z gehörigen minimalen Zustandsbeschreibung über einige Schritte:

$$\begin{array}{lll}
 & \langle(\ldots z = \#z \ldots),\ \texttt{let}\ y = t_y\ \texttt{in}\ t,\ x_1 : \cdots x_n : \#z : S\rangle & (Z) \\
\gtrdot & \langle(\ldots z = \#z \ldots),\ (\texttt{let}\ y = t_y\ \texttt{in}\ t)x_1,\ x_2 : \cdots x_n : \#z : S\rangle & (Z_1) \\
> & \langle(\ldots z = \#z \ldots),\ ((\texttt{let}\ y = t_y\ \texttt{in}\ t)x_1 \cdots x_n),\ \#z : S\rangle & (Z_n) \\
\gtrdot & \langle(\ldots z = ((\texttt{let}\ y = t_y\ \texttt{in}\ t)x_1 \cdots x_n) \ldots),\ z,\ S\rangle & (Z_{n+1}) \\
> & \langle(),\ t_0,\ S\rangle & (Z_{min})\ \text{- } t_0 \text{ siehe unten}
\end{array}$$

Aus einer einfachen Überlegung unter Zuhilfenahme von Korollar 6.2.1 folgt die Gültigkeit von $[\![Z]\!]_\ell = [\![Z_{n+1}]\!]_\ell$ sowie die Existenz zweier Kontexte C und D, so daß $[\![Z_{n+1}]\!]_\ell = t_0$ gilt. Gemäß Lemma 6.2.3 wird die Variable z bei der Anwendung des no-Markierungsalgorithmus mit einem E-Label versehen. Die weitere Markierung des no-Redex sowie die entstehende no-Reduktion haben das folgende Aussehen:

$$\begin{array}{lll}
 & D[\mathtt{let}\ z = ((((\mathtt{let}\ y = t_y\ \mathtt{in}\ t)x_1)^E x_2)^e \cdots x_n)^e\ \mathtt{in}\ C[z^e]] & (t_0 = \llbracket Z \rrbracket_\ell) \\
\xrightarrow{no,lapp} & D[\mathtt{let}\ z = (((\mathtt{let}\ y = t_y\ \mathtt{in}\ t\, x_1)x_2)^E \cdots x_n)^e\ \mathtt{in}\ C[z^e]] & (t_1) \\
\xrightarrow{no,lapp,*} & D[\mathtt{let}\ z = (\mathtt{let}\ y = t_y\ \mathtt{in}\ t\, x_1\, x_2 \cdots x_n)\ \mathtt{in}\ C[z^E]] & (t_n) \\
\xrightarrow{no,llet} & D[\mathtt{let}\ y = t_y\ \mathtt{in}\ (\mathtt{let}\ z = t\, x_1\, x_2 \cdots x_n\ \mathtt{in}\ C[z])] & (t_{n+1} = \llbracket Z' \rrbracket_\ell)
\end{array}$$

Wir betrachten nun die Zustandsbeschreibung Z' nach Anwendung des Zustandsübergangs und verfolgen wiederum die Entwicklung der zugehörigen minimalen Zustandsbeschreibung über einige Schritte:

$$\begin{array}{lll}
 & \langle (\ldots y = t_y, z = \#z \ldots),\ t,\ x_1 : \cdots x_n : \#z : S \rangle & (Z') \\
> & \langle (\ldots y = t_y, z = \#z \ldots),\ ((t\, x_1) \cdots x_n),\ \#z : S \rangle & (Z'_n) \\
\gtrdot & \langle (\ldots y = t_y, z = ((t\, x_1) \cdots x_n) \ldots),\ z,\ S \rangle & (Z'_{n+1}) \\
> & \langle (),\ t_{n+1},\ S \rangle & (Z'_{min})
\end{array}$$

Aus Korollar 6.2.1 folgt $\llbracket Z' \rrbracket_\ell = \llbracket Z'_{n+1} \rrbracket_\ell$. Ein Vergleich von Z_{n+1} und Z'_{n+1} zeigt, daß $\llbracket Z'_{n+1} \rrbracket_\ell$ und damit auch $\llbracket Z' \rrbracket_\ell$ dem Ausdruck t_{n+1} entspricht.
Kommt an der Spitze des Stacks direkt eine Update-Markierung vor, so fallen alle $(no, lapp)$-Reduktionen weg und es bleibt eines einzelne $(no, llet)$-Reduktion übrig.

3. Fall $S = x_1 : \cdots x_n$, der Stack verfügt über keine Update-Markierung.
Dieser Fall ist eine simplifizierte Form des letzten Falls. Wir betrachten die Herleitung der zu Z gehörigen minimalen Zustandsbeschreibung über einige Schritte:

$$\begin{array}{lll}
 & \big\langle (\overrightarrow{x = t}),\ \mathtt{let}\ y = t_y\ \mathtt{in}\ t,\ x_1 : \cdots x_n \big\rangle & (Z) \\
= & \big\langle (\overrightarrow{x = t}),\ (\mathtt{let}\ y = t_y\ \mathtt{in}\ t)x_1,\ x_2 : \cdots x_n \big\rangle & (Z_1) \\
= & \big\langle (\overrightarrow{x = t}),\ ((\mathtt{let}\ y = t_y\ \mathtt{in}\ t)x_1 \cdots x_n),\ \varepsilon \big\rangle & (Z_n)
\end{array}$$

Aus einer einfachen Überlegung unter Zuhilfenahme von Korollar 6.2.1 folgt die Gültigkeit von $\llbracket Z \rrbracket_\ell = \llbracket Z_n \rrbracket_\ell$ sowie $\llbracket Z_n \rrbracket_\ell = t_0$. Die Kenntnis aus Lemma 6.2.3 über das Anbringen des E-Label bei Anwendung des no-Markierungsalgorithmus erlaubt folgende Entwicklung:

$$\begin{array}{lll}
 & L_R^*[((((\mathtt{let}\ y = t_y\ \mathtt{in}\ t)x_1)^E x_2)^e \cdots x_n)^e] & (t_0 = \llbracket Z \rrbracket_\ell) \\
\xrightarrow{no,lapp} & L_R^*[(((\mathtt{let}\ y = t_y\ \mathtt{in}\ t\, x_1)x_2)^E \cdots x_n)^e] & (t_1) \\
\xrightarrow{no,lapp,*} & L_R^*[\mathtt{let}\ y = t_y\ \mathtt{in}\ t\, x_1\, x_2 \cdots x_n] & (t_n = \llbracket Z' \rrbracket_\ell)
\end{array}$$

Wir betrachten nun die Zustandsbeschreibung Z' nach Anwendung des Zustandsübergangs und verfolgen wiederum die Entwicklung der zugehörigen minimalen

Zustandsbeschreibung über einige Schritte:

$$\begin{array}{ll} & \big\langle(\overrightarrow{x=t}, y=t_y),\ t,\ x_1 : \ldots x_n\big\rangle \quad (Z') \\ > & \big\langle(\overrightarrow{x=t}, y=t_y),\ t\,x_1 \ldots x_n,\ \varepsilon\big\rangle \quad (Z'_n) \end{array}$$

Aus Korollar 6.2.1 folgt $[\![Z']\!]_\ell = [\![Z'_n]\!]_\ell$. Ein Vergleich von Z_n und Z'_n zeigt, daß $[\![Z'_n]\!]_\ell$ und damit auch $[\![Z']\!]_\ell$ dem Ausdruck t_n entspricht.

Left und *Right*
Wir werden zeigen, daß jeder *Left*-Zustandsübergang einer $(no, nd, left)$-Reduktion und jeder *Right*-Zustandsübergang einer $(no, nd, right)$-Reduktion im Λ_{let}-Kalkül entspricht. Wir betrachten stellvertretend den Fall eines *Left*-Zustandsübergangs, der andere Fall kann nach demselben Schema gezeigt werden.

Die Argumentation erfolgt fast identisch wie beim *Share*-Zustandsübergang. Der *Left*-Zustandsübergang habe die Form $Z \xrightarrow{Left} Z'$ mit
$Z = \langle\Gamma, \texttt{choice}\ , x : S\rangle$ und $Z' = \langle\Gamma, \lambda y.\lambda z.y, x : S\rangle$. Wir wählen $Z'' = \langle\Gamma, \texttt{choice}\ x, S\rangle$, offensichtlich gilt $Z{>}Z''$. Aus Korollar 6.2.1 folgt nun $[\![Z]\!]_\ell = [\![Z'']\!]_\ell$ sowie die Existenz eines Kontexts C, so daß $[\![Z'']\!]_\ell = C[\texttt{choice}\ x]$ gilt. Lemma 6.2.3 besagt, daß bei Anwendung des no-Markierungsalgorithmus eine Markierung $C[(\texttt{choice}\ x)^E]$ erhalten wird. Somit liegt offensichtlich ein (no, nd)-Redex vor. Bei der Reduktion des no-Redex entscheiden wir uns für die Linksregel und erhalten den Ausdruck $C[(\lambda y.\lambda z.y)x]$, der genau $[\![\langle Z'\rangle]\!]_\ell$ entspricht, wie aus einer Gegenüberstellung von Z' und Z'' in Kombination mit Lemma 6.2.1 sofort ersichtlich ist. □

Korollar 6.2.2. *Gegeben zwei Zustandsbeschreibungen* Z *und* Z' *sowie zwei Ausdrücke* t *und* t' *mit* $[\![Z]\!]_\ell = t$ *und* $[\![Z']\!]_\ell = t'$. *Wenn* $Z \xrightarrow{*} Z'$, *dann gilt* $t \xrightarrow{no,*} t'$.

Beweis. Das Korollar ist eine einfache transitive Erweiterung von Lemma 6.2.4. □

Lemma 6.2.5. *Jede Folge* F *von Zustandsübergängen, die alleinig aus Lookup-, Unwind- und Let*$_\varepsilon$*-Zustandsübergängen besteht terminiert.*

Beweis. *Let*$_\varepsilon$-Zustandsübergängen können nur als geschlossene Gruppe am Anfang von F erscheinen, da sie alleinig bei leerem Stack anwendbar sind. Ein *Lookup*-, wie auch ein *Unwind*-Zustandsübergang, legen jedoch stets ein Element auf den Stack. Es können nur endlich viele *Let*$_\varepsilon$-Zustandsübergänge direkt nacheinander erfolgen, da jeder *Let*$_\varepsilon$-Zustandsübergang den mittleren Ausdruck einer Zustandsbeschreibung um eine *Let*-Bindung verkürzt.

Bleibt zu zeigen daß jede Folge von *Lookup*-, *Unwind*-Zustandsübergangen terminiert. Wir definieren uns dazu folgendes Maß für Zustandsbeschreibungen:

Gegeben eine Zustandsbeschreibung $\langle \Gamma, t, S\rangle$. Das Maß besteht aus zwei lexikographisch geordneten Komponenten:

1. Komponente: Die Anzahl der Bindungen $x = t_x$ des Heap Γ, bei denen t_x keine Update-Markierung ist.

2. Komponente: Die Anzahl an Applikationen die in dem Ausdruck t vorkommen.

Es ist offensichtlich, daß

- jeder *Lookup*-Zustandsübergang die erste Komponente verringert und die zweite Komponente vergrößert oder unverändert läßt,
- jeder *Unwind*-Zustandsübergang die erste Komponente unverändert läßt jedoch die zweite Komponente stets verkleinert.

Insgesamt terminiert somit die Folge F. □

Lemma 6.2.6. *Gegeben eine Zustandsbeschreibung Z sowie einen $\Lambda_{let}^{\ominus}$-Ausdruck t mit $[\![Z]\!]_\ell = t$. Wenn ausgehend von Z eine unendlich lange Folge F_∞ von Zustandsübergängen existiert, dann existiert ausgehend von t eine unendlich lange no-Reduktion.*

Beweis. Der Beweis ergibt sich aus der Kombination von Lemma 6.2.4 und Lemma 6.2.5. Gemäß Abbildung 6.3 erhält alleinig ein *Lookup*-, *Unwind*- oder *Let*$_\varepsilon$-Zustandsübergang nach Anwendung der Interpretation $[\![\ ]\!]_\ell$ die syntaktische Äquivalenz. Alle anderen Zustandsübergänge korrespondieren mit mindestens einer no-Reduktion. Gemäß Lemma 6.2.5 terminiert jede Folge, die alleinig aus *Lookup*-, *Unwind*- und *Let*$_\varepsilon$-Zustandsübergangen besteht. Somit muß eine unendliche Folge von Zustandsübergängen unendlich viele Zustandsübergänge enthalten die kein *Lookup*-, *Unwind*- oder *Let*$_\varepsilon$-Zustandsübergang sind. Jeder dieser Zustandsübergänge korrespondiert jedoch mit mindestens einer no-Reduktion. Daraus folgt, daß ausgehend von t eine unendliche no-Reduktion existiert. □

Wir betrachten nun die umgekehrte Richtung und zeigen, daß die no-Reduktion des Λ_{let}-Kalküls durch die abstrakte Maschine $M_{\Lambda_{let}}$ simuliert werden kann.

Lemma 6.2.7. *Gegeben ein Ausdruck $t \in \Lambda_{let}^{\ominus}$. Wenn $t \xrightarrow{no} t_1$, dann existiert ein Ausdruck $t_2 \in \Lambda_{let}^{\ominus}$ mit $t_1 \xrightarrow{no,*} t_2$, so daß die Zustandsbeschreibung $Z = \langle (),\ t,\ \varepsilon\rangle$ in eine Zustandsbeschreibung Z' überführt werden kann, für die $[\![Z']\!]_\ell = t_2$ gilt. Graphisch:*

$$\begin{tikzcd}
t \arrow[d, "{no,\cdot}"] & =_{\llbracket\,\rrbracket_\ell} & \langle((),t,\varepsilon\rangle \arrow[dd, dashed, "*"] \\
t_1 \arrow[d, dashed, "{no,*}"] & & \\
t_2 & =_{\llbracket\,\rrbracket_\ell} & Z'
\end{tikzcd}$$

Beweis. Wir untersuchen alle möglichen no-Reduktionen und greifen dabei auf die aus Lemma 2.3.3 bekannten Informationen über das Aussehen der Kontexte zurück, in denen die verschiedenen Arten von no-Redizes vorkommen:

1) (no, cp)-Reduktion
Die an der (no, cp)-Reduktion beteiligten Ausdrücke seien $t \equiv L_R^*[\texttt{let}\ x = t_x\ \texttt{in}\ C[x]]$ und $t_1 \equiv L_R^*[\texttt{let}\ x = t_x\ \texttt{in}\ C[t_x]]$ mit t_x eine Abstraktion oder die Konstante `choice` . Wir benutzen folgende Entwicklung ausgehend von dem Startzustand Z:

$$\begin{array}{rll}
& \langle (),\ L_R^*[\texttt{let}\ x = t_x\ \texttt{in}\ W[x]],\ \varepsilon\rangle & (Z) \\
\xrightarrow{Let_\varepsilon,*} & \big\langle (\overrightarrow{y = t_y}, x = t_x),\ W[x],\ \varepsilon\big\rangle & (Z_1) \\
\xrightarrow{*} & \big\langle (\overrightarrow{y = t_y}, x = t_x, \overrightarrow{z = t_z}),\ x,\ S\big\rangle & (Z_2),\ \text{Lemma 6.2.2} \\
\xrightarrow{Lookup} & \big\langle (\overrightarrow{y = t_y}, x = \#x, \overrightarrow{z = t_z}),\ t_x,\ \#x : S\big\rangle & (Z_3) \\
\xrightarrow{Update} & \big\langle (\overrightarrow{y = t_y}, x = t_x, \overrightarrow{z = t_z}),\ t_x,\ S\big\rangle & (Z_4)
\end{array}$$

Aus Korollar 6.2.1 sowie Lemma 6.2.2 folgt direkt $\llbracket Z\rrbracket_\ell = \llbracket Z_1\rrbracket_\ell = \llbracket Z_2\rrbracket_\ell = \llbracket Z_3\rrbracket_\ell$. Letzteres liefert in Kombination mit einer Gegenüberstellung der Zustandsbeschreibungen Z_2 und Z_4, daß $\llbracket Z_4\rrbracket_\ell = t_1$ gelten muß. Indem wir $t_2 \equiv t_1$ und $Z' \equiv Z_4$ wählen haben wir die Hypothese für diesen Fall gezeigt.

