A Non-Deterministic Call-by-Need Lambda Calculus: Proving Similarity a Precongruence by an Extension of Howe's Method to Sharing

Dissertation
zur Erlangung des Doktorgrades
der Naturwissenschaften

vorgelegt beim Fachbereich 15
der Johann Wolfgang Goethe–Universität
in Frankfurt am Main

von
Matthias Mann
aus Frankfurt

Frankfurt 2005
(D F 1)

vom Fachbereich Informatik und Mathematik der Johann Wolfgang Goethe–
Universität als Dissertation angenommen.

Dekan: Prof. Dr.-Ing. Detlef Krömker

Gutachter: Prof. Dr. Manfred Schmidt-Schauß
             Johann Wolfgang Goethe-Universität
             Fachbereich Informatik und Mathematik
             Robert-Mayer-Str. 11-15
             60325 Frankfurt

             Prof. David Sands, Ph.D.
             Chalmers University of Technology and University of Göteborg
             Department of Computing Science
             S-412 96 Göteborg
             Sweden

Datum der Disputation: 17. August 2005

Zusammenfassung

Gegenstand der vorliegenden Arbeit ist ein nicht-deterministischer Lambda-Kalkül mit call-by-need Auswertung. Lambda-Kalküle spielen im allgemeinen eine grundlegende Rolle in der theoretischen Informatik, sei es als Basis für Theorembeweiser im Bereich der Verifikation oder beim Entwurf sowie der semantischen Analyse von Programmiersprachen, insbesondere funktionalen.

Im Zuge der weiter zunehmenden Komplexität von Hard- und Softwaresystemen werden funktionale Programmiersprachen eine immer größere Bedeutung erlangen. Der Grund liegt darin, daß in der Softwareentwicklung stets eine gewisse Diskrepanz entsteht zwischen dem Konzept, welches ein Programmierer entwirft, und der Art und Weise, wie er es in einer bestimmten Programmiersprache umzusetzen hat.

Um robuste und verläßliche Software zu gewährleisten, sollte es eines der Hauptanliegen der Informatik sein, diese Diskrepanz so klein wie möglich zu halten. Im Vergleich zu imperativen oder objekt-orientierten ist dies bei funktionalen Programmiersprachen der Fall, weil sie aufgrund ihrer Herkunft aus der Semantik stärker am Ergebnis einer Berechnung orientiert sind. Imperative bzw. objekt-orientierte Programmiersprachen hingegen betonen oft zu stark, welche Schritte notwendig sind, um das gewünschte Ergebnis zu erhalten.

Lambda-Kalkül und funktionales Programmieren

Obwohl der λ-Kalkül die konzeptionelle Grundlage der funktionalen Sprachfamilie bildet, spiegelt er die Implementierung moderner funktionaler Programmiersprachen nicht angemessen wider. Dies trifft besonders für die sogenannten *verzögert auswertenden* funktionalen Programmiersprachen zu, d.h. solche, bei denen Argumente nur dann ausgewertet werden, wenn sie zur Ermittlung des Ergebnisses beitragen. Denn um eine Mehrfachauswertung bei der Anwendung

einer Funktion auf nicht vollständig ausgewertete Argumente zu vermeiden, verwenden alle heutzutage relevanten Implementierungen *Graphreduktion*, d.h. sie operieren nicht auf einem Termbaum sondern auf einem Graphen. Ein Beispiel mag dies verdeutlichen: Die Ausdrücke $\lambda x.(x + x)$ und $\lambda x.(2x)$ stellen beide eine Funktion dar, die ihr Argument x verdoppelt. Eine Anwendung auf z.B. das noch nicht vollständig ausgewertete Argument $(5 - 3)$ würde bei der üblichen *call-by-name* Auswertung, d.h. die Argumentausdrücke werden einfach für die Argumentvariablen eingesetzt, aber $(5 - 3) + (5 - 3)$ bzw. $2(5 - 3)$ liefern.

Offensichtlich würde dann das Ergebnis des Ausdruckes $(5-3)$ bei der ersten Variante zwei Mal ermittelt, was überflüssig ist, da das zweimalige Vorkommen des Ausdruckes $(5 - 3)$ ein- und denselben Wert bezeichnet. Indem diese Information in einem Graphen gespeichert wird, umgeht die Implementierung diese Mehrfachberechnung. Dies ist aber nur zulässig, weil in funktionalen Programmiersprachen das Prinzip der *referentiellen Transparenz* erfüllt ist, d.h. ein Ausdruck verändert während der Ausführung eines Programmes nicht seinen Wert[1] sondern wird nur in eine andere, äquivalente Form überführt.

Um die operationale Semantik einer solchen Implementierung mittels Graphreduktion entsprechend abzubilden, werden sogenannte *call-by-need* Kalküle herangezogen. Bei diesen wird durch eine geschickte Wahl der Kalkülregeln sichergestellt, daß Ausdrücke erst kopiert werden können, wenn sie in einer bestimmten Form vorliegen, die man als Basiswert bezeichnen könnte. Der Charakter der verzögerten Auswertung bleibt hierbei freilich erhalten, d.h. eine Funktionsanwendung vermag sofort ein Ergebnis zu liefern, sobald alle relevanten Argumentausdrücke ausgewertet sind. Für das genannte Beispiel bedeutet das, daß die Anwendung von $\lambda x.(x + x)$ auf das Argument $(5 - 3)$ zuerst in einen Ausdruck der Form `let` $x = (5 - 3)$ `in` $(x + x)$ überführt wird, wobei das `let`-Konstrukt hier die Graphdarstellung, das *Sharing*, explizit darstellt. Erst wenn das Ergebnis 2 für den Term $(5 - 3)$ ermittelt ist, wird es für x in den Rumpf $x + x$ eingesetzt und somit die Mehrfachberechnung vermieden.

Nichtdeterminismus und Gleichheit

Die mitunter wichtigste Frage in einem Lambda-Kalkül, nicht nur call-by-need betreffend, ist wohl, welche Arten von Termen kopiert werden dürfen. Wesentlich beinflußt wird hierdurch auch die Frage, welche Programmtransformationen zulässig sind bzw. welche Gleichheiten auf Programmen gelten sollen.

---

[1]Man beachte, daß diese Aussage für imperative Programmiersprachen nicht zutrifft.

Wie das oben angeführte Beispiel bereits verdeutlicht hat, sollten die beiden Ausdrücke $\lambda x.(x+x)$ und $\lambda x.(2x)$ als gleich betrachtet werden. Anderseits kann die im $\boldsymbol{\lambda}$-Kalkül übliche (β)-Regel nicht mehr uneingeschränkt gelten.

$$(\lambda x.M)N = M[N/x] \qquad (\beta)$$

Hierbei bezeichnet $M[N/x]$ die Einsetzung des Argumentes N im Term M für die Variable x, was, wir im obigen Beispiel gesehen haben, zu Mehrfachauswertungen führen kann.

Wie aber läßt sich nun ein adäquater Gleichheitsbegriff finden, wenn doch auch der Ausdruck $(5-3)+(5-3)$ dasselbe Ergebnis, nämlich 4, liefert? Es genügt also offensichtlich nicht, nur die Ergebnisse von Auswertungen zu betrachten. Einige Arbeiten verfolgen daher den Ansatz, die Anzahl der Auswertungsschritte zu zählen, was auch erfolgversprechend scheint.

Diese Arbeit jedoch schlägt einen anderen Weg ein: Es wird ein Sprachkonstrukt `pick` eingeführt, welches nichtdeterministisch eines seiner beiden Argumenten auswählt. Nichtdeterminismus in dieser Form verletzt offensichtlich die referentielle Transparenz, hilft aber im Gegenzug zu erkennen, wann Sharing erhalten bleibt bzw. wann nicht: Die beiden Ausdrücke `pick 5 3 + pick 5 3` und `let x = pick 5 3 in (x + x)` liefern nun unterschiedliche Mengen von möglichen Resultaten, nämlich $\{6, 8, 10\}$ gegenüber $\{6, 10\}$ für den `let`-Ausdruck.