2) $(no, lbeta)$-Reduktion
Die Vorgehensweise ist dieselbe wie bei der (no, cp)-Reduktion. Die an der $(no, lbeta)$-Reduktion beteiligten Ausdrücke seien $t \equiv W[(\lambda x.s)y]$ und $t_1 \equiv W[\texttt{let}\ x = y\ \texttt{in}\ s]$. Wir benutzen folgende Entwicklung ausgehend von dem zugehörigen Startzustand Z:

$$\begin{array}{rll}
& \langle (),\ W[(\lambda x.s)y],\ \varepsilon\rangle & (Z) \\
\xrightarrow{*} & \langle \Gamma,\ (\lambda x.s)y,\ S\rangle & (Z_1),\ \text{Lemma 6.2.2} \\
\xrightarrow{Unwind} & \langle \Gamma,\ \lambda x.s,\ y : S\rangle & (Z_2) \\
\xrightarrow{Share} & \langle \Gamma,\ \texttt{let}\ x = y\ \texttt{in}\ s,\ S\rangle & (Z_3)
\end{array}$$

Aus Korollar 6.2.1 sowie Lemma 6.2.2 folgt direkt $\llbracket Z\rrbracket_\ell = \llbracket Z_1\rrbracket_\ell = \llbracket Z_2\rrbracket_\ell$. Letzteres liefert in Kombination mit einer Gegenüberstellung der Zustandsbeschreibungen

Z_1 und Z_3, daß $\llbracket Z_3 \rrbracket_\ell = t_1$ gelten muß. Indem wir $t_2 \equiv t_1$ und $Z' \equiv Z_3$ wählen, haben wir die Hypothese für diesen Fall gezeigt.

3) $(no, lapp)$-Reduktion
Die $(no, lapp)$-Reduktion gehe von dem Ausdruck $t \equiv W[(\texttt{let}\ x = t_x\ \texttt{in}\ s)y]$ aus. Wir benutzen folgende Entwicklung ausgehend von dem zugehörigen Z:

$$\begin{array}{rll}
 & \langle (),\ W[(\texttt{let}\ x = t_x\ \texttt{in}\ s)y],\ \varepsilon \rangle & (Z) \\
\xrightarrow{*} & \big\langle (\overrightarrow{z = t_z}),\ (\texttt{let}\ x = t_x\ \texttt{in}\ s)y,\ S \big\rangle & (Z_1),\ \text{Lemma 6.2.2} \\
\xrightarrow{Unwind} & \big\langle (\overrightarrow{z = t_z}),\ \texttt{let}\ x = t_x\ \texttt{in}\ s,\ y : S \big\rangle & (Z_2) \\
\xrightarrow{Let} & \big\langle (\overrightarrow{z = t_z}, x = t_x),\ s,\ y : S \big\rangle & (Z_3)
\end{array}$$

Gemäß Lemma 6.2.4 entspricht der von Z_2 ausgehende *Let*-Zustandsübergang einer nichtleeren Folge F von $(no, lapp)$-Reduktionen an deren Ende potentiell eine einzelne $(no, llet)$-Reduktion steht. Ferner gilt gemäß Lemma 6.2.2 und Korollar 6.2.1 $\llbracket Z \rrbracket_\ell = \llbracket Z_1 \rrbracket_\ell = \llbracket Z_2 \rrbracket_\ell$. Kombiniert impliziert dies, daß die erste $(no, lapp)$-Reduktion aus der Folge F, genau der $(no, lapp)$-Reduktion entsprechen muß, die von t ausgeht.
Als Z' wählen wir Z_3. Die Existenz eines Ausdrucks t_2 mit $t_1 \xrightarrow{no,*} t_2$ und $t_2 = \llbracket Z_3 \rrbracket_\ell = t_2$ zeigt Lemma 6.2.4.

Bei der obigen Argumentation ist beachtenswert, daß die Position des $(no, lapp)$-Redex nicht anhand des W-Kontext fixiert wird, sondern vielmehr nur von Interesse ist, daß jeder W-Kontext zunächst durch eine Folge von Zustandsübergängen auf Stack und Heap bewegt wird.
Bei allen anderen no-Reduktionen kann eine Argumentation nach demselben Schema wie bei der $(no, lapp)$-Reduktion erfolgen.

4) $(no, llet)$-Reduktion
Die $(no, llet)$-Reduktion gehe von dem Ausdruck $t \equiv L_R^*[\texttt{let}\ x = (\texttt{let}\ y = t_y\ \texttt{in}\ t_x)\ \texttt{in}\ W[x]]$ aus. Wir benutzen folgende Entwicklung ausgehend von dem Startzustand Z:

$$\begin{array}{rll}
 & \langle (),\ L_R^*[\texttt{let}\ x = (\texttt{let}\ y = t_y\ \texttt{in}\ t_x)\ \texttt{in}\ W[x]],\ \varepsilon \rangle & (Z) \\
\xrightarrow{Let_\varepsilon,*} & \big\langle (\overrightarrow{z = t_z}),\ \texttt{let}\ x = (\texttt{let}\ y = t_y\ \texttt{in}\ t_x)\ \texttt{in}\ W[x],\ \varepsilon \big\rangle & (Z_1) \\
\xrightarrow{Let_\varepsilon} & \big\langle (\overrightarrow{z = t_z}, x = (\texttt{let}\ y = t_y\ \texttt{in}\ t_x)),\ W[x],\ \varepsilon \big\rangle & (Z_2) \\
\xrightarrow{*} & \Big\langle (\overrightarrow{z = t_z}, x = (\texttt{let}\ y = t_y\ \texttt{in}\ t_x), \overrightarrow{z' = t'_z}),\ x,\ S \Big\rangle & (Z_3),\ \text{Lemma 6.2.2} \\
\xrightarrow{Lookup} & \Big\langle (\overrightarrow{z = t_z}, x = \#x, \overrightarrow{z' = t'_z}),\ (\texttt{let}\ y = t_y\ \texttt{in}\ t_x),\ \#x : S \Big\rangle & (Z_4) \\
\xrightarrow{Let} & \Big\langle (\overrightarrow{z = t_z}, y = t_y, x = \#x, \overrightarrow{z' = t'_z}),\ t_x,\ \#x : S \Big\rangle & (Z_5)
\end{array}$$

Gemäß Lemma 6.2.4 entspricht der von Z_4 ausgehende *Let*-Zustandsübergang einer einzelnen $(no, llet)$-Reduktion. (Beachte, an der Spitze des Stack befinden sich keine Variablen.) Ferner gilt gemäß Lemma 6.2.2 und Korollar 6.2.1, daß die Interpretation aller Zustandsbeschreibungen außer Z_5 dem Ausdruck $\llbracket Z \rrbracket_\ell$ entspricht. Kombiniert impliziert dies, daß der von Z_4 ausgehende *Let*-Zustandsübergang genau der einzelnen von t ausgehenden $(no, llet)$-Reduktion entspricht. Als Z' wählen wir Z_5. Die Existenz eines Ausdrucks t_2 mit $t_1 \xrightarrow{no,*} t_2$ und $t_2 = \llbracket Z_5 \rrbracket_\ell = t_2$ ergibt sich wie im $(no, lapp)$-Fall direkt aus Lemma 6.2.4.

5) $(no, nd, left)$*-Reduktion*
Die $(no, nd, left)$-Reduktion gehe von dem Ausdruck $t \equiv W[\texttt{choice}\ x]$ aus. Wir benutzen folgende Entwicklung ausgehend von dem Startzustand Z:

$$\begin{array}{rll}
 & \langle (),\ W[\texttt{choice}\ x],\ \varepsilon \rangle & (Z) \\
\xrightarrow{*} & \langle \Gamma,\ \texttt{choice}\ x,\ S \rangle & (Z_1),\ \text{Lemma 6.2.2} \\
\xrightarrow{Unwind} & \langle \Gamma,\ \texttt{choice}\ ,\ x : S \rangle & (Z_2) \\
\xrightarrow{Left} & \langle \Gamma,\ \lambda y.\lambda z.y,\ x : S \rangle & (Z_3)
\end{array}$$

Gemäß Lemma 6.2.4 entspricht der von Z_2 ausgehende $Left$-Zustandsübergang einer einzelnen $(no, nd, left)$-Reduktion. Ferner gilt gemäß Lemma 6.2.2 und Korollar 6.2.1, $\llbracket Z \rrbracket_\ell = \llbracket Z_2 \rrbracket_\ell$. Kombiniert impliziert dies, daß der von Z_2 ausgehende $Left$-Zustandsübergang genau der einzelnen von t ausgehenden $(no, nd, left)$-Reduktion entspricht. Als Z' wählen wir Z_3. Die Existenz eines Ausdrucks t_2 mit $t_1 \xrightarrow{no,*} t_2$ und $t_2 = \llbracket Z_3 \rrbracket_\ell = t_2$ ergibt sich wie bei den beiden vorherigen Fällen direkt aus Lemma 6.2.4.

Bei einer $(no, nd, right)$-Reduktion erfolgt eine simultane Argumentation unter Zuhilfenahme eines *Right*-Zustandsübergangs. □

Lemma 6.2.8. *Gegeben zwei Zustandsbeschreibungen* Z_1 *und* Z_1' *mit* $\llbracket Z_1 \rrbracket_\ell = \llbracket Z_1' \rrbracket_\ell$. *Wenn* $Z_1 \xrightarrow{*} Z_2$, *dann existiert eine Zustandsbeschreibung* Z_2', *so daß* $Z_1' \xrightarrow{*} Z_2'$ *und* $\llbracket Z_2 \rrbracket_\ell = \llbracket Z_2' \rrbracket_\ell$ *gilt. Graphisch:*

$$\begin{tikzcd}
Z_1 \arrow[r, phantom, "=_{\llbracket\,\rrbracket_\ell}"] \arrow[d, "*"] & Z_1' \arrow[d, dashed, "*"] \\
Z_2 \arrow[r, phantom, "=_{\llbracket\,\rrbracket_\ell}"] & Z_2'
\end{tikzcd}$$

Beweis. Gemäß Korollar 6.2.1 impliziert $\llbracket Z_1 \rrbracket_\ell = \llbracket Z_1' \rrbracket_\ell$ eine der drei Alternativen $Z_1 < Z_1'$, $Z_1' < Z_1$ oder $Z_1 = Z_1'$. Für den Fall $Z_1 = Z_1'$ ist der Beweis der Hypothese trivial.

$Z_1 < Z_1'$
Gemäß Korollar 6.2.1 gilt dann $Z_1 \xrightarrow{+} Z_1'$. Aus der Singularität der Folge von Zustandsübergängen von Z_1 nach Z_1' ergibt sich, daß entweder $Z_1' \xrightarrow{*} Z_2$ oder

$Z_2 \xrightarrow{*} Z_1'$ gelten muß. Im ersten Fall wählen wir $Z_2' = Z_2$. Im zweiten Fall wählen wir $Z_2' = Z_1'$. Die Korrektheit der Wahl kann in beiden Fällen mittels der Aussagen von Korollar 6.2.1 leicht überprüft werden. Wesentlich ist dabei, daß beim zweiten Fall beim Übergang von Z_2 zu Z_1 nur *Unwind* -, *Lookup*- und Let_ε-Zustandsübergänge vorkommen, die die Gleichheit der Interpretation stets bewahren.

$Z_1' < Z_1$
Gemäß Korollar 6.2.1 gilt dann $Z_1' \xrightarrow{+} Z_1$. Wir wählen $Z_2' = Z_2$, die Korrektheit der Wahl ist offensichtlich. □

Wir beweisen nun eine erweiterte Version von Lemma 6.2.7, bei der die gegebene no-Reduktion beliebige Länge besitzen darf.

Lemma 6.2.9. <L9-4>*Gegeben ein Ausdruck* $t \in \Lambda_{let}^{\ominus}$. *Wenn* $t \xrightarrow{no,*} t_1$, *dann existiert ein Ausdruck* $t_2 \in \Lambda_{let}^{\ominus}$ *sowie eine Zustandsbeschreibung* Z', *so daß* $\langle (),\ t,\ \varepsilon\rangle \xrightarrow{*} Z'$, $t_1 \xrightarrow{no,*} t_2$ *und* $[\![Z']\!]_\ell = t_2$ *gilt. Graphisch:*

$$\begin{tikzcd}
t \arrow[r, phantom, "=_{[\![\,]\!]_\ell}"] \arrow[d, "{no,*}"] & \langle (), t, \varepsilon \rangle \arrow[dd, dashed, "*"] \\
t_1 \arrow[d, dashed, "{no,*}"] & \\
t_2 \arrow[r, phantom, "=_{[\![\,]\!]_\ell}"] & Z'
\end{tikzcd}$$

Beweis. Wir führen eine Induktion über die Menge aller der Länge nach aufsteigend geordneten no-Reduktion. Als Induktionshypothese benutzen wir die Hypothese des Lemmas.

Induktionsbeginn, $t \equiv t_1$
Dieser Fall ist trivial, wir wählen $t_2 \equiv t_1$.

Induktionsschritt, $t \xrightarrow{no,\cdot} t_3 \xrightarrow{no,*} t_1$
Gemäß Lemma 6.2.7 existiert ein Ausdruck t_4 mit $t_3 \xrightarrow{no,*} t_4$, so daß die Zustandsbeschreibung $\langle (),\ t,\ \varepsilon\rangle$ in eine Zustandsbeschreibung Z_1 überführt werden kann, für die $[\![Z_1]\!]_\ell = t_4$ gilt. Dabei beinhaltet die no-Reduktion von t_3 nach t_4 keine (no, nd)-Reduktionen, was im Bezug auf die Eindeutigkeit der no-Reduktion wesentlich ist. Somit sind die zwei Fälle $t_4 \xrightarrow{no,*} t_1$ und $t_1 \xrightarrow{no,*} t_4$ zu unterscheiden. Im ersten Fall folgt aus der Induktionsvoraussetzung die Existenz eines Ausdrucks t_5 sowie einer Zustandsbeschreibung Z_2, so daß $\langle (),\ t_4,\ \varepsilon\rangle \xrightarrow{*} Z_2$, $t_4 \xrightarrow{no,*} t_5$ und $[\![Z_2]\!]_\ell = t_5$ gilt. Da zusätzlich $[\![\langle (),\ t_4,\ \varepsilon\rangle]\!]_\ell = [\![Z_1]\!]_\ell$ gilt, folgt aus Lemma 6.2.8 die Existenz einer Zustandsbeschreibung Z_3, so daß $Z_1 \xrightarrow{*} Z_3$ und $[\![Z_2]\!]_\ell = [\![Z_3]\!]_\ell$ gilt. Wir wählen nun $t_2 \equiv t_5$ sowie $Z' = Z_3$ und haben die Existenz der gesuchten Elemente gezeigt.

Im zweiten Fall wählen wir $t_2 \equiv t_4$ und $Z' = Z_1$; die Korrektheit der Wahl ist offensichtlich. □

Wir werden nun zeigen, daß ein direkter Bezug zwischen den Begriffen WHNF im Λ_{let}-Kalkül und Endzustand bei der abstrakten Maschine besteht.