Darüberhinaus gibt es eine Vielzahl von Anwendungsgebieten für nichtdeterministische Lambda-Kalküle, u.a. als theoretische Grundlage für nebenläufige Prozesse oder von deklarativer, Seiteneffekt-behafteter Ein-/Ausgabe. Insbesondere im Bezug auf das letztere sind an der Professur schon einige Arbeiten durchgeführt worden, z.B. die Dissertation von Arne Kutzner oder der technische Bericht zum Kalkül FUNDIO, zu denen sich diese als eine Ergänzung bzw. Fortführung versteht.

Denn die Kenntnis über die in einem Kalkül geltenden Gleichheiten spielt eine herausragende Rolle für dessen Verständnis. Die bisherigen Arbeiten stützen sich dabei hauptsächlich auf den Begriff der *kontextuellen Gleichheit*, durch die zwei Terme s und t miteinander identifiziert werden, wenn deren Auswertung in allen möglichen Programmkontexten $C[\ ]$ dasselbe Terminierungsverhalten, gekennzeichnet mit $\Downarrow$, zeigt:

$$s \simeq_c t \iff (\forall C : C[s] \Downarrow \iff C[t] \Downarrow)$$

Die kontextuelle Gleichheit stellt ein mächtiges Werkzeug für die Beurteilung der Korrektheit von Programmtransformationen dar, insbesondere weil sie per

Definition eine *Kongruenz* ist, d.h. ihre Gleichheiten wiederum in Kontexte einsetzbar sind, also die Implikation $s \simeq_c t \implies (\forall C : C[s] \simeq_c C[t])$ erfüllt. Mitunter sind Gleichheiten aber schwierig nachzuweisen, weil ja die Terminierung der Auswertung in allen möglichen Kontexten betrachtet werden muß.

Daher ist es oftmals sinnvoll, einen anderen Äquivalenzbegriff zu Hilfe zu nehmen. Die Rede ist von der Methode der *Bisimulation*, welche einen stärker stufenweisen Gleichheitstest beschreibt. Hier werden zwei Terme als gleich betrachtet, wenn sie auf allen möglichen Stufen dasselbe Verhalten zeigen, solange man also nicht das Gegenteil nachweisen kann. Die Technik fand zuerst im Bereich der Zustandsübergangssysteme Anwendung, ist mittlerweile aber auf dem Gebiet der funktionalen Programmierung und Lambda-Kalküle gut etabliert und existiert in verschiedenen Ausprägungen. Der Stil, den Abramsky *applicative bisimulation* nennt, ist am einfachsten in Form einer Präordnung $\lesssim$ zu illustrieren.

$$s \lesssim t \iff (s \Downarrow \lambda x.s' \implies (t \Downarrow \lambda x.t' \land \forall r : (\lambda x.s')\ r \lesssim (\lambda x.t')\ r))$$

Dies ist nicht etwa eine zyklische Definition, sondern als Fixpunktgleichung zu verstehen, über welcher der größte Fixpunkt gebildet wird. Dadurch wird das Beweisprinzip der Co-Induktion anwendbar.

Setzt man $\sim = \lesssim \cap \gtrsim$ so ist leicht zu sehen, daß $\sim \subseteq \simeq_c$ gefolgert werden kann, vorausgesetzt $\sim$ ist eine Kongruenz. Gilt die Einsetzbarkeit in Kontexte für eine Präordnung, so wird von einer *Präkongruenz* gesprochen. Es ist also zu zeigen, daß die Relation $\lesssim$, genannt *similarity*, eine Präkongruenz ist. Nicht nur dieser Beweis sondern bereits der Entwurf der passenden Relation für einen nichtdeterministischen call-by-need Lambda-Kalkül stellt einen wesentlichen Erkenntnisgewinn dar.

Denn der übliche Ansatz für die Similarity-Definition ist nicht ohne weiteres auf call-by-need übertragbar. Dies liegt daran, daß die Resultate der Auswertung hier `let`-Umgebungen mit unausgewerteten Termen, in diesem Fall auch nichtdeterministische Auswahlen, umfassen können. Aus diesem Grund wird in dieser Arbeit ein weiterer Kalkül, der sogenannte „Approximationskalkül", definiert, bei dem die Resultate der Auswertung durch Mengen von Abstraktionen, also Termen der Form $\lambda x.s$, dargestellt werden.

Dies ist die wesentliche Grundlage, um Similarity überhaupt definieren zu können. Die eigentliche Herausforderung besteht nun darin, den Präkongruenzbeweis zu führen. Hierfür wird die grundlegende Methode von Howe, die für eine ganze Klasse von Sprachen anwendbar ist, als Ausgangspunkt genommen und entsprechend erweitert, damit Sharing berücksichtigt wird.

Übersicht

Kapitel 2 ist somit auch den *lazy computation systems* gewidmet, für die Howe einen allgemeinen Präkongruenzbeweis entwickelt hat. Dazu wird eine spezielle Relation, der Präkongruenzkandidat, eingeführt, welcher per Definition einsetzbar in Kontexte ist, aber nicht notwendigerweise transitiv. Grundlage des Beweises ist zu zeigen, daß der Präkongruenzkandidat transitiv ist und dadurch mit der zugrunde liegenden Präordnung zusammenfällt, insgesamt also eine Präkongruenz vorliegt.

Dazu wird eine Fortsetzung von Relationen auf offenen Termen benötigt, die Howe in seiner ursprünglichen Arbeit über alle schließenden Substitutionen erklärt. Dies ist mit Sharing offensichtlich nicht vereinbar. Daher wird hier von einer konkreten Definition für die Erweiterung von Relationen auf offenen Termen abstrahiert und der Beweis mit Hilfe der *Zulässigkeit* einer solchen offenen Erweiterung geführt. Bedingt dadurch kann ein Substitutions-Lemma für den Präkongruenzkandidaten in der allgemeinen Form, in der es bei Howe aufgestellt wird, nicht hergeleitet werden. Stattdessen folgen, sobald die entsprechenden Begriffe definiert sind, zwei spezialisierte Substitutions-Lemmata.

Daher wird in Kapitel 3 erst einmal der $\boldsymbol{\lambda}_{\mathrm{ND}}$-Kalkül, der eigentliche Untersuchungsgegenstand dieser Arbeit, vorgestellt. Er verfügt u.a. über ein `let` für die Darstellung von Sharing, `seq` für die Sequentialisierung der Auswertung und das bereits angesprochene `pick`-Konstrukt für die nichtdeterministische Auswahl.

Anhand eines Beispiels wird, wie schon erwähnt, aufgezeigt, daß die übliche Definition von Similarity nicht möglich ist. Deshalb wird in Abschnitt 3.2 der Kalkül $\boldsymbol{\lambda}_{\approx}$ eingeführt, in dem Similarity definiert werden kann, weil die Resultate von Auswertungen reine Abstraktionen ohne `let`-Umgebung sind. Dies wird mittels einer zusätzlichen Konstante $\odot$ bewerkstelligt, wodurch der Auswertungsprozeß abgeschnitten werden kann. Auf diese Weise wird ermöglicht, in den `let`-Umgebungen mit einer quasi variablen Ressourcenbeschränkung beliebig lange auszuwerten.

Die entscheidende Aufgabe von Kapitel 3 besteht darin zu zeigen, daß $\boldsymbol{\lambda}_{\mathrm{ND}}$-Terme in der Tat durch entsprechende Mengen von $\boldsymbol{\lambda}_{\approx}$-Termen repräsentiert werden können. Dies wird durch das *Approximation Theorem* bewerkstelligt, eines der bedeutendsten Resultate dieser Arbeit.

In Kapitel 4, dem wohl herausragendsten Teil dieser Arbeit, wird der Präkongruenzbeweis für Similarity im $\boldsymbol{\lambda}_{\approx}$-Kalkül geführt, welcher schließlich im *Precongruence Theorem* mündet. Außerdem werden an dieser Stelle auch die zuvor genannten Substitutions-Lemmata aufgestellt. Sie spielen beim Beweis,

daß der Präkongruenzkandidat unter Reduktion mittels der (cpa)-Regel des $\boldsymbol{\lambda}_{\approx}$-Kalküls stabil ist, eine wesentliche Rolle. Stabilität des Präkongruenzkandidaten unter Reduktion wird in Abschnitt 4.3.1 bewiesen und bildet eine der wichtigsten Grundlagen, um die Ergebnisse aus Kapitel 2 anwenden zu können.