Lemma 6.2.10. *Jeder Endzustand wird mittels der Interpretation* $[\![\ ]\!]_\ell$ *auf einen Ausdruck in WHNF abgebildet.*

Beweis. Für jeden Endzustand $Z_F = \langle \Gamma, r, \varepsilon\rangle$ gilt $[\![Z_F]\!]_\ell = L_R^*[r]$, wobei r eine Abstraktion oder die Konstante `choice` . Letzteres folgt direkt aus der Definition der Interpretation $[\![\ ]\!]_\ell$. Gemäß Lemma 2.3.1 befindet sich jeder Ausdruck der Form $L_R^*[r]$ mit r eine Abstraktion oder die Konstante `choice` in WHNF. □

Lemma 6.2.11. *Gegeben ein geschlossener Ausdruck* $t \in \Lambda_{let}^{\ominus}$. *Wenn* t *in WHNF, dann existiert ein Endzustand* Z_f, *so daß jede Zustandsbeschreibung* Z *mit* $[\![Z]\!]_\ell = t$ *die Eigenschaft* $Z \xrightarrow{Let_\varepsilon,*} Z_f$ *erfüllt.*

Beweis. t in WHNF impliziert gemäß Lemma 2.3.1, daß t die Form $L_R^*[r]$ mit r eine Abstraktion oder die Konstante `choice` besitzt. Im weiteren sei Z_t die Zustandsbeschreibung $\langle (), t, \varepsilon\rangle$. Aus der Form des Ausdrucks t folgt, daß Z_t mittels einer singulären Folge F von Let_ε-Zustandsübergängen in einen Endzustand $Z_f = \langle \Gamma, r, \varepsilon\rangle$ überführt werden kann. Gemäß Korollar 6.2.1 gilt für alle Zustandsbeschreibungen Z mit $[\![Z]\!]_\ell = t$ $(= [\![Z_t]\!]_\ell)$ die Eigenschaft $Z_t < Z$ bzw. $Z_t = Z$. Der Fall $Z < Z_t$ ist nicht möglich, da Z_t minimal ist. Daraus folgt unter nochmaliger Anwendung von Korollar 6.2.1, daß Z_t mittels einer Folge von Zustandsübergängen in Z überführt werden kann. Da die Folge F singulär ist, also weder $Left$- noch $Right$ - Zustandsübergänge beinhaltet, muß auf dem Weg von Z_t zu Z_f irgendwann Z besucht werden. Damit ist aber zugleich gezeigt, daß ausgehend von Z eine Folge von Let_ε-Zustandsübergängen zu dem Endzustand Z_f existiert. □

Wir werden nun zunächst verschiedene modellangepaßte Varianten der kontextuellen Äquivalenz definieren. Im Anschluß daran werden die zuletzt bewiesenen Lemmata dazu benutzen eine gdw.-Beziehung zwischen den neu definierten Varianten der kontextuellen Äquivalenz zu zeigen.

Definition 6.2.4. *Ein* $C^{\ominus}$*-Kontext ist ein* $\Lambda_{let}^{\ominus}$*-Ausdruck mit einem Loch* $[\cdot]$ *und folgenden syntaktischen Bildungsgesetzen:*

$$C^{\ominus} := [\cdot] \mid \lambda x.C^{\ominus} \mid C^{\ominus}x \mid \texttt{let}\ x = s\ \texttt{in}\ C^{\ominus} \mid \texttt{let}\ x = C^{\ominus}\ \texttt{in}\ s$$

s entspricht dabei stets einem $\Lambda_{let}^{\ominus}$*-Ausdruck.* $C^{\ominus}[t]$ *entspricht dem Ausdruck, der erhalten wird, wenn das Loch* $[\cdot]$ *des Kontexts* $C^{\ominus}$ *durch den Ausdruck* t *ersetzt wird.*

Die Menge aller $C^{\ominus}$-Kontexte ist somit eine spezielle Kontextklasse, die den Anforderungen bezüglich der Einschränkung bei $\Lambda_{let}^{\ominus}$-Ausdrücken gerecht wird, daß der rechte Teilausdruck jeder Applikation eine Variable sein muß. Wir formulieren nun einen der Klasse der $C^{\ominus}$-Kontexte angepaßten Begriff der kontextuellen Äquivalenz.

Definition 6.2.5. *s, t seien zwei $\Lambda_{let}^{\ominus}$-Ausdrücke. s und t sind kontextuell äquivalent bezüglich der Klasse der $C^{\ominus}$-Kontexte, bezeichnet durch $s \sim_{c,\ominus} t$ gdw. für alle Kontexte $C^{\ominus}$ mit der Eigenschaft, daß $C^{\ominus}[s]$ und $C^{\ominus}[t]$ geschlossen ist, gilt*

$$C^{\ominus}[s]\Downarrow \iff C^{\ominus}[t]\Downarrow \quad \wedge \quad C^{\ominus}[s]\Uparrow \iff C^{\ominus}[t]\Uparrow$$

Wir dehnen nun den Begriff der kontextuellen Äquivalenz auf die Menge der Zustandsbeschreibungen der Maschine $M_{\Lambda_{let}}$ aus. Wir definieren dazu zunächst die Begriffe Konvergenz und Divergenz für Zustandsbeschreibungen.

Definition 6.2.6. *Sei Z eine beliebige geschlossene Zustandsbeschreibung.*

$Z\Downarrow_M$	$\Leftrightarrow_{Def}$	*es existiert ein Endzustand Z_f, so daß $Z \xrightarrow{*} Z_f$*	*(Konvergenz)*
$Z\Uparrow_M$	$\Leftrightarrow_{Def}$	*ausgehend von Z existiert eine nicht terminierende Folge einzelner Zustandsübergänge*	*(Divergenz)*

Die zuletzt definierte Konvergenz und Divergenz wird nun benutzt um eine auf die Maschine $M_{\Lambda_{let}}$ abgestimmte kontextuelle Äquivalenz zu formulieren.

Definition 6.2.7. *Seien s, t zwei $\Lambda_{let}^{\ominus}$-Ausdrücke. s und t sind kontextuell äquivalent bezüglich der Maschine $M_{\Lambda_{let}}$, bezeichnet durch $s \sim_{c,M} t$, gdw. für alle Kontexte $C^{\ominus}$ mit der Eigenschaft, daß $C^{\ominus}[s]$ und $C^{\ominus}[t]$ geschlossen ist, gilt*

$$\begin{array}{lcl} & \langle (), C^{\ominus}[s], \varepsilon\rangle\Downarrow_M \iff \langle (), C^{\ominus}[t], \varepsilon\rangle\Downarrow_M \\ \wedge & \langle (), C^{\ominus}[s], \varepsilon\rangle\Uparrow_M \iff \langle (), C^{\ominus}[t], \varepsilon\rangle\Uparrow_M \end{array}$$

Wir sind nun in der Lage mittels der verschiedenen Varianten der kontextuellen Äquivalenz zu zeigen, daß zur Evaluation eines $\Lambda_{let}^{\ominus}$-Ausdrucks der Λ_{let}-Kalkül und die abstrakte Maschine austauschbar sind, d.h. beide dasselbe Terminierungsverhalten zeigen.

Satz 6.2.1. *Gegeben ein Ausdruck $t \in \Lambda_{let}^{\ominus}$. Für alle Kontexte $C^{\ominus}$ mit der Eigenschaft, daß $C^{\ominus}[t]$ geschlossen, gilt*

$$\begin{array}{lcl} & C^{\ominus}[t]\Downarrow \iff \langle (), C^{\ominus}[t], \varepsilon\rangle\Downarrow_M \\ \wedge & C^{\ominus}[t]\Uparrow \iff \langle (), C^{\ominus}[t], \varepsilon\rangle\Uparrow_M \end{array}$$

Beweis. Da $C^{\ominus}[t]$ stets zugleich ein $\Lambda^{\ominus}_{let}$-Ausdruck t' eingesetzt in den leeren Kontext $[\cdot]$ ist, reicht es, die geforderten Eigenschaften für alle $\Lambda^{\ominus}_{let}$-Ausdrücke zu zeigen.

$\Rightarrow$
Gegeben seien zwei Ausdrücke t und t' aus $\Lambda^{\ominus}_{let}$ mit $t \xrightarrow{no,*} t'$ und t' in WHNF. Gemäß Lemma 6.2.9 existiert ein Ausdruck t'' mit $t' \xrightarrow{no,*} t''$ und eine Zustandsbeschreibung Z, so daß $\langle (), t, \varepsilon \rangle \xrightarrow{*} Z$ und $[\![Z]\!]_\ell = t''$ gilt. Da t' bereits in WHNF ist und somit auf t' keine weiteren no-Reduktionen angewendet werden können, muß $t'' \equiv t'$ gelten. Eine nun folgende Anwendung von Lemma 6.2.11 liefert die Existenz eines Endzustands Z_f, für den $Z \xrightarrow{Let_\varepsilon,*} Z_f$ gilt. Damit ist gezeigt, daß ausgehend von der Zustandsbeschreibung $\langle (), t, \varepsilon \rangle$ eine Folge von Zustandsübergängen zu einem Endzustand existiert.
Gemäß Korollar 2.4.3 enthält jede unendliche no-Reduktion unendlich viele (no, cp)-Reduktionen sowie unendlich viele $(no, lbeta)$-Reduktionen. Dem Beweis von Lemma 6.2.7 kann entnommen werden, daß die Simulation einer (no, cp)-Reduktion stets einen *Update*-Zustandsübergang erfordert bzw. die Simulation einer $(no, lbeta)$-Reduktion stets einen *Share*-Zustandsübergang erfordert. Lemma 6.2.9 greift bei seiner induktiven Argumentation direkt auf Lemma 6.2.7 zurück. Dies ermöglicht folgende Argumentation:
Indem schrittweise eine immer mehr (no, cp)- und $(no, lbeta)$-Reduktionen umfassende no-Reduktion ausgehend von t gewählt wird, kann mittels der oben beschriebenen Übertragungseigenschaft und Lemma 6.2.9 eine immer längere Folge von Zustandsübergängen ausgehend von der Zustandsbeschreibung $\langle (), t, \varepsilon \rangle$ konstruiert werden. Damit kann mittels Königs Lemma aus der Existenz einer unendlichen no-Reduktionen ausgehend von t die Existenz einer unendlichen Folge von Zustandsübergängen ausgehend von $\langle (), t, \varepsilon \rangle$ gefolgert werden.

$\Leftarrow$
Existiert ausgehend von einer Zustandsbeschreibung $\langle (), t, \varepsilon \rangle$ eine Folge von Zustandsübergängen zu einem Endzustand Z_f, so befindet sich der Ausdruck $[\![Z_f]\!]_\ell$ gemäß Lemma 6.2.10 in WHNF.

Die Existenz einer unendlichen no-Reduktionen bei einer gegebenen unendlich langen Folge von Zustandsübergängen wird in Lemma 6.2.6 gezeigt. □

Korollar 6.2.3. *Seien s, t zwei $\Lambda^{\ominus}_{let}$-Ausdrücke. Dann*

$$s \sim_{c,\ominus} t \;\Leftrightarrow\; s \sim_{c,M} t$$

Beweis. Beide Richtungen sind jeweils eine einfache Entwicklung für Konvergenz und Divergenz unter Zuhilfenahme des obigen Lemmas:

$\Leftarrow$
$C^{\ominus}[s]\Downarrow \Leftrightarrow \langle (), C^{\ominus}[s], \varepsilon \rangle \Downarrow_M \Leftrightarrow \langle (), C^{\ominus}[t], \varepsilon \rangle \Downarrow_M \Leftrightarrow C^{\ominus}[t]\Downarrow$
$C^{\ominus}[s]\Uparrow \Leftrightarrow \langle (), C^{\ominus}[s], \varepsilon \rangle \Uparrow_M \Leftrightarrow \langle (), C^{\ominus}[t], \varepsilon \rangle \Uparrow_M \Leftrightarrow C^{\ominus}[t]\Uparrow$

$\Rightarrow$
Erfolgt genauso wie oben, nur daß $s \sim_c t$ als Voraussetzung benutzt wird. □

Das obige Korollar formuliert nur eine Aussage bezüglich der Klasse der $C^{\ominus}$-Kontexte. In diesem Abschnitt soll jedoch auch gezeigt werden, daß die abstrakten Maschine $M_{\Lambda_{let}}$ und der no-Reduktion des Λ_{let}-Kalküls bezüglich beliebiger Kontexte dasselbe Terminierungsverhalten aufweist. Eine solche Ausdehnung wird nun gezeigt. Dazu wird zunächst eine Abbildung definiert, die eine Beziehung zwischen der Menge Λ_{let}- und $\Lambda_{let}^{\ominus}$-Ausdrücke formuliert.

Definition 6.2.8. *Wir definieren die Abbildung τ, die jedem Ausdruck aus Λ_{let} einen Ausdruck aus $\Lambda_{let}^{\ominus}$ zuordnet wie folgt:*

$$\begin{array}{rcl}
\tau(v) & := & v \quad \textit{wobei } v \textit{ eine Variable oder die Konstante } \texttt{choice} \\
\tau(\lambda x.t) & := & \lambda x.\tau(t) \\
\tau(st) & := & \begin{cases} \tau(s)\, t & \text{falls } t \text{ eine Variable} \\ \texttt{let } x = \tau(t) \texttt{ in } \tau(s)\, x & \text{sonst, wobei } x \text{ eine frische Variable} \end{cases} \\
\tau(\texttt{let } x = t_x \texttt{ in } t) & := & \texttt{let } x = \tau(t_x) \texttt{ in } \tau(t)
\end{array}$$

Die Abbildung τ überführt jeden Λ_{let}-Ausdruck in eine Form, bei der auf der rechten Seite jeder Applikation eine Variable vorkommt. Wir benötigen diese Abbildung um eine Beziehung zwischen dem Λ_{let}-Kalkül und der abstrakten Maschine formulieren zu können, da letztere nur die auf $\Lambda_{let}^{\ominus}$ eingeschränkte Klasse von Ausdrücken verarbeiten kann.

Lemma 6.2.12. *Gegeben zwei geschlossene Ausdrücke t und s. Wenn $t \equiv \tau(s)$, dann gilt $t \xrightarrow{\{ucp,ldel\},*} s$.*

Beweis. Die Abbildung τ schachtelt um jede Applikation, bei der der rechte Subausdruck keine Variable ist, einen Let-Ausdruck, wobei die dabei neu hinzukommende Variable singulär vorkommt und durch eine *ucp*-Reduktion kopierbar ist, da sich das Kopierziel nicht unterhalb einer Abstraktion befindet. Deshalb kann jede so neu hinzukommende Let-Schachtelung durch Anwendung einer *ucp*- gefolgt von einer *ldel*-Reduktion wieder entfernt und der Ursprungsausdruck zurückerhalten werden. Die einzelnen Paare, bestehend aus einer *ucp*- und *ldel*-Reduktion, sind dabei voneinander unabhängig und können in beliebiger Reihenfolge angewendet werden. □

Lemma 6.2.13. *Für jeden geschlossenen Ausdruck t gilt $t \sim_c \tau(t)$.*

Beweis. Gemäß Lemma 6.2.12 gilt $\tau(t) \xrightarrow{\{ucp,ldel\},*} t$. Jede *ucp*- und jede *ldel*-Reduktion erhält gemäß Satz 3.4.5 und Satz 3.4.1 die kontextuelle Äquivalenz. Daraus folgt in Kombination mit der Symmetrie und Transitivität der kontextuellen Äquivalenz die Hypothese. □

Korollar 6.2.4. *Gegeben ein geschlossener Ausdruck s aus Λ_{let}. Für alle Kontexte C gilt:*

- $C[s]\Downarrow \iff \tau(C[s])\Downarrow$
- $C[s]\Uparrow \iff \tau(C[s])\Uparrow$

Beweis. Da $C[s]$ stets selbst ein Λ_{let}-Ausdruck ist, gilt für alle Kontexte C gemäß 6.2.13 Lemma $C[s] \sim_c \tau(C[s])$. Die Hypothese entspricht der Aussage, die $C[s] \sim_c \tau(C[s])$ für den leeren Kontext $[\cdot]$ formuliert. □

Lemma 6.2.14. *Gegeben zwei geschlossene Ausdrücke t und s aus Λ_{let}. Es gilt:*

$$t \sim_c s \iff \tau(t) \sim_c \tau(s)$$

Beweis.

$\Rightarrow$

Aus Lemma 6.2.13 ist bekannt, daß $t \sim_c \tau(t)$ und $s \sim_c \tau(s)$ gilt. Daraus folgt in Kombination mit $t \sim_c s$ sowie der Symmetrie und Transitivität von $\sim_c$, daß $\tau(t) \sim_c \tau(s)$ gilt.