Kapitel 5 behandelt die Gleichheitstheorie, die von $\simeq_c$, der kontextuellen Gleichheit, induziert wird. Hier trägt das Precongruence Theorem bei, Similarity als Beweismethode für die kontextuelle Präordnung, das ist die der kontextuellen Gleichheit zugrunde liegende Präordnung $\lesssim_c$, nutzbar zu machen. D.h. im *Main Theorem* wird gezeigt, daß die Erweiterung von Similarity auf offene Terme in der kontextuellen Präordnung enthalten ist. Similarity kann aber nicht zur vollständigen Charakterisierung der kontextuellen Präordnung verwendet werden, weil die o.g. Inklusion strikt ist. Das bedeutet, es gibt geschlossene Terme, für die zwar $s \lesssim_c t$ aber nicht $s \lesssim t$ gilt. Der Nachweis dieser Eigenschaft ist nicht trivial und führt über den Begriff der *syntaktischen Stetigkeit*. Im wesentlichen bedeutet dies, daß der Fixpunktkombinator $\mathbf{Y}$, mit dessen Hilfe sich bekanntlich rekursive Funktionen definieren lassen, zur Berechnung von kleinsten oberen Schranken aufsteigender Ketten von Termen eingesetzt und dabei durch „Kontexte hindurch geschoben" werden kann. Bezüglich der kontextuellen Präordnung bzw. Gleichheit ist diese Aussage zutreffend, bezüglich Similarity jedoch nicht. Dies wird in Abschnitt 5.2 näher ausgeführt.

In Kapitel 6 werden mögliche Erweiterungen des Basiskalküls besprochen. Um diese Arbeit möglichst unkompliziert und zielgerichtet durchführen zu können, verfügt der $\boldsymbol{\lambda}_{\mathrm{ND}}$-Kalkül weder über Datentypen noch über rekursive Bindungen in seinem `let`-Konstrukt. Die Betrachtungen in Abschnitt 6.1 erscheinen vielversprechend, daß Datentypen, also `case` und Konstruktoren, ohne größere Schwierigkeiten zum Kalkül hinzugefügt werden können. Für rekursive `let`-Bindungen hingegen wird in Abschnitt 6.2 gezeigt, daß sie nicht einfach mittels eines Fixpunktkombinators, wie z.B. $\mathbf{Y}$, dargestellt werden können. Hier sind also noch eingehendere Untersuchungen notwendig.

Abschließend ist zu sagen, daß die vorliegende Arbeit eine erweiterbare und vielseitig anwendbare Methodik beschreibt, wie Bisimulations-Relationen in Lambda-Kalkülen mit `let` implementiert werden können, so daß Einsetzbarkeit in Kontexte gewährleistet ist.

Preface

This dissertation reports on research which has been carried out under the supervision of Prof. Dr. Manfred Schmidt-Schauß during the years 2000 to 2005 at Johann Wolfgang Goethe-Universität Frankfurt. Some of the results from the dissertation are moderate extensions of work published in [Man04, Man05].

Acknowledgements

Many people have given me great support while working on this thesis. First of all I would like to express my gratitude to Manfred Schmidt-Schauß who aroused my interest in call-by-need lambda calculi. He was always keen to discuss the progress of this work and provided me with new insights. I am glad to have worked with him and grateful for having been his student. Without his guidance and motivation all the time this thesis would not have come into existence.

Special thanks go to Søren Lassen for his valuable input and to David Sands for his interest in this work. I also benefitted a lot from the extensive discussions with my colleague David Sabel and his constructive comments. I would like to thank Angelika Schifignano for her assistance in many of my duties which helped me immensely to concentrate on this work.

Last, but not least, I am particularly indebted to Anke Eggelbusch for her incredible patience and for letting me participate in her vitality. I wish to thank Sebastian Meiss for always giving me pleasure and Elfriede Meiss as well as Andrea Münch for their helpful advice on language use. This research would have been impossible without the education, which I owe to my parents.

Contents

Chapter 1

Introduction

The complexity of computer systems increases excessively and this is unlikely to change in the near future. While formal verification of hardware has become quite standard, the vast majority of the existing software is not rigorously founded on mathematical models. This is even more surprising as lots of rather critical applications are affected. However, most of the software systems have been growing over years and a thorough verification is often not feasible.