$\Leftarrow$

Die Argumentation erfolgt genauso, nur daß wir bei dieser Richtung aus der Gültigkeit von $\tau(t) \sim_c \tau(s)$ auf die Gültigkeit von $t \sim_c s$ schließen. □

Das obige Lemma zeigt, daß zwei Ausdrücke nach Anwendung der Abbildung τ im Λ_{let}-Kalkül kontextuell äquivalent bleiben und umgekehrt. Eine Übertragbarkeit auf die Menge der $\Lambda_{let}^{\ominus}$-Ausdrücke, wie im Zusammenhang mit der abstrakten Maschine benötigt, scheitert jedoch an der Allgemeinheit der umschließenden Kontexte aus der Definition der kontextuellen Äquivalenz $\sim_c$.

Mit Hilfe von Definition 6.2.5 können wir die kontextuelle Äquivalenz zwischen der abstrakten Maschine $M_{\Lambda_{let}}$ und des Λ_{let}-Kalküls wechselseitig übertragen.

Satz 6.2.2. *s, t seien zwei Ausdrücke aus Λ_{let}. Dann*

$$s \sim_c t \iff \tau(s) \sim_{c,\ominus} \tau(t)$$

Beweis.

$\Rightarrow$

Gemäß Lemma 6.2.14 gilt $s \sim_c t \;\Rightarrow\; \tau(s) \sim_c \tau(t)$. Da jeder $C^{\ominus}$ -Kontexte gleichzeitig ein C-Kontext ist, gilt $\tau(s) \sim_c \tau(t) \;\Rightarrow\; \tau(s) \sim_{c,\ominus} \tau(t)$.

$\Leftarrow$
Gegeben seien zwei Ausdrücke s, t mit $\tau(s) \sim_{c,\ominus} \tau(t)$. Bei allen Kontexten C läßt sich folgende Argumentation anwenden:

$\tau(C)$ ist stets ein $C^{\ominus}$-Kontext. Gemäß Voraussetzung gilt somit:

$\tau(C)[\tau(s)]\Downarrow \Leftrightarrow\; \tau(C)[\tau(t)]\Downarrow \wedge \tau(C)[\tau(s)]\Uparrow \Leftrightarrow\; \tau(C)[\tau(t)]\Uparrow$

Wir wenden eine offensichtliche Zerlegbarkeit der Abbildung τ an und erhalten:

$\tau(C)[\tau(s)] = \tau(C[s])$ und $\tau(C)[\tau(t)] = \tau(C(t))$

Wir benutzen Korollar 6.2.4 und erhalten:

$\tau(C[s])\Downarrow \Leftrightarrow\; C[s]\Downarrow$ und $\tau(C[t])\Downarrow \Leftrightarrow\; C[t]\Downarrow$ sowie
$\tau(C[s])\Uparrow \Leftrightarrow\; C[s]\Uparrow$ und $\tau(C[t])\Uparrow \Leftrightarrow\; C[t]\Uparrow$

Die Hypothese ergibt sich nun aus den Zusammenfassungen:

$C[s]\Downarrow \Leftrightarrow\; \tau(C[s])\Downarrow \Leftrightarrow\; \tau(C[t])\Downarrow \Leftrightarrow\; C[t]\Downarrow$ sowie
$C[s]\Uparrow \Leftrightarrow\; \tau(C[s])\Uparrow \Leftrightarrow\; \tau(C[t])\Uparrow \Leftrightarrow\; C[t]\Uparrow$ □

Das Ziel dieses Abschnitts, eine Gleichheit im Terminierungsverhalten der abstrakten Maschine $M_{\Lambda_{let}}$ und des Λ_{let}-Kalküls zu beweisen, ist in der wechselseitigen Übertragbarkeit der kontextuellen Äquivalenz enthalten, die untenstehendes Lemma zeigt.

Korollar 6.2.5. *Seien s, t zwei $\Lambda^{\ominus}_{let}$-Ausdrücke. Dann*

$$s \sim_c t \quad\Longleftrightarrow\quad \tau(s) \sim_{c,\ominus} \tau(t) \quad\Longleftrightarrow\quad \tau(s) \sim_{c,M} \tau(t)$$

6.3 $M^{\oplus}_{\Lambda_{let}}$ - eine Variante der abstrakten Maschine $M_{\Lambda_{let}}$

Bei der zuvor betrachteten abstrakten Maschine $M_{\Lambda_{let}}$ wird der Heap als Folge einzelner Bindungen aufgefaßt. Die allgemein mit dem Begriff Heap assoziierten Struktur, von einem Container, zu dem Objekte hinzugefügt bzw. aus dem Objekte entnommen werden können, beinhaltet jedoch keine Reihenfolge der Objekte auf dem Heap. Die spezielle Form des Heaps der Maschine $M_{\Lambda_{let}}$ wurde, wie bereits früher erwähnt, im Zusammenhang mit den Beweisen der letzten Abschnitte eingeführt.
Es stellt sich nun die Frage, ob nicht ein Maschinenmodell existiert, das das gleiche Terminierungsverhalten wie die Maschine $M_{\Lambda_{let}}$ aufweist, dabei aber über

einen "echten" Heap verfügt, d.h. ohne die Eigenschaft auskommt, daß die Elemente auf dem Heap geordnet vorkommen. In diesem Abschnitt wird ein solches Maschinenmodell mit dem Namen $M^{\oplus}_{\Lambda_{let}}$ vorgestellt und die Gleichheit des Terminierungsverhaltens von dem neuen Modell und $M_{\Lambda_{let}}$ gezeigt.

Wir benötigen zunächst eine leicht veränderte Form der Definition des Begriffs Zustandsbeschreibung.

Definition 6.3.1. *Eine Zustandbeschreibung der abstrakten Maschine* $M^{\oplus}_{\Lambda_{let}}$ *hat die Form* $\langle \Gamma, t, S\rangle$, *wobei* Γ *den Heap,* t *den zu evaluierenden Ausdruck und* S *den Stack bezeichnet. Für den zu evaluierenden Ausdruck* t *und den Stack* S *gelten die für die Maschine* $M_{\Lambda_{let}}$ *vereinbarten Eigenschaften.* $\mathrm{dom}(S)$ *liefert die Menge aller Variablen* x *für die auf dem Stack* S *eine Update-Markierung* $\#x$ *vorkommt. Der Heap* Γ *besteht aus einer Menge von Bindungen der Form* $x = t_x$ *wobei* x *eine Variable und* t_x *einen* $\Lambda^{\ominus}_{let}$*-Ausdruck bezeichnet. Die Schreibweise* $\Gamma\{x = t_x\}$ *bezeichnet den Heap* Γ *erweitert um die Bindung der Variablen* x *an den Ausdruck* t_x. *Eine Gruppe von Bindungen kann durch* $\overrightarrow{x = t_x}$ *spezifiziert werden. Dem entsprechend bezeichnet* $\Gamma\{\overrightarrow{x = t_x}\}$ *den Heap* Γ *erweitert um alle Bindungen aus* $\overrightarrow{x = t_x}$. *Der leere Heap wird durch das Symbol* $\emptyset$ *bezeichnet.* $\mathrm{dom}(\Gamma)$ *liefert die Menge aller Variablen* x, *für die der Heap* Γ *eine Bindung* $x = t_x$ *formuliert.*

Eine Zustandsbeschreibung der Maschine $M^{\oplus}_{\Lambda_{let}}$ *ist gültig, wenn sie zusätzlich folgende Eigenschaften erfüllt:*

1. *Alle Bindungen auf dem Heap werden durch verschiedene Variablen hergestellt. Dasselbe gilt für alle Ausdrücke, die auf dem Heap vorkommen, sowie bei dem zu evaluierenden Ausdruck. Alle Update-Markierungen auf dem Stack sind verschieden und* $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(S) = \emptyset$ *gilt.*
2. *Die Bindungen des Heap* Γ *sind zyklenfrei. Existiert eine Zerlegung des Stacks* $S = T'\#x : T$, *so darf die Variable* x *weder in* T' *noch in dem zu evaluierenden Ausdruck* t *vorkommen.*

Zustandsbeschreibungen der abstrakten Maschine $M^{\oplus}_{\Lambda_{let}}$ *werden durch das Symbol* $Z^{\oplus}$ *zuzüglich möglicher Indizes bezeichnet.*

Eine Zustandsbeschreibung $Z^{\oplus} = \langle \Gamma, t, S\rangle$ *mit* $\Gamma = \{x_1 = t_1, \ldots, x_n = t_n\}$ *ist geschlossen, wenn*

$$(\bigcup_{i=1\ldots n} V_{free}(t_i) \cup V_{free}(t) \cup V_s) \subseteq (\mathrm{dom}(\Gamma) \cup \mathrm{dom}(S))$$

wobei V_S *der Menge aller Variablen auf dem Stack entspricht.*

Alle Regeln, gemäß denen die abstrakte Maschine $M^{\oplus}_{\Lambda_{let}}$ *eine Zustandsbeschreibung* $Z^{\oplus}_1$ *in eine Zustandsbeschreibung* $Z^{\oplus}_2$ *überführen kann, enthält Abbildung 6.4. Die Begriffe* Zustandsübergang, Startzustand *und* Endzustand *werden unter Angleichung des Heaps von der Maschine* $M_{\Lambda_{let}}$ *übernommen.*

$$\begin{array}{llcl}
\textit{(Lookup)} & \langle \Gamma\{x = t_x\},\ x,\ S\rangle & \rightarrow & \langle (\Gamma,\ t_x,\ \#x : S\rangle \\
\textit{(Update)} & \langle (\Gamma,\ r,\ \#x : S\rangle & \rightarrow & \langle (\Gamma\{x = r\},\ r,\ S\rangle \text{ wobei } (*) \\
\textit{(Unwind)} & \langle \Gamma,\ t\,x,\ S\rangle & \rightarrow & \langle \Gamma,\ t,\ x : S\rangle \\
\textit{(Share)} & \langle \Gamma,\ \lambda x.t,\ y : S\rangle & \rightarrow & \langle \Gamma,\ \texttt{let}\ x = y\ \texttt{in}\ t,\ S\rangle \\
\textit{(Let)} & \langle \Gamma,\ \texttt{let}\ x = t_x\ \texttt{in}\ s,\ S\rangle & \rightarrow & \langle \Gamma\{x = t_x\},\ s,\ S\rangle \\
\textit{(Left)} & \langle \Gamma,\ \texttt{choice}\ ,\ x : S\rangle & \rightarrow & \langle \Gamma,\ (\lambda y.\lambda z.y),\ x : S\rangle \\
\textit{(Right)} & \langle \Gamma,\ \texttt{choice}\ ,\ x : S\rangle & \rightarrow & \langle \Gamma,\ (\lambda y.\lambda z.z),\ x : S\rangle
\end{array}$$

(∗) r ist eine Abstraktion oder die Konstante `choice` . r' ist eine Kopie von r bei der alle Bindungen mittels frischer Variablen umbenannt wurden.

Abbildung 6.4: Die Überführungsregeln der abstrakten Maschine $M^{\oplus}_{\Lambda_{let}}$.

Im weiteren gilt die Vereinbarung, daß bei der Benutzung des Begriffs Zustandsbeschreibung stets davon ausgegangen wird, daß diese gültig ist. Es ist offensichtlich, daß durch keine der Überführungsregeln der Maschine $M^{\oplus}_{\Lambda_{let}}$ eine gültige in eine ungültig Zustandsbeschreibung überführt werden kann.

6.3.0.1 Gegenüberstellung der Maschinen $M_{\Lambda_{let}}$ und $M^{\oplus}_{\Lambda_{let}}$

Bei der Maschine $M^{\oplus}_{\Lambda_{let}}$ handelt es sich offensichtlich um eine simplifizierte Version der Maschine $M_{\Lambda_{let}}$. Die Struktur sowie die Verwaltung des Heaps sind vereinfacht worden. In diesem Zusammenhang haben sich auch die Eigenschaften der Update-Markierungen geändert. Sie kommen nun nicht mehr auf dem Heap vor und beinhalten dafür eine Zusatzinformation, nämlich den Namen der Variablen, unter dem die *Update*-Regel eine Bindung auf dem Heap einrichten soll.

6.3.0.2 Austauschbarkeit von $M_{\Lambda_{let}}$ und $M^{\oplus}_{\Lambda_{let}}$

Ziel dieses Abschnitts ist zu zeigen, daß das Terminierungsverhalten der beiden abstrakten Maschinen $M_{\Lambda_{let}}$ und $M^{\oplus}_{\Lambda_{let}}$ bezüglich aller Startzustände identisch ist. Der Begriff identisch ist dabei derart zu verstehen, daß egal in welchen Kontext $C^{\ominus}$ ein Ausdruck t eingesetzt wird, stets für die beiden auf Basis von $C^{\ominus}[t]$ bildbaren Startzustände dasselbe Terminierungsverhalten vorliegt.
Wir formulieren dazu einen Begriff der kontextuellen Äquivalenz für die Maschine $M^{\oplus}_{\Lambda_{let}}$.

Definition 6.3.2. *Sei $Z^{\oplus}$ eine beliebige geschlossene Zustandsbeschreibung.*

$Z^{\oplus}\Downarrow_{M^{\oplus}}$ $\Leftrightarrow_{Def}$ *es existiert ein Endzustand $Z^{\oplus}_{f}$, so daß $Z^{\oplus} \xrightarrow{*} Z^{\oplus}_{f}$* *(Konvergenz)*

$Z^{\oplus}\Uparrow_{M^{\oplus}}$ $\Leftrightarrow_{Def}$ *ausgehend von $Z^{\oplus}$ existiert eine nicht endlich beschränkte Folge einzelner Zustandsübergänge* *(Divergenz)*

Definition 6.3.3. *Seien s, t zwei $\Lambda^{\ominus}_{let}$-Ausdrücke. s und t sind* kontextuell äquivalent *bezüglich der Maschine $M^{\oplus}_{\Lambda_{let}}$, bezeichnet durch $s \sim_{c,M^{\oplus}} t$, gdw. für alle Kontexte $C^{\ominus}$ mit der Eigenschaft, daß $C^{\ominus}[s]$ und $C^{\ominus}[t]$ geschlossen ist, gilt*

$$\begin{array}{rlcl} & \langle \emptyset, C^{\ominus}[s], \varepsilon \rangle \Downarrow_{M^{\oplus}} & \Longleftrightarrow & \langle \emptyset, C^{\ominus}[t], \varepsilon \rangle \Downarrow_{M^{\oplus}} \\ \wedge & \langle \emptyset, C^{\ominus}[s], \varepsilon \rangle \Uparrow_{M^{\oplus}} & \Longleftrightarrow & \langle \emptyset, C^{\ominus}[t], \varepsilon \rangle \Uparrow_{M^{\oplus}} \end{array}$$

Definition 6.3.4. *Die Abbildung ϑ, die jeder Zustandsbeschreibung der abstrakten Maschine $M^{\oplus}_{\Lambda_{let}}$ eine Zustandsbeschreibung der Maschine $M_{\Lambda_{let}}$ zuordnet, wird wie folgt definiert:*

$$\vartheta(\langle (x_1 = t_1, \ldots, x_n = t_n),\ t,\ S \rangle) := \langle \vartheta_H \left((x_1 = t_1, \ldots, x_n = t_n) \right),\ t,\ S \rangle$$

wobei

$$\begin{array}{rcl} \vartheta_H\left(()\right) & := & \emptyset \\ \vartheta_H\left((x = t_x, \overrightarrow{y = t_y})\right) & := & \left\{ \begin{array}{ll} \vartheta_H\left((\overrightarrow{y = t_y})\right) & \text{falls } t_x \equiv \#x \\ \vartheta_H\left((\overrightarrow{y = t_y})\right)\{x = t_x\} & \text{sonst} \end{array} \right. \end{array}$$

Wir sind nun in der Lage die Verhaltensgleichheit beider Systeme zu zeigen.

Lemma 6.3.1. *Gegeben zwei Zustandbeschreibungen Z und $Z^{\oplus}$ mit $Z^{\oplus} = \vartheta(Z)$. Dann gilt für alle Überführungsregeln $a \in \{$LookUp, Update, Unwind, Share, Let, Left, Right$\}$*

- *Wenn $Z \xrightarrow{a} Z_1$, dann existiert eine Zustandsbeschreibung $Z^{\oplus}_1$ mit $Z^{\oplus} \xrightarrow{a} Z^{\oplus}_1$ und $Z^{\oplus}_1 = \vartheta(Z_1)$.*
- *Wenn $Z^{\oplus} \xrightarrow{a} Z^{\oplus}_1$, dann existiert eine Zustandsbeschreibung Z_1 mit $Z \xrightarrow{a} Z_1$ und $Z^{\oplus}_1 = \vartheta(Z_1)$.*

Beweis. Dies erbringt ein einfacher Abgleich zwischen den Überführungsregeln von $M_{\Lambda_{let}}$ und $M^{\oplus}_{\Lambda_{let}}$. Unabhängig von der Reihenfolge der Bindungen auf dem Heap bei der Zustandsbeschreibung Z kann auf die beiden Zustandsbeschreibungen Z und $Z^{\oplus}$ immer dieselbe Überführungsregel angewendet werden. Die nach Anwendung der Überführungsregeln erhaltenen Zustandsbeschreibungen Z_1 und $Z^{\oplus}_1$ können mittels der Abbildung ϑ stets wieder in einander überführt werden, da die Überführungsregeln von $M_{\Lambda_{let}}$ und $M^{\oplus}_{\Lambda_{let}}$ Effekt-äquivalent sind. Bei dem mehrdeutigen Sonderfall von *Left* und *Right* ist dabei jeweils auf beiden Seiten die gleiche Überführungsregel anzuwenden. □

Lemma 6.3.2. *Z und $Z^{\oplus}$ seien zwei geschlossene Zustandsbeschreibungen mit $Z^{\oplus} = \vartheta(Z)$. Dann*

$$Z{\Downarrow}_M \iff Z^{\oplus}{\Downarrow}_{M^{\oplus}} \quad \wedge \quad Z{\Uparrow}_M \iff Z^{\oplus}{\Uparrow}_{M^{\oplus}}$$

Beweis. Durch Induktion über alle der Länge nach aufsteigend sortierten Folgen von Zustandsübergängen, läßt sich unter Zuhilfenahme des obigen Lemmas leicht zeigen, daß aus der Existenz einer Folge von Zustandsübergangen der Länge n ausgehend von Z die Existenz einer Folge von Zustandsübergangen derselben Länge ausgehend von $Z^{\oplus}$ folgt und umgekehrt. Daraus folgt direkt die wechselseitige Übertragbarkeit eines Zustandsübergangs zu einem Endzustand sowie eines unendlichen Zustandsübergangs. □

Im Unterschied zu der Maschine $M_{\Lambda_{let}}$ können wir bei der Maschine $M^{\oplus}_{\Lambda_{let}}$ keine Abbildung von Zustandsbeschreibungen auf einzelne Ausdrücke definieren. Es existiert jedoch eine Subklasse aller Zustandsbeschreibungen von $M^{\oplus}_{\Lambda_{let}}$, bei der eine solche Abbildung möglich ist. Dies ist die Subklasse aller Startzustände.

Lemma 6.3.3. *Sei t ein beliebiger $\Lambda^{\ominus}_{let}$-Ausdruck. Für alle Kontexte $C^{\ominus}$ gilt:*

$$\begin{array}{rcl} & \langle\emptyset, C^{\ominus}[t], \varepsilon\rangle{\Downarrow}_{M^{\oplus}} \iff \langle(), C^{\ominus}[t], \varepsilon\rangle{\Downarrow}_M \\ \wedge & \langle\emptyset, C^{\ominus}[t], \varepsilon\rangle{\Uparrow}_{M^{\oplus}} \iff \langle(), C^{\ominus}[t], \varepsilon\rangle{\Uparrow}_M \end{array}$$

Beweis. Die Abbildung ϑ wird bei Beschränkung des Definitionsbereichs auf alle Startzustände bijektiv, wie leicht überprüft werden kann. Wesentlich ist dabei, daß bei der Maschine $M_{\Lambda_{let}}$ das Vorkommen von Update-Markierungen auf dem Heap das Vorkommen von Update-Markierungen auf dem Stack impliziert.
Die Bijektivität sichert eine eindeutige Zuordnung der Startzustände beider Maschinenmodelle durch die Abbildung ϑ und liefert in Verbindung mit Lemma 6.3.2 die Hypothese. □

Satz 6.3.1. *s, t seien zwei $\Lambda^{\ominus}_{let}$-Ausdrücke. Dann*

$$s \sim_{c,M} t \iff s \sim_{c,M^{\oplus}} t$$

Beweis. Beide Richtungen sind jeweils eine einfache Entwicklung für Konvergenz und Divergenz unter Zuhilfenahme von Lemma 6.3.3:

$\Rightarrow$

$$\langle\emptyset, C^{\ominus}[s], \varepsilon\rangle{\Downarrow}_{M^{\oplus}} \Leftrightarrow \langle(), C^{\ominus}[s], \varepsilon\rangle{\Downarrow}_M \Leftrightarrow \langle(), C^{\ominus}[t], \varepsilon\rangle{\Downarrow}_M \Leftrightarrow \langle\emptyset, C^{\ominus}[t], \varepsilon\rangle{\Downarrow}_{M^{\oplus}}$$
$$\langle\emptyset, C^{\ominus}[s], \varepsilon\rangle{\Uparrow}_{M^{\oplus}} \Leftrightarrow \langle(), C^{\ominus}[s], \varepsilon\rangle{\Uparrow}_M \Leftrightarrow \langle(), C^{\ominus}[t], \varepsilon\rangle{\Uparrow}_M \Leftrightarrow \langle\emptyset, C^{\ominus}[t], \varepsilon\rangle{\Uparrow}_{M^{\oplus}}$$

$\Leftarrow$

Erfolgt genauso wie oben, nur daß $s \sim_{c,M^{\oplus}} t$ als Voraussetzung benutzt wird. □

Alle Ergebnisse dieses Abschnitts können nun als Erweiterung von Korollar 6.2.5 in einem einzelnen Korollar zusammengefaßt werden.

Korollar 6.3.1. *s, t seien zwei $\Lambda_{let}^{\ominus}$-Ausdrücke. Dann*

$$s \sim_c t \iff \tau(s) \sim_{c,\ominus} \tau(t) \iff \tau(s) \sim_{c,M} \tau(t) \iff \tau(s) \sim_{c,M^{\oplus}} \tau(t)$$

Damit ist gezeigt, daß sowohl die Maschine $M_{\Lambda_{let}}$ wie auch ihre vereinfachte Variante $M_{\Lambda_{let}}^{\oplus}$ geeignet sind, die no-Reduktion des Λ_{let}-Kalkül nachzuahmen. Das Terminierungsverhalten eines gegeben Ausdruck ist in allen Kontexten bei den beiden Maschinenmodellen sowie der no-Reduktion des Λ_{let}-Kalkül immer identisch.

6.4 Abschließende Bemerkung

Die zuletzt vorgestellten abstrakten Maschinen $M_{\Lambda_{let}}$ und $M_{\Lambda_{let}}^{\oplus}$ besaßen alleinig den Zweck, ein maschinelles Modell vorzustellen, mit dessen Hilfe es möglich ist, die no-Reduktion des Λ_{let}-Kalküls nachzuahmen. Für eine Anwendung in der Praxis sind beide Modelle jedoch ungeeignet, da beispielsweise keine rekursiven Strukturen auf dem Heap möglich sind und Konstruktoren sowie ein case-Konstrukt fehlen. Ferner sind die Überführungsregeln partiell ineffizient. Ein Beispiel dafür zeigt die Regel *Share*, die durch eine Regel *Subst* ersetzt werden könnte, bei der eine direkte Ersetzung der abstrahierten Variable vorgenommen wird, anstatt einen Let-Ausdruck zu erzeugen. Auf eine Vorstellung von Optimierungs- und Erweiterungsmöglichkeiten hin zu einem praktisch anwendbaren Modell wird in der vorliegenden Arbeit verzichtet, da die Komplexität dieser Aufgabenstellung nur eine oberflächliche Betrachtung zuließe. Einen Einblick in die Tiefe der Problematik gibt die Vielzahl der bestehenden Maschinenmodelle (G-machine [Aug84] sowie [Joh84], STG-machine [PJ92], ABC-machine [Koo90], TIM [FW87]), die zur Evaluation verzögernd auswertender funktionaler Programme entwickelt wurden.

Kapitel 7

Maschinelle Unterstützung der Beweisführung

Die Gabel- und Vertauschungsdiagramme der in den letzten Kapiteln betrachteten Reduktionen verfügen zum Teil über eine erhebliche Komplexität, als extremes Beispiel sei hier auf die *ucp*-Reduktion verwiesen. Ein kritischer Leser wird daher die berechtigte Frage stellen, welche Methodik gewählt wurde, die Diagramme zu entwickeln bzw. ob, und dann wie, diese überprüft wurden. Vielen Autoren, die sich in ihrer Arbeit mit Reduktionskalkülen oder Termersetzungssystemen auseinandergesetzt haben, dürfte es eine bekannte Erfahrung sein, daß bei der Entwicklung von Diagrammen leicht ein Fall übersehen wird, der potentiell den ganzen Beweis sprengt. Dies gilt insbesondere dann, wenn die Diagrammbetrachtungen ausschließlich auf dem Papier geschehen. Es wird im allgemeinen als peinlich empfunden, wenn ein Autor von einem genauen Leser auf solch eine Unachtsamkeit aufmerksam gemacht wird. Um eine gewisse Sicherheit zu erhalten, wurde zur Überprüfung der Diagramme bei der vorliegenden Arbeit maschinelle Unterstützung in Form eines funktionalen Programms zur Hilfe gezogen. In diesem Kapitel soll kurz vorgestellt werden, wie diese maschinelle Unterstützung aussah und welchen Wert sie hatte.

7.1 Grundzüge des maschinellen Werkzeugs

Bevor die Struktur, Arbeitsweise des entwickelten Programms vorgestellt wird, soll zunächst kurz beschrieben werden, welche Methodik für dessen Arbeitsweise angedacht wurde. Die Zielsetzung einen vollständigen automatischen Beweiser zu entwickeln, der für einen gegebenen Satz von Gabel- bzw. Vertauschungsdiagrammen entscheidet, ob diese vollständig sind, d.h. jede Situation bei einer betrachteten Reduktion abdecken, wurde als zu hochgesteckt angesehen und verworfen. Die Gründe dafür waren folgende:

- Eine intuitive Vermutung war, daß die Menge möglicher Diagramme bei einigen Reduktionen nicht endlich ist, wenngleich eine endliche Beschreibung existiert. Die Vermutung beruhte auf Versuchen mittels einer variierten Knuth-Bendix-Methodik die Vollständigkeit der Diagramme auf dem Papier zu prüfen, was jedoch nicht gelang, da die Betrachtungen zu immer neuen und komplexeren Fällen führten, ohne daß ein Terminierungskriterium gefunden werden konnte. Eine Methodik, die dieser besonderen Situation gerecht geworden wäre, konnte in der Literatur nicht gefunden werden.
- Die Differenzierung zwischen no-Redizes und nicht no-Redizes war eine zusätzliche Schwierigkeit, die durch einen automatischen Beweiser zu leisten gewesen wäre.
- Der zeitliche Rahmen war sehr eng, da das Werkzeug bereits bei der Veröffentlichung [KSS98] eingesetzt werden sollte.

Dies führte dazu, auf eine Methodik zurückzugreifen, deren Anwendung zwar keine vollständige Sicherheit gibt, jedoch ein hohes Maß an Gewißheit. Die angedachte Methodik besaß folgendes Aussehen:

> Wir geben eine zu betrachtende Reduktionsart r, z.B. $(i, llet)$, vor. Für eine möglichst große Menge von Tupel (t, s), bei denen stets der Ausdruck s durch r-Reduktion aus t erhalten wurde, testen wir, ob ein gegebener Satz von Vertauschungs- bzw. Gabeldiagrammen ein passendes Diagramm enthält. Damit können wir testen, ob ein gegebener Satz von Vertauschungs- bzw. Gabeldiagrammen bezogen auf eine große Menge von r-Reduktionen vollständig ist.

Somit war das angedachte Programm ein maschinelles Überprüfungswerkzeug jedoch kein automatischer Beweiser. Die Arbeitsweise des maschinellen Überprüfungswerkzeugs soll nun vorgestellt werden.

7.2 Arbeitsweise des maschinellen Überprüfungswerkzeugs

In einem früheren Kapitel wurde für Vertauschungs- und Gabeldiagramme gezeigt, daß sie sehr verwandt, in den meisten Fällen sogar wechselseitig überführbar sind. Die Verbindung wurde dort mittels des Reduktionsdiagramms aus Abbildung 7.1 hergestellt, das sowohl als Gabel- wie auch als Vertauschungsdiagramm interpretiert werden kann. Das Diagramm aus Abbildung 7.1 ist im Kontext der

$$\begin{tikzcd}
t \arrow[r, "r"] \arrow[d, dotted] \arrow[dr, "r_L"'] & s \arrow[d, "r_R"] \\
\circ \arrow[r, dotted] & s'
\end{tikzcd}$$

wobei r_L wie r_R stets in eine Form $\xrightarrow{no,*} \circ \xrightarrow{r'}$ zerlegt werden können, bei der die Reduktion r' keine no-Reduktionen mehr beinhaltet.

Abbildung 7.1:

weiteren Ausführungen grundlegend, da die benutzte Methodik an diesem Diagramm gedanklich verankert werden kann.
Im Zentrum steht die Reduktion $t \xrightarrow{r} s$, egal ob eine Interpretation als Gabel- wie Vertauschungsdiagramm vorliegt; einmal bildet diese den Zweig einer Gabel, im anderen Fall wird sie über eine no-Reduktion (in diesem Fall r_R) hinwegbewegt. Wir betrachten diese Reduktion im weiteren als gegeben und assoziieren sie mit einem Tupel (t, s). Zwei weitere Beobachtungen sind wichtig:

- Die linke Seite wird bei obigem Diagramm mit einer einzelnen r_L-Reduktion assoziiert, so daß sich insgesamt eine 'Dreieckssicht' ergibt.
- Bei den Reduktionen r_L und r_R erfolgt keine Differenzierung zwischen gegebenen und existenten Reduktionen. Das Fortlassen dieser Differenzierung wird durch die Eindeutigkeit des no-Redex in Kombination mit der stets gegebenen Zerlegbarkeit der Reduktionen r_L und r_R ermöglicht. Abgesehen vom Fall einer (no, nd)-Reduktion ist eine no-Reduktion für einen Ausdruck t immer eindeutig gegeben sowie eindeutig existent, was eine solche Differenzierung überflüssig macht. Da (no, nd)-Reduktionen im Kontext der Gabel- und Vertauschungsdiagramme immer leicht auf dem Papier überprüfbar waren, wurde ihr maschinelle Untersuchung zunächst hinten angestellt und damit die vereinfachte Sichtweise ermöglicht.