Nonetheless, in order to manage complexity, formal methods will become inevitable in many areas. To a certain extent some of the techniques from extreme programming already point into this direction e.g., by regarding a test as a specification rather than as a necessity which may be deferred.

Although aiming at development processes mainly, extreme programming is oriented towards object-oriented programming. Like imperative programming languages, object-oriented programming is largely based on the so-called *von Neumann computer* model, which emphasises the concept of a *state* that is changed by a sequence of instructions. Though this view may be suitable for some application domains like graphical user interfaces or databases, it does not appear to capture very well complex relations, which are encoded in many applications, cf. [BS02]. Hence there is a kind of semantic gap, i.e., how the solution of a certain task is reflected by its translation into a specific programming language. This gap is not expected to be filled in completely, but computer science shall make it as small as possible.

In his famous article [Bac78], Backus proposes a paradigm shift and develops basic algebraic laws for establishing the correctness of *functional* programs. Instead of manipulating a state by a sequence of instructions, functional pro-

gramming is based on *evaluation*, i.e., expressions are taken into an explicit form by a sequence of transformations. During execution of a functional program, every expression retains its value, a principle that is called *referential transparency*. Hence it is sound to "replace equals by equals" and thus functional programming benefits from algebraic reasoning. In [FH88, p. 11], a small example illustrates that this is not the case for imperative languages. We conclude that for functional programming languages the aforementioned semantic gap is smaller than for imperative and object-oriented programming languages.

Both programming paradigms, the functional as well as the imperative, have their roots in computability theory. In the first half of the 20th century Church [Chu36] and Turing [Tur36] introduced two distinct formal models of computation. While the *Turing machine* concentrates on the aspects of symbol manipulation w.r.t. a store and order of execution, the λ-*calculus*, cf. [Chu41], is a system for representing functions and their application to arguments. The computational power of the λ-calculus stems from the fact that functions itself may be arguments to other functions. As turned out, Turing machines and λ-calculus have the same expressiveness. Hence the so-called "Church-Turing thesis" states that all sensible notions of computability are equivalent. Of course, this cannot be proven for *every* possible formalism, but it has been performed for every of the subsequent ones, e.g., random access machines. So the Church-Turing thesis is generally accepted.

Turing has shown that the *halting problem* for Turing machines is undecidable, i.e., there is no Turing machine which, given the input and the encoding of a second Turing machine, always returns whether the second Turing machine stops on its input. By the Church-Turing thesis this is also valid for the λ-calculus, hence it is undecidable if the evaluation of an arbitrary λ-expression terminates. However, unlike Turing machines the λ-calculus does not depend on a certain evaluation order. The reason is that evaluation is based on *reduction* according to a set of rules and there are in general several possibilities when or where a rule may be applied. The *Church-Rosser Theorem* says that the results will be identical whenever evaluation terminates in two different ways. This property is also called *confluence* and will be discussed in section 2.3.1 in more detail. A further significant result due to Church and Rosser is that a certain evaluation order, namely the *normal-order*, terminates at least as often as any other evaluation order does.

As both formalisms, λ-calculus and Turing machines, capture the notion of computability, they might both be used to model the semantics of programming languages. However, because of its very detailed description, a specification by a Turing machine would be rather tedious. But following the underlying pattern,

the notion of an *abstract machine* has become a quite successful technique for specifying the *operational* semantics of programming languages. Also the λ-calculus has proven to be useful in operational semantics, i.e., describing the execution behaviour of programs in a certain language, cf. [Lan65a, Lan65b].

However, the λ-calculus had perhaps a greater impact on *denotational* semantics, where each language construct a function is assigned to as its respective denotation. Since the λ-calculus was designed as a formal system to reason about functions, it seems quite right for this purpose. From the use of the lambda notation as a semantic language it is not a big step to a "real" programming language. Therefore today's functional programming languages are still closely related to their foundation, i.e., the λ-calculus. Furthermore, semantic languages are used to describe *what* the meaning of a program is and hence the functional paradigm emphasises *what* has to be computed. This is contrary to imperative programming for which it is necessary to specify *when* and *where* the state, i.e., storage, should be manipulated.