O.b.d.A. gehen wir nun davon aus, daß wir für die r-Reduktion(sregel) einen Satz von Gabeldiagrammen entwickelt haben; die nun vorgestellte Methodik kann gleichermaßen für Vertauschungsdiagramme angewendet werden. Alle Gabeldiagramme aus dem gegebenen Satz transformieren wir nun derart, daß wir zu jedem Gabeldiagramm ein Tupel (r_L, r_R) erhalten. Dies soll an einem kleinen Beispiel demonstriert werden:

Beispiel 7.2.1. Die gegebene r-Reduktion sei vom Typ $(i, lbeta)$. Eines der Gabeldiagramme der $(i, lbeta)$-Reduktion ist:

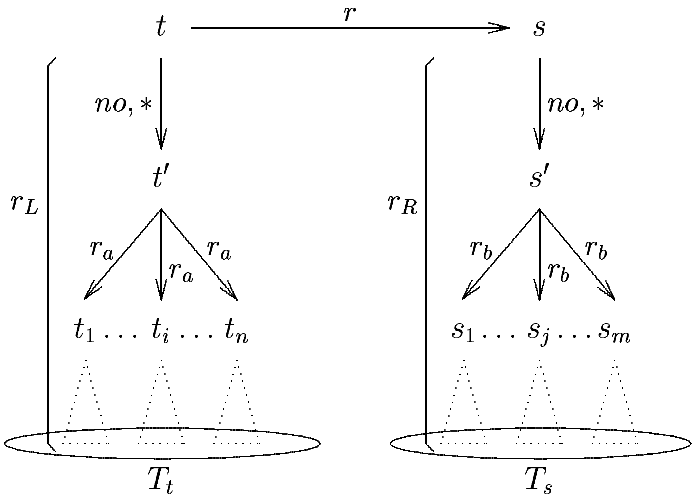


Abbildung 7.2:

$$\xleftarrow{no,cp} \circ \xrightarrow{i,lbeta} \rightsquigarrow \xrightarrow{i,lbeta} \circ \xrightarrow{i,lbeta} \circ \xleftarrow{no,cp}$$

Das zu diesem Gabeldiagramm gehörige (r_L, r_R)-Tupel würde somit lautet:

$$(\xrightarrow{no,cp}\xrightarrow{i,lbeta}\xrightarrow{i,lbeta}, \xrightarrow{no,cp})$$

Wir werden nun sehen, wie wir maschinell prüfen können, ob ein (r_L, r_R)-Tupel zu der gegebenen Reduktion $t \xrightarrow{r} s$ paßt. Dazu betrachten wir Abbildung 7.2, die den Überprüfungsprozeß grafisch wiedergibt. Von beiden Ausdrücken s und t aus bilden wir durch maschinelles Reduzieren zwei Mengen T_s und T_t auf folgende Weise:
Wir arbeiten zunächst maschinell den no-Teil bis t' bzw. s' ab, während dieser Phase erhalten wir immer genau einen Nachfahren, da die no-Reduktion wegen des Ausschlusses von nd-Reduktionen immer eindeutig bleibt. Ab t' bzw. s' treten zwei Veränderungen ein:

- Keine weitere Reduktion wird eine no-Reduktion sein.
- Alle Reduktionen außerhalb der no-Reduktionen sind per Voraussetzung existenzquantifiziert. Dies hat zur Folge, daß wir ab nun in jedem Schritt für einen Ausdruck alle Redizes eines bestimmten Typs wie beispielsweise $(i, lbeta)$ maschinell bestimmen, reduzieren und weiterverfolgen müssen. Deshalb entsteht ab t' bzw. s' eine baumartige Struktur.

Wesentlich ist, daß wir r_L bzw. r_R hier nicht als spezifische Reduktionen ansehen, sondern als Folge von Typspezifikationen aufeinanderfolgender Einzelreduktionen, somit als eine schematische Beschreibung der gesuchten zwei Reduktionen

r_L und r_R. Solche Typspezifikationen lauten beispielsweise (no, cp), $(i, lbeta)$ oder $ldel$. Die Pfade der entstehenden Bäume entsprechen der Menge aller Reduktionen ausgehend von t bzw. s, die schematisch r_L bzw. r_R gehorchen.

Ein (r_L, r_R)-Tupel paßt nun zu der Reduktion $t \xrightarrow{r} s$, wenn abstrahiert von Umbenennungen gebundener Variablen in T_s und T_t derselbe Ausdruck vorkommt. Dies bedeutet, daß wir die gesuchten Teile der beiden Reduktionen r_L und r_R des Reduktionsdiagramms aus Abbildung 7.1 gefunden haben. Wie bereits aufgeführt, sind dabei alle Nebenbedingungen bezüglich gesuchter und gegebener Teile eines Gabeldiagramms erfüllt.

Auf diese Art können für eine gegebene Reduktion $t \xrightarrow{r} s$ alle Gabeldiagramme, die für eine r-Reduktion entwickelt wurden, überprüft werden. Mindestens ein Diagramm muß dabei passen, ansonsten deckt die Menge der entwickelten Diagramme diesen Fall nicht ab, und ist nicht vollständig.

Damit ist der Kern des Überprüfungsprozesses beschrieben. Bleibt zu zeigen, wie die gegebenen Reduktionen $t \xrightarrow{r} s$ erzeugt werden. Dies geschieht nach einem sehr einfachen Schema:

> Wir geben einen Ausdruck t vor und ermitteln alle Redizes vom Typ r, die in t vorkommen. Jeden Redex betrachten wir als eigenen Fall und erzeugen für diesen ein Tupel (t, s), wobei s durch Reduktion des jeweiligen Redex in t erhalten wird. Eine Menge von Ausdrücken, für die wir so eine Menge von (t, s)-Tupeln bestimmen, können wir uns mittels eines Generators erzeugen.

Wenden wir den obigen Überprüfungsprozeß auf eine 'große Menge' generierter Ausdrücke an, so betrachten wir also noch um so mehr (t, s)-Tupel, d.h. Reduktionen des untersuchten Typs r.

7.2.1 Hinzunahme von (no, nd)-Reduktionen

Wie bereits erwähnt, wurde die Betrachtung von nd-Reduktionen zunächst hinten angestellt, d.h. es wurden keine Ausdrücke erzeugt, die die Konstante `choice` enthalten. Im Zusammenhang mit der Untersuchung der $detpar$-Reduktion wurde jedoch eine Hinzunahme von `choice` bei den generierten Ausdrücken erforderlich, da der Nichtdeterminismus bei dieser Reduktion von elementarer Bedeutung ist. Ein Problem im Kontext der gewählten Methodik entstand dadurch, daß die gegebene no-Reduktion obgleich eines eindeutigen no-Redex im Falle eines (no, nd)-Redex nicht eindeutig ist. Zur Lösung dieser Problematik wurde auf einen 'Trick' zurückgegriffen, der im Kontext der $detpar$-Reduktion zulässig war und den Aufwand an zusätzlicher Programmierung minimal hielt:

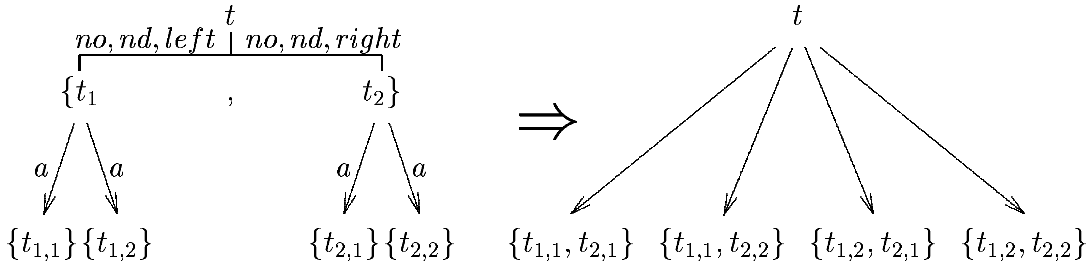


Abbildung 7.3:

Die Sichtweise bei no-Reduktionen wurde auf Mengen von Ausdrücken ausgedehnt, d.h. ähnlich wie bei der Mengenkonfluenz wurde ein (no, nd)-Redex maschinell zu der Menge der beiden Ausdrücke reduziert, die durch die $(no, nd, left)$- und $(no, nd, right)$-Reduktionen erhalten worden wären. Die Baumsichtweise aus Abbildung 7.2 wird dadurch erweitert. Mit den Ausdrücken t und s werden in der Folge zwei Mengen von Ausdrucksmengen statt zwei Mengen einzelner Ausdrücke assoziiert. Wie diese Mengen von Ausdrucksmengen entwickelt werden soll an einem Beispiel verdeutlicht werden:

Beispiel 7.2.2. t sei ein Ausdruck mit einem (no, nd)-Redex, dessen Reduzierung zu zwei Ausdrücken t_1 und t_2 führe. t_1 und t_2 wiederum seien zwei Ausdrücke, die jeweils zwei Redizes eines Typs a besitzen, die keine no-Redizes und nicht vom Typ nd seien und deren Reduzierung jeweils zu $t_{1,1}$ und $t_{1,2}$ sowie $t_{2,1}$ und $t_{2,2}$ führe. Abbildung 7.3 zeigt grafisch das Vorgehen in diesem Fall. Wir bilden alle 4 möglichen Kombinationen, die sich aus den 2 Paaren von Mengen einzelner Ausdrücke bilden lassen. Diese so gebildeten 4 Mengen beschreiben alle Ausdruckskombinationen, die von t aus mittels einer (no, nd)-Reduktion gefolgt von einer a-Reduktion erreichbar sind.
Kommen somit bei der maschinellen Reduktion keine nd-Reduktionen vor, so entsteht bei dieser Methodik derselbe Baum wie bei der zuvor vorgestellten einfacheren Methodik, mit dem einzigen Unterschied, daß die Blätter aus Mengen einzelner Ausdrücke statt einzelner Ausdrücke bestehen.

Statt zwei gleicher einzelner Ausdrücke in den Mengen T_t und T_s wird nun die Existenz zweier gleicher Mengen von Ausdrücken als Kriterium dafür formuliert, ob ein Diagramm paßt oder nicht.
Es sei an dieser Stelle ausdrücklich darauf hingewiesen, daß die oben vorgestellte Technik nicht im Kontext aller Reduktionen zulässig ist, so wäre z.B. die Kombination von (i, nd)- und (no, nd)-Reduktionen wegen der Abschnitt 2.4.2 aufgeführten Probleme unzulässig.

Bevor auf den Wert des maschinellen Überprüfungswerkzeugs eingegangen wird, soll kurz noch die Implementation skizziert werden.

Modul	*Funktion des Moduls*
Basis	Basisfunktionen vergleichbar dem Prelude
Expression	Definition von algebraischen Datenstrukturen zur Repräsentation von Ausdrücken und Redizes, Basisfunktionen zum Umgang mit Ausdrücken wie Bestimmung der freien Variablen, Generator zur Erzeugung von Mengen von Ausdrücken
Reduce	Funktionen zum Markieren und Reduzieren von Redizes
Rules	Gabel- bzw. Vertauschungsdiagramme repräsentiert als algebraischen Datenstruktur
Analyser	Funktionen zum Prüfen der Vollständigkeit der in 'Rules' kodierten Gabel- bzw. Vertauschungsdiagramme auf endlichen Ausdrucksmengen
Parser	Parser zum Einparsen von Ausdrücken
Main	Formulierung einer spezifischen Vollständigkeitsprüfung

Tabelle 7.1:

7.2.2 Struktur der Implementation

Zur Implementation des maschinellen Überprüfungswerkzeugs wurde die verzögernd auswertende funktionale Programmiersprache Haskell ([PHA$^+$97]) benutzt. Wie für größere Programme üblich, erfolgte eine Zerlegung in Module. Alle Module zusammen mit einer Beschreibung ihrer Funktion zeigt Tabelle 7.1. Aus Zeitgründen wurde auf die Realisation einer IO-Schnittstelle verzichtet. Ein gewünschter Vollständigkeitstest ist im Modul 'Main' als Funktion 'main' zu spezifizieren. Um den Überprüfungsprozeß nicht nur auf generierte Ausdrücke, sondern auch auf ganz spezifische einzelne Ausdrücke anwenden zu können wurde zusätzlich ein Parser bereitgestellt.

7.3 Wert des maschinellen Überprüfungswerkzeugs

Für die Arbeit mit dem maschinellen Überprüfungswerkzeug stand eine übliche Workstation mit 256 MB Hauptspeicher zur Verfügung. Da verzögernd auswertende funktionale Programme im allgemeinen mit wachsender Rechenzeit immer mehr Heapspeicher verlangen, setzte der verfügbare Hauptspeicher Grenzen, was die Anzahl überprüfbarer Ausdrücke betraf. Eine Menge von 2.84 Mio. generierten Ausdrücken wurde bei allen Reduktionen der vorliegenden Arbeit erfolgreich überprüft. Die Zahl von 2.84 Mio. entspricht ca. der Zahl aller möglichen geschlossenen Λ_{let}-Ausdrücke, die sich aus 3 Abstraktionen, 3 Let-Ausdrücken sowie 2 Applikationen zusammensetzen. Das Wort 'ca.' ist darin begründet, daß der Generator bei der Generierung bestimmte Ausdrucksformen unterdrückt, wie beispielsweise Subausdrücke mit mehr als zwei `let` unterhalb einer Abstraktion.

Schon nach kurzer Zeit erwies sich die Entwicklung des maschinellen Überprüfungswerkzeugs als sehr wertvoll. Es zeigte sich, daß bei der Entwicklung der Vertauschungsdiagramme der *llet*-Reduktion auf dem Papier die beiden Diagramme *iv)* und *v)* aus Lemma 2.4.7 übersehen wurden. Tatsächlich tauchten diese Sonderfälle erst bei verhältnismäßig großen Ausdrücken auf, die üblicherweise außerhalb einer Papierbetrachtung liegen. Einer der kleinsten Ausdrücke, bei dem das Vertauschungsdiagramm *iv)* für $\omega = 1$ vorkommt lautet:

$$\texttt{let } a = \underline{(\texttt{let } b = \texttt{let } c = \lambda d.d \texttt{ in } c \texttt{ in } b)}(\lambda e.e) \texttt{ in } a$$

Dabei ist der unterstrichene *llet*-Redex die gegebene zu vertauschende interne Reduktion. Besonders wertvoll war, daß das maschinelle Überprüfungswerkzeug auch noch auf einen größeren Bruder des obigen Ausdrucks für $\omega = 2$ aufmerksam machte:

$$\texttt{let } a = (\underline{(\texttt{let } b = \texttt{let } c = \lambda d.d \texttt{ in } c \texttt{ in } b)}(\lambda e.e))(\lambda f.f) \texttt{ in } a$$

Eine Analyse der Situation bei den beiden obigen Ausdrücke zeigte dann, daß das Vertauschungsdiagramm *iv)* für beliebige ω vorkommen konnte. Im Rahmen dieser Analyse wurde auch die Existenz und Struktur der W-Kontexte erkannt, die in vielen Beweisen eine Schlüsselstellung einnehmen.

7.3.1 Zusätzliche maschinelle Analysen

Der Rahmen von Funktionen, die bei der Implementation des zuletzt beschriebenen maschinellen Überprüfungswerkzeugs realisiert wurden, ließ auch andere interessante Analysen zu, die nun kurz beschrieben werden sollen.

Der in dieser Arbeit vorgestellte Λ_{let}-Kalkül ist bei Ausblendung des Nichtdeterminismus sehr verwandt mit dem in [AFM$^+$95] vorgestellten λ_{let}-Kalkül. Ein interessanter Unterschied zwischen beiden Kalkülen betrifft jedoch die Definition der no-Reduktion, die in [AFM$^+$95] über Evaluationskontexte und “Standard Reduction rules” geschieht. Dies hat zur Folge, daß die no-Reduktion des λ_{let}-Kalküls leicht strikter als die des Λ_{let}-Kalküls ist. Der geschlossene Ausdruck

$$(\texttt{let } x = (\texttt{let } y = \lambda a.a \texttt{ in } y) \texttt{ in } x)(\lambda b.b)$$

verdeutlicht den Unterschied. Die no-Reduktion des Λ_{let}-Kalkül wählt den äußeren *lapp*-Redex als no-Redex, bei [AFM$^+$95] hingegen wird der innere *cp*-Redex als no-Redex gewählt. Eine Fragestellung, die sich nun ergab, war, welchen Einfluß auf die Länge der no-Reduktion die unterschiedliche Wahl des no-Redex hat. Der obige Ausdruck gibt bereits ein Beispiel, daß die Wahl des no-Redex gemäß [AFM$^+$95] zur Verlängerung der no-Reduktion führen kann. Dies zeigt die Gegenüberstellung 7.3.1.

no-Reduktion gemäß [AFM$^+$95]

$$\begin{aligned}
& (\texttt{let}\ x = (\underline{\texttt{let}\ y = \lambda a.a\ \texttt{in}\ y})\ \texttt{in}\ x)(\lambda b.b) \\
\xrightarrow{no,cp}\ & (\underline{\texttt{let}\ x = (\texttt{let}\ y = \lambda a.a\ \texttt{in}\ \lambda a.a)\ \texttt{in}\ x})(\lambda b.b) \\
\xrightarrow{no,llet}\ & (\texttt{let}\ y = \lambda a.a\ \texttt{in}\ \underline{\texttt{let}\ x = \lambda a.a\ \texttt{in}\ x)}(\lambda b.b) \\
\xrightarrow{no,cp}\ & \underline{(\texttt{let}\ y = \lambda a.a\ \texttt{in}\ \texttt{let}\ x = \lambda a.a\ \texttt{in}\ \lambda a.a)(\lambda b.b)} \\
\xrightarrow{no,lapp}\ & \texttt{let}\ y = \lambda a.a\ \texttt{in}\ \underline{(\texttt{let}\ x = \lambda a.a\ \texttt{in}\ (\lambda a.a)(\lambda b.b))} \\
\xrightarrow{no,lapp}\ & \texttt{let}\ y = \lambda a.a\ \texttt{in}\ \texttt{let}\ x = \lambda a.a\ \texttt{in}\ \underline{((\lambda a.a)(\lambda b.b))} \\
\xrightarrow{no,lbeta}\ & \texttt{let}\ y = \lambda a.a\ \texttt{in}\ \texttt{let}\ x = \lambda a.a\ \texttt{in}\ \underline{(\texttt{let}\ a = \lambda b.b\ \texttt{in}\ a})\\
\xrightarrow{no,cp}\ & \text{WHNF}
\end{aligned}$$

no-Reduktion beim Λ_{let}-Kalkül

$$\begin{aligned}
& \underline{(\texttt{let}\ x = (\texttt{let}\ y = \lambda a.a\ \texttt{in}\ y)\ \texttt{in}\ x)(\lambda b.b)} \\
\xrightarrow{no,lapp}\ & \underline{\texttt{let}\ x = (\texttt{let}\ y = \lambda a.a\ \texttt{in}\ y)\ \texttt{in}\ (x(\lambda b.b))} \\
\xrightarrow{no,llet}\ & \texttt{let}\ y = \underline{\lambda a.a}\ \texttt{in}\ \texttt{let}\ x = \underline{y}\ \texttt{in}\ (x(\lambda b.b)) \\
\xrightarrow{no,cp}\ & \texttt{let}\ y = \lambda a.a\ \texttt{in}\ \texttt{let}\ x = \underline{\lambda a.a}\ \texttt{in}\ (\underline{x}(\lambda b.b)) \\
\xrightarrow{no,cp}\ & \texttt{let}\ y = \lambda a.a\ \texttt{in}\ \texttt{let}\ x = \lambda a.a\ \texttt{in}\ \underline{((\lambda a.a)(\lambda b.b))} \\
\xrightarrow{no,lbeta}\ & \texttt{let}\ y = \lambda a.a\ \texttt{in}\ \texttt{let}\ x = \lambda a.a\ \texttt{in}\ \underline{(\texttt{let}\ a = \lambda b.b\ \texttt{in}\ a})\\
\xrightarrow{no,cp}\ & \text{WHNF}
\end{aligned}$$

Gegenüberstellung 7.3.1

Indem zusätzlich die Wahl des no-Redex gemäß [AFM$^+$95] implementiert wurde, konnten beide no-Reduktionen für eine große Menge generierter Ausdrücke gegenübergestellt werden. Dabei zeigten sich folgende Sachverhalte:

- Entstand ein Unterschied in der Länge der no-Reduktion so immer zuungunsten von [AFM$^+$95]. Es konnte kein Ausdruck gefunden werden, bei dem die Wahl des no-Redex gemäß [AFM$^+$95] zu einer kürzeren no-Reduktion führte.
- Die Unterschiede konnten beachtliches Ausmaß annehmen. Ein Beispiel zeigt der Ausdruck

  $$(\texttt{let}\ a = (\texttt{let}\ b = (\texttt{let}\ c = (\texttt{let}\ d = ((\lambda e.(\lambda f.f))(\lambda g.g))\ \texttt{in}\ d)\ \texttt{in}\ c)\ \texttt{in}\ b)\ \texttt{in}\ a)$$

  bei dem ein Verhältnis von 9 zu 15 no-Reduktionen vorliegt. Jedoch betraf die Differenz interessanterweise niemals die Zahl der $(no, lbeta)$- und (no, cp)- Reduktionen sondern ausschließlich die Zahl der $(no, llet)$- und $(no, lapp)$- Reduktionen. Diese Beobachtung ist im Hinblick auf eine Bewertung des aufgedeckten Sachverhalts von Bedeutung. $(no, llet)$- und $(no, lapp)$-Reduktionen werden deutlich schwächer als $(no, lbeta)$- und (no, cp)-Reduktionen bewertet, da sie in keinem direkten Zusammenhang mit dem Instanziierungsprozeß stehen, der beiden Kalkülen gedanklich zugrundeliegt.

Eine weiterer Zusatz, der durch Ergänzung des bestehenden Satzes von Funktionen der Implementation leicht realisierbar war, war eine maschinellen Unterstützung der Suche von Gabel- und Vertauschungsdiagrammen. Dies hatte den Hintergrund, daß die Entwicklung der Diagramme für die aufgedeckten Fälle auf dem Papier zum Teil mühselig und zeitaufwendig war. Die Verfahrensweise war dabei verhältnismäßig einfach:

Eine Generatorfunktion erzeugt eine Menge von Diagrammen und testet für jedes so erzeugt Diagramm, ob es paßt. Vorteilhaft war dabei die Eigenschaft verzögert auswertender funktionaler Programme auch bei der Verwendung unendlicher Strukturen terminieren zu können, da die zu erzeugende Diagrammmenge unendlich sein konnte.

7.4 Abschließende Bemerkung

Das entwickelte Überprüfungswerkzeug war im Kontext der vorliegenden Arbeit sinnvoll und hilfreich. Jedoch ist der aus Vereinfachungsgründen erfolgte Verzicht auf eine Trennung zwischen gegebenen und gesuchten Reduktionen nachträglich als Designfehler zu bewerten. Eine solche Trennung hätte beispielsweise, eine 'saubere' und vollständige Betrachtung der nd-Reduktionen ermöglicht.

Eine bleibende Problemstellung, auf die abschließend noch einmal aufmerksam gemacht werden soll, ist die Entwicklung einer Methodik, die in der Lage ist, maschinell die Vollständigkeit eines Satzes von Vertauschungs- bzw. Gabeldiagrammen zu beweisen.

Kapitel 8

Zusammenfassung und Ausblick

In dieser Arbeit wurde ein nichtdeterministisch erweiterter call-by-need Lambda-Kalkül vorgestellt, den wir als Λ_{let}-Kalkül bezeichnet haben. Zur Spezifikation des Nichtdeterminismus verfügt der Λ_{let}-Kalkül über eine `choice` -Konstante, deren Semantik erratic Nichtdeterminismus ist. Es wurde gezeigt, daß der Λ_{let}-Kalkül eine spezielle Form der Konfluenz erfüllt, die wir als Mengenkonfluenz bezeichnet haben. Wir haben für den Λ_{let}-Kalkül auf algorithmischer Basis eine no-Reduktion sowie die Begriffe WHNF und VWHNF definiert. Es wurde gezeigt, daß bestimmte Typen von no-Redizes mit bestimmten Kontextformen assoziiert werden können. Auf Basis der no-Reduktion wurde eine kontextuelle Präordnung $\leq_c$ definiert und darauf basierend wiederum der Begriff der kontextuellen Äquivalenz gebildet. Passend zur kontextuellen Äquivalenz wurde ein Kontext-Lemma bewiesen, dessen Zweck in der Beweisunterstützung lag. Von einer Vielzahl von Reduktionen wurde gezeigt, daß sie die kontextuelle Äquivalenz bewahren. Auf dieser Basis ist die Zulässigkeit von einfachem Lambda-Lifting beim Λ_{let}-Kalkül gezeigt worden. Der Begriff des deterministischen Subausdrucks wurde definiert und bewiesen, daß deterministische Subausdrücke sich bei Erhaltung der kontextuellen Äquivalenz stets kopieren lassen. Auf dieser Basis wurde die Zulässigkeit des Liftens deterministischer Subausdrücke gezeigt sowie die Austauschbarkeit der no-Reduktion des Λ_{let}-Kalküls mit dem call-by-name Evaluator des klassischen Lambda-Kalküls gezeigt. Eine rein operational beschränkte Betrachtung von Rekursion beim Λ_{let}-Kalkül wurde durchgeführt. Die Existenz eines Fixpunktkombinators Y_T wurde gezeigt und seine Eigenschaften untersucht. Eine zum Λ_{let}-Kalkül passende abstrakte Machine wurde vorgestellt und eine optimierte Form entwickelt. Es wurde ein maschinelles Werkzeug vorgestellt, dessen Zweck in der Beweisunterstützung lag.

Der Λ_{let}-Kalkül verfügt weder über ein rekursives `let` noch über Konstruktoren oder ein `case`-Konstrukt. Die vorliegende Arbeit enthält keine Überlegungen, wie der Λ_{let}-Kalkül um solche Elemente erweitert werden könnte. Solche Ergänzungen sind jedoch die Voraussetzung, um den Λ_{let}-Kalkül als die in der Einführung er-

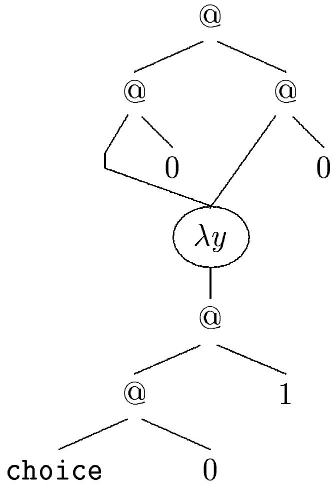


Abbildung 8.1: Reduktionsgraph G

wähnte mathematisch formale Grundlage einer funktionalen Programmiersprache benutzen zu können.

Die Modellierung von Terminal-I/O mittels erratic Nichtdeterminismus wurde in dieser Arbeit nicht weiter untersucht, sondern nur als Motivierung für die Einführung von erratic Nichtdeterminismus beim Λ_{let}-Kalkül aufgeführt. Es scheint, als ob die hier vorgestellte Form der `choice`-Regel nur geeignet ist, die Eingabeseite zu modellieren. Eine offene Fragestellung würde damit die Modellierung der Ausgabeseite bilden. Die Fragestellung der I/O-Modellierung wird weiter erschwert, wenn komplexere I/O-Formen wie Fenster-basiertes I/O modelliert werden sollen, da hier zusätzlich das Problem der Nebenläufigkeit zu beachten ist. Die Abbildung von I/O auf Kalkülebene kann als eigener Problemkreis angesehen werden, der eine intensive Betrachtung verdient.

Die vorliegende Arbeit beschränkt sich auf rein operationale Betrachtungen und steht damit in der Tradition der Dissertationen von Lassen [Las98] und Moran [Mor98]. Die Entwicklung einer denotationalen Semantik für den Λ_{let}-Kalkül wird in dieser Arbeit nicht betrachtet. Die Kenntnis einer denotationalen Semantik wäre jedoch wünschenswert, um über ein mathematisches Modell zum Λ_{let}-Kalkül zu verfügen.

Zwischen verzögernd auswertenden funktionalen Programmiersprachen und Reduktion auf Graphen wird im allgemeinen ein sehr enger Bezug gesehen. In dieser Arbeit wird die Beziehung zwischen dem Λ_{let}-Kalkül und Graphenreduktion nicht untersucht.
Ähnlich dem in der Einführung beschriebenen Unterschied zwischen call-by-name und call-by-need Auswertung, der bei der Anwesenheit von erratic Nichtdeterminismus entsteht, ist ein Unterschied zwischen dem Λ_{let}-Kalkül und Graphenreduktion zu beobachten, der auch durch die Anwesenheit von erratic Nichtdetermi-

nismus hervorgerufen wird. Anhand des folgenden Beispiels soll der Unterschied verdeutlicht werden:
Die beiden Ausdrücke

$$\begin{aligned} t_1 \equiv\ & \texttt{let}\ z = (\texttt{let}\ x = \texttt{choice}\ 0\ 1\ \texttt{in}\ \lambda y.x)\ \texttt{in}\ (z\,0) + (z\,0) \\ \text{und}\quad t_2 \equiv\ & \texttt{let}\ z = \lambda y.(\texttt{let}\ x = \texttt{choice}\ 0\ 1\ \texttt{in}\ x)\ \texttt{in}\ (z\,0) + (z\,0) \end{aligned}$$

werden bei einer Interpretation als Reduktionsgraph auf denselben Graphen G aus Abbildung 8.1 abgebildet. Jedoch unterscheiden sich die Mengen möglicher Resultate bei beiden Ausdrücken, wenn sie mittels des Λ_{let}-Kalküls[1] ausgewertet werden. Beim ersten Ausdruck entspricht die Menge möglicher Resultate $\{0, 2\}$ beim zweiten Ausdruck $\{0, 1, 2\}$.
Somit verhindert der Nichtdeterminismus offensichtlich eine direkte Interpretation von Ausdrücken des Λ_{let}-Kalküls als Reduktionsgraphen. Die Ursache dafür ist ein teilweises "Verschlucken" von Bindungsinformationen, die auf Ausdrucksebene vorhanden sind, auf Graphenebene jedoch verloren gehen. Eine vertiefende Betrachtung dieses Problemkreises wäre im Bezug auf implementatorische Fragestellungen von großem Interesse.

---

[1] Numerik kann beim Λ_{let}-Kalkül wie beim klassischen Lambda-Kalkül mittels der Church-Zahlen (siehe [Bar84], Def. 6.4.4) erhalten werden.

Literaturverzeichnis

[Abr90] Abramsky, Samson: *The Lazy Lambda Calculus.* In: Turner, D. (Herausgeber): *Research Topics in Functional Programming*, Seiten 65–116. Addison-Wesley, 1990.

[AC79] Astesiano, Egidio und Gerardo Costa: *Sharing in Nondeterminism.* In: *Proceedings of the 6th International Colloquium on Automata, Languages, and Programming*, Band 71 der Reihe *LNCS*, Seiten 1–15. Springer-Verlag, 1979.

[ACCL90] Abadi, Martín, Luca Cardelli, Pierre-Louis Curien und Jean-Jacques Lévy: *Explicit Substitutions.* In: *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, Seiten 31–46, San Francisco, California, Januar 1990. Journal version is [ACCL91].

[ACCL91] Abadi, M., L. Cardelli, P.-L. Curien und J.-J Lévy: *Explicit substitutions.* J. functional programming, 4(1):375–416, 1991.

[Ach96] Achten, Peter: *Interactive functional programs: models, methods and implementation.* Doktorarbeit, Computer Science Department, University Nijmegen, Februar 1996.

[AF97] Ariola, Z.M. und M Felleisen: *The call-by-need lambda calculus.* J. functional programming, 7(3):265–301, 1997.

[AFM$^+$95] Ariola, Zena M., Matthias Felleisen, John Maraist, Martin Odersky und Philip Wadler: *The Call-by-Need Lambda Calculus.* In: *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Seiten 233–246, San Francisco, California, Januar 1995.

[ANS85] ANSI: *Programming Languages - COBOL*, 1985. ANSI X3.23-1985 (R1991).

[AO93] Abramsky, S. und C.-H. L. Ong: *Full abstraction in the lazy lambda calculus.* Information and Computation, 105:159–267, 1993.

[AP95] Achten, Peter und Rinus Plasmeijer: *The ins and outs of Clean I/O.* Journal of functional programming, 5(1):81–110, 1995.

[Aug84] Augustsson, L.: *A compiler for lazy ML.* In: *Proceedings of the ACM Symposium on Lisp and Functional Programming*, Seiten 218–227, Austin, August 1984.

[Bar84] Barendregt, H. P.: *The lambda calculus its syntax and semantics.* North-Holland, Amsterdam, 1984.

[BEvG+87] Barendregt, H. P., M. C. J. D. Eekelen van, J. R. W. Glaubert, J. R. Kennaway, M. J. Plasmeijer und M. R. Sleep: *Term graph rewriting.* In: Bakker, J. W., A. J. Nijman und P. C. Treleaven (Herausgeber): *Proc. Parallel Architectures and Languages Europe*, Band 259 der Reihe *Lecture Notes in Computer Science*, Seiten 141–158. Springer-Verlag, 1987.