LISP was the first programming language inspired from the λ-calculus. It is a *hybrid* language rather than a pure functional one, i.e., contains functional as well as imperative elements. Such hybrid programming languages are not uncommon and can be found in object-oriented programming, too. Several functional programming languages like, e.g., Erlang or ML and its derivatives SML and OCaml, permit imperative elements, i.e., side-effects. All the aforementioned languages are *strict*, i.e., before a function is called, its arguments are evaluated. Therefore the independence of the evaluation order from the λ-calculus is more or less abandoned. For a sequential implementation this has the advantage that side-effects become manageable. But it would be of no help, if arguments were evaluated in parallel.

Thus *pure* functional programming languages are of particular interest. The absence of side-effects promotes the design of *non-strict* languages, where arguments need *not* to be evaluated before function calls take place. Non-strictness enables concise representations of possibly infinite data structures such as "the list of all primes". In order to implement this, an evaluation strategy is required, that yields termination whenever termination is possible. In [Tur79] Turner describes such an evaluation strategy which corresponds to the normal-order reduction mentioned before, but avoids duplication of work by using graph reduction. Since then progress has been made towards efficient implementations of non-strict functional programming languages, cf. [Aug84, PJ87, PJS89, PvE93].

Influenced from Milner's work [Mil78] on polymorphic type systems, the non-strict functional programming language Miranda, cf. [Tur85], was developed. Nowadays all relevant implementations of non-strict functional programming

languages respect "sharing", i.e., they employ graph reduction to avoid dupli-
cation of work. Among these, Haskell [PJ03] and Clean [PvE99] are probably
the most popular. However, because of sharing these language implementa-
tions possess an operational semantics which is not reflected accurately by the
reduction in the original $\boldsymbol{\lambda}$-calculus.

Therefrom the main motivation arises for dealing with a call-by-need lambda
calculus in this work: Such a calculus represents sharing explicitely and thus may
act as an operational semantics for modern functional programming languages.
Furthermore, the specific calculus of this work provides a non-deterministic
language construct which may evaluate to either of its arguments. In this form,
non-determinism serves two purposes. On one hand, it is crucial for modelling
side-effecting I/O in non-strict functional programming languages. The Natural
Expert [HNSSH97] programming language proposes such a declarative style and
also for Haskell [Sab03] side-effecting I/O has been implemented. Its theoretical
foundations have been explored in [Kut99, SS03a].

On the other hand, the presence of non-determinism helps to detect when
sharing is violated. In order to say that sharing is violated requires to know
when two terms should be equivalent. So for the understanding of a calculus
it is vital to have a sensible notion of equality. It is the merit of this work to
develop such a notion, namely *mutual open similarity*, and to establish it as
a powerful proof tool. Although the combination of non-determinism, call-by-
need evaluation and mutual open similarity as an equational theory seems quite
useful, there has not been much research in this area yet. A reason might be
that this is a non-trivial topic which requires an extensive treatment. Hopefully,
this will change once this work has laid the foundations.

1.1   Survey and Related Work

As stated before, lambda calculi are ubiquitous in theoretical computer science
and fundamental to the study of programming language semantics and design,
in particular for functional programming languages. Apart from this, lambda
calculi have a great influence on automated theorem proving as well as on formal
methods for system analysis.

Numerous facets of lambda calculi exist therefore: typed and untyped, with
and without data constructors, deterministic and non-deterministic, with differ-
ent evaluation strategies and notions of what a "value" is, i.e., how far reduction
should proceed. E.g., Andrews [And02] builds a higher-order logic on top of sim-
ply typed lambda calculus which in turn forms the basis for the Isabelle theorem

prover, cf. [Pau89]. Or, in [Bar91], Barendregt describes various type systems
for lambda calculus and their correspondence to certain logics. Out of these, the
system $\lambda P\omega$, also called the "calculus of constructions", is used as a foundation
for the theorem prover Coq, cf. [CH88].

Equality plays a prominent role in all these areas and apparently there are
several forms. Some of these will be discussed in section 1.1.3. *Contextual
equivalence* due to [Mor68] is especially worth mentioning. It discriminates
terms by their behaviour in all term contexts, typically termination, cf. [Mil77].