[BvOK98] Bezem, Marc, Vincent van Oostrom und Jan Willem Klop: *Diagram Techniques for Confluence.* Information and Computation, 141(2):172–204, März 1998.

[BW88] Bird, Richard und Philip Wadler: *Introduction to Functional Programming.* Prentice-Hall International Series in Computer Science. Prentice-Hall International, 1988. Japanese translation, 1991. Dutch translation, 1991. German translation, 1992.

[CF91] Crank, Erik und Matthias Felleisen: *Parameter-Passing and the Lambda Calculus.* In: *Proc. 18th ACM Symposium on Principles of Programming Languages*, Seiten 233–245, 1991.

[CH98] Carlsson, M. und T. Hallgren: *Fudgets – Purely Functional Processes with Applications to Graphical User Interfaces.* Doktorarbeit, Department of Computing Sciences, Chalmers University of Technology and University of Gothenborg, Gothenborg, Sweden, März 1998.

[Cli82] Clinger, William: *Nondeterministic call by need is neither lazy nor by name.* In: *ACM Symp. on Lisp and Functional Programming*, Seiten 226–234. ACM, 1982.

[DB72] De Bruijn, N.: *Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation.* Indag. Mat., 34:381–392, 1972.

[DC96] Di Cosmo, R.: *On the power of simple diagramms.* In: Ganzinger, Harald (Herausgeber): *Proc. 7th Int. Conf. on Rewriting Techniques and Applications*, Band 1103 der Reihe *LNCS*, Seiten 200–214. Springer-Verlag, 1996.

[DM79] Dershowitz, N. und Z. Manna: *Proving Termination with multiset orderings.* Communications of the ACM, 22:465–476, 1979.

[ES91] Ellis, Margaret A. und Bjarne Stroustrup: *The Annotated C++ Reference Manual.* Addison-Wesley Publishing Company, Mai 1991.

[FF89] Felleisen, M. und D. P. Friedman: *A syntactic theory of sequential state.* Theoretical Computer Science, 69(3):243–287, 1989.

[FH92] Felleisen, Matthias und Robert Hieb: *The revised report on the syntactic theories of sequential control and state.* Theoretical Computer Science, 102:235–271, 1992. Original version in: Technical Report 89-100, Rice University, June 1989.

[Fit90] Fitting, Melving: *First-order logic and automated theorem proving.* Springer-Verlag, New York, 1990.

[FPJ95] Finne, Sigbjorn und Simon Peyton Jones: *Composing Haggis.* In: *Proc. of the Fifth Eurographics Workshop on Programming Paradigms for Computer Graphics*, Maastricht, September 1995. Springer Verlag.

[FW87] Fairbairn, J. und S. C. Wray: *TIM: A simple, lazy abstract machine to execute supercombinators.* In: Kahn, G. (Herausgeber): *Functional Programming Languages and Computer Architecture*, Band 274 der Reihe *Lecture Notes in Computer Science*, Seiten 34–45. Springer-Verlag, Portland, Oregon, 1987.

[Gal87] Gallier, Jean H.: *Logic for computer science: Foundations of Automatic Theorem Proving.* John Wiley & Sons, Inc., 1987.

[Gor92] Gordon, A.D.: *Functional Programming and Input/Output.* Doktorarbeit, University of Cambridge Computer Laboratory, 1992. Published in the series Distinguished Dissertations in Computer Science, Cambridge University Press, 1994.

[Gor93] Gordon, A.D.: *An operational semantics for I/O in a lazy functional language.* In: *Conf. on Functional Programming and Computer Architecture 1993*, Seiten 136–145. ACM Press, 1993.

[HA77] HENNESSY, M. C. B. und E. A. ASHCROFT: *Parameter-passing mechanism and non determinism.* In: *Proceedings of the 9th ACM Symp. Theory of Computing*, Seiten 306–311, Boulder, Colorado, Mai 1977.

[HAB+96] HAMMOND, K., L. AUGUSTSSON, B. BOUTEL, W. BURTON, J. FAIRBAIRN, J. FASEL, A. GORDON, M. GUZMÁN, J. HUGHES, P. HUDAK, T. JOHNSSON, M. JONES, D. KIEBURTZ, R. NIKHIL, W. PARTAIN, J. PETERSON, S. PEYTON JONES und P WADLER: *Report on the programming language Haskell 1.3.* Technischer Bericht Department of Computer Science, University of Glasgow, 1996.

[HM95] HUGHES, J. und A. K. MORAN: *Making Choices Lazily.* In: *Proc. of FPCA'95, ACM Conference on Functional Programming Languages and Computer Architecture*, Seiten 108–119. ACM Press, Juni 1995.

[HNSSH97] HUTCHISON, N.W.O., U. NEUHAUS, M. SCHMIDT-SCHAUSS und C.V HALL: *Natural Expert: A Commercial Functional Programming Environment.* J. of Functional Programming, 7(2):163–182, 1997.

[HP86] HINDLEY, J. R. und SELDIN J. P.: *Introduction to Combinators and λ-Calculus.* Cambridge University Press, Cambridge, 1986.

[HPW+92] HUDAK [ED.], PAUL, SIMON L. PEYTON JONES [ED.], PHILIP WADLER [ED.], BRIAN BOUTEL, JON FAIRBAIRN, JOSEPH FASEL, MARÍA M. GUZMÁN, KEVIN HAMMOND, JOHN HUGHES, THOMAS JOHNSSON, DICK KIEBURTZ, RISHIYUR NIKHIL, WILL PARTAIN und JOHN PETERSON: *Report on the Programming Language Haskell. A Non-strict Purely Functional Language. Version 1.2*, 1992.

[Hue80] HUET, G.P.: *Confluent reductions: Abstract properties and applications to term rewriting systems.* J. of the ACM, 27:797–821, 1980.

[Hug89] HUGHES, J.: *Why Functional Programming Matters.* Computer Journal, 32(2):98–107, 1989.

[ISO97] ISO/IEC: *Information technology - Programming languages - FORTRAN - Part 1: Base language*, 1997. ISO/IEC 1539-1:1997.

[JGF96] JONES, SIMON PEYTON, ANDREW GORDON und SIGBJORN FINNE: *Concurrent Haskell.* In: *Conference Record of POPL '96: The* 23$^{\text{rd}}$ *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Seiten 295–308, St. Petersburg Beach, Florida, 21–24 Januar 1996.

[Joh84] JOHNSSON, T.: *Efficent compilation of lazy evaluation.* In: *Proceedings of the ACM Conference on Compiler Construction*, Seiten 58–69, Montreal, Juni 1984.

[Kön36] KÖNIG, D.: *Theorie der endlichen und unendlichen Graphen.* Teubner, Leipzig, 1936.

[Koo90] KOOPMAN, P. W. M.: *Functional Programs as Executable Specifications.* Doktorarbeit, University of Nijmegen, 1990.

[KR64] KREIDER, D. L. und R. W. RITCHIE: *Predictably computable functions and definition by recursion.* Zeitschrift für mathematische Logik und Grundlagen der Mathematik, 10:65–80, 1964.

[KR77] KERNIGHAN, BRIAN W. und DENNIS M. RITCHIE: *The C Programming Language.* Prentice-Hall International, 1977.

[KSS98] KUTZNER, ARNE und MANFRED SCHMIDT-SCHAUSS: *A Nondeterministic Call-by-Need Lambda Calculus.* In: *International Conference on Functional Programming 1998*, Seiten 324–335. ACM Press, 1998.

[Lan64] LANDIN, P. J.: *The mechanical evaluation of expressions.* Computer Journal, 6(4), 1964.

[Las98] LASSEN, S. B.: *Relational Reasoning about Functions and Nondeterminism.* Doktorarbeit, Department of Computer Science, Aarhus University, Mai 1998.

[Lau93] LAUNCHBURY, J: *A natural semantics for lazy evaluation.* In: *Proc. 20th Principles of Programming Languages*, 1993.

[MAE$^+$62] MCCARTHY, JOHN, PAUL W. ABRAHAMS, DANIEL J. EDWARDS, TIMOTHY P. HART und MICHAEL I. LEVIN: *LISP 1.5 Programmer's Manual.* MIT Press, Cambridge, Mass., 1962.

[McC61] MCCARTHY, J.: *A basis for a mathematical theory of computations.* In: *Proc. Western Joint Computer Conference*, Seiten 225–238, Mai 1961.

[Mey88] MEYER, BERTRAND: *Object-oriented Software Construction.* Prentice-Hall International, Cambridge, Great Britain, 1988.

[Mil77] MILNER, R.: *Fully abstract models of typed λ-calculi.* J. Th. Computer Science, 4:1–22, 1977.

[Mog89] MOGGI, E.: *Computational lambda-calculus and monads.* In: *Proceedings 4th Annual Symposium on Logic in Computer Science*, Seiten 14–23, Washington, 1989. IEEE Computer Society Press.

[Mog91] MOGGI, E.: *Notions of computations and monads.* Information and Computation, 93(1):55–92, 1991.

[Mor68] MORRIS, J.H.: *Lambda-Calculus Models of Programming Languages.* Doktorarbeit, MIT, 1968.

[Mor98] MORAN, A. K.: *Call-by-name, Call-by-need, and McCarthy's Amb.* Doktorarbeit, Department of Computing Science, Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden, September 1998.

[MOW98] MARAIST, JOHN, MARTIN ODERSKY und PHILIP WADLER: *The Call-by-Need Lambda calculus.* J. of Functional programming, 8(3):275–317, Mai 1998.

[MS99] MORAN, A. K. und D. SANDS: *Improvement in a Lazy Context: An Operational Theory for Call-By-Need.* In: *Proc. POPL'99, the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Seiten 43–56. ACM Press, Januar 1999.

[MSC99] MORAN, A. K., D. SANDS und M. CARLSSON: *Erratic Fudgets: A semantic theory for an embedded coordination language.* In: *Coordination '99*, Band 1594 der Reihe *Lecture Notes in Computer Science.* Springer-Verlag, April 1999.

[MTH90] MILNER, R., M. TOFTE und R. HARPER: *The definition of Standard ML.* MIT Press, 1990.

[New42] NEWMAN, M.H.A.: *On theories with a combinatorial definition of equivalence.* Annals of Mathematics, 2:223–243, 1942.

[Ong92] ONG, C.-H. L.: *Lazy Lambda Calculus: Theories, Models and Local Structure Characterisation.* In: *Proc. of 19th International Colloquium on Automata, Programming and Languages*, Band 623 der Reihe *LNCS.* Springer-Verlag, 1992.

[Ong93] ONG, C.-H. L.: *Non-Determinism in a Functional Setting.* In: *Proc. 8th IEEE Symposium on Logic in Computer Science (LICS '93)*, Seiten 275–286. IEEE Computer Society Press, 1993.

[PHA+97] PETERSON [ED.], J., K. HAMMOND [ED.], L. AUGUSTSSON, B. BOUTEL, W. BURTON, J. FASEL, A. D. GORDON, J. HUGHES, P. HUDAK, TH. JOHNSSON, M. JONES, E. MEIJER, S. PEYTON JONES, A. REID und P. WADLER: *Report on the Programming Language Haskell: A Non-strict, Purely Functional Language, Version 1.4*, 1997.

[Pit99] PITCHER, CORIN: *Functional Programming and Erratic Non-Determinism.* Doktorarbeit, Programming Research Group, Oxford University Computing Laboratory, Oxford, United Kingdom, 1999. To appear.

[PJ87] PEYTON JONES, SIMON L.: *The Implementation of Functional Programming Languages.* Prentice-Hall International, London, 1987.

[PJ92] PEYTON JONES, SIMON L.: *Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine.* J. functional programming, 2(2):127–202, April 1992.

[PJW93] PEYTON JONES, SIMON L. und PHILIP WADLER: *Imperative Functional Programming.* In: *Proceedings 20th Symposium on Principles of Programming Languages, Charleston, South Carolina,*, Seiten 71–84. ACM, 1993.

[Plo75] PLOTKIN, G. D.: *Call-by-name, call-by-value and the λ-Calculus.* Theoretical Computer Science, 1:125–159, 1975.

[Plo76] PLOTKIN, G. D.: *A Powerdomain Construction.* SIAM J. Comput., 5(3):452–487, 1976.

[Plo77] PLOTKIN, G. D.: *LCF considered as a programming language.* Theoretical Computer Science, 5(3):223–255, 1977.

[PvE98] PLASMEIJER, R. und M. VAN EEKELEN: *Concurrent Clean: Version 1.3.* Technischer Bericht Dept. of Computer Science, University of Nijmegen, 1998.

[RCe86] REES, J. und W. CLINGER (EDS.): *The revised report on the algorithmic language Scheme.* ACM SIGPLAN Notices, 21(12):37–79, 1986.

[Ros98] ROSE, KRISTOFFER: *Explizit Substitution Annotated Bibliography.* available at ftp://ftp.ens-lyon.fr/pub/users/LIP/krisrose/ESAB/, April 1998. Revision 1.4.

[RS59] RABIN, M. O. und D. SCOTT: *Finite automata and their decision problems.* IBM Journal of Research, 3(2):115–125, 1959.

[Ses97] SESTOFT, P.: *Deriving a lazy abstract machine.* Journal of Functional Programming, 7(3):231–264, Mai 1997.

[SI96] SEAMAN, JILL und S. PURUSHOTHAMAN IYER: *An operational semantics of sharing in lazy evaluation.* Science of Computer Programming, 27(3):289–322, November 1996.

[SS92] SØNDERGARD, H. und P. SESTOFT: *Non-determinism in functional languages.* The Computer Journal, 35(5):514–523, 1992.

[SS96] SCHMIDT-SCHAUSS, M.: *A Partial Rehabilitation of Side-Effecting I/O: Non-Determinism in Non-Strict Functional Languages.* Technischer Bericht Fachbereich Informatik, Universität Frankfurt, Germany, 1996. available at http://www.ki.informatik.uni-frankfurt.de.

[Sto77] STOY, JOSEPH E.: *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, Cambridge, Massachusetts, and London, England, 1977.

[Tho96] THOMPSON, S.: *Haskell: The Craft of Functional Programming.* Addison-Wesley, 1996.

[Tur37] TURING, A. M.: *The* $\mathfrak{p}$*-functions in* λ*-*K*-conversion.* J. Symbolic Logic, 2:164, 1937.

[Tur90] TURNER, D.: *An overview of Miranda.* In: TURNER, D. (Herausgeber): *Research Topics in Functional Programming.* Addison Wesley, 1990.

[vO94] OOSTROM, VINCENT VAN: *Confluence by Decreasing Diagrams.* Theoretical Computer Science, 126(1):259–280, April 1994.

[Wad71] WADSWORTH, C. P.: *Semantics and Pragmatics of the Lambda Calculus.* Doktorarbeit, Oxford University, 1971.

[Wad95] WADLER, P.: *Monads for functional programming.* In: JEURING, J. und E. MEIJER (Herausgeber): *Advanced Functional Programming,* LNCS 925, Seiten 24–52. Springer-Verlag, 1995.

Tabellarischer Lebenslauf

Arne Kutzner
Am Alten See 21
60489 Frankfurt/Main

05.03.1966	geboren in Bielefeld/Westfalen
1972-1976	Besuch der Grundschule in Bielefeld
1976-1985	Besuch des Gymnasiums zuerst in Bielefeld ab 1977 in Frankfurt/Main
1985	Abitur
1985-1994	Studium der Informatik an der Johann Wolfgang Goethe-Universität Frankfurt/Main Diplomarbeit unter Prof. Dott. Ing. Roberto Zicari mit dem Titel *Modellierung von SQL3 in EIFFEL: Das SQL-Environment* Abschluß als Diplom-Informatiker.
1994-1999	Promotion an der Professur für Künstliche Intelligenz und Softwaretechnologie unter Prof. Dr. Manfred Schmidt-Schauß. Die Promotion wurde in den Jahren 1994 bis 1996 durch ein Stipendium des Landes Hessen gefördert.