Before we consider equality and, in section 1.1.2, some variants of the the-
ory, we will examine the "classical" lambda calculus in section 1.1.1 first. The
standard reference for the classical lambda calculus is [Bar84], a comprehensive
book. Though it treats many aspects like models and consistent extensions, it
mainly focuses on the lambda calculus as a formal system in which equations
between lambda terms are deduced using inference rules. This yields the so-
called notion of convertibility, i.e., two terms are considered equivalent, if they
can be transformed to each other according to these rules.

There, the evaluation of terms accords to the call-by-name strategy insofar
that arguments of function applications are simply substituted for the formal
parameter and thus copied. Since arguments of functions need not be evaluated
in advance, call-by-name may result in the duplication of work.

In order to overcome this, some work on call-by-need lambda calculi has
already been done. One of the most important questions for call-by-need is
which kind of terms may be copied. To get an impression of the right answer
for this question we advocate either to let contextual equivalence regard the
number of reduction steps as in [MS99] or to use a non-deterministic calculus.
Based on the calculus from [Kut99] this work pursues the latter and will develop
a behavioural equivalence that is sound w.r.t. the semantics of the language
which is defined in terms of contextual equivalence.

The behavioural equivalence which is referred to is similarity, or more pre-
cisely, mutual open similarity. It is based on the technique of bisimulation,
which is due to Park [Par81] and Milner [Mil71] and has frequently been ap-
plied to functional programming, cf. [Abr90, San91, Gor94a, Gor99]. Bisimula-
tion emphasises that two programs or expressions are equivalent as long as no
difference can be discovered by a possibly infinite series of experiments. For a
process calculus like in [Mil89] such an experiment usually comprises observ-
ing the respective actions performed by two processes whereas in functional
programming it involves evaluation. Although experiments are therefore not
decidable in general, bisimulation describes a more stepwise procedure to tackle
the equivalence of terms than contextual equivalence. Furthermore, because of

its definition as a greatest fixed point of some suitable operator, bisimulation is accessible by the proof technique of coinduction [Gor94b, NNH99].

However, employing bisimulation, or rather mutual open similarity, for the correctness of program transformations requires substitutivity, i.e., the ability to "replace equals by equals" within every program or term. An equivalence which adheres to this principle is called a congruence. Proving congruence for bisimulation-like relations is in general a non-trivial task. This is in particular true for mutual open similarity within the non-deterministic call-by-need calculus λ_{ND} of this work which establishes a completely new result. Therefore the calculus λ_{ND} is intentionally designed as simple and concise as possible. That means, as a fundamental study this work will devise the notions and methods for the congruence proof. Once the technique has been developed it should be possible to transfer it to more sophisticated calculi.

1.1.1 Classical λ-calculus

The classical λ-calculus as set out in [Bar84] has a relatively simple syntactic structure, but it is a powerful system. Its building blocks are *variables*, *abstractions* and *applications*. The *terms* are given by the symbol E in the grammar

$$E ::= V \mid (\lambda V.E) \mid (E\ E)$$

Figure 1.1: Syntax for terms in the λ-calculus

of figure 1.1 where V stands for a countable set of variables. The rationale is to regard an abstraction $\lambda x.M$ as an anonymous function whose argument variable is x whereas a term $M\ N$ stands for the application of M to its argument N.

The formal theory of the λ-calculus consists of axioms and inference rules from which *equations* are deduced, see [Dav89, Chapter 5] for an introductory presentation. Complemented by rules inducing a congruence, i.e., an equivalence relation for which substitutivity holds, the basic *conversion rules* look like

$$(\lambda x.M) = (\lambda y.M[y/x]) \tag{α}$$
$$(\lambda x.M)N = M[N/x] \tag{β}$$

where $M[N/x]$ stands for the capture-free substitution of N for x in M. Note that a variable x becomes *bound* in an abstraction $\lambda x.M$, otherwise it is considered *free*. Hence the rule (α) needs the side-condition that y must *not* occur

free in M. However, usually the rule (α) is abandoned in favour of regarding terms as syntactically equal up to renaming of bound variables. Therefore, it is customary to adopt a convention for bound variables, cf. [Bar84, p. 26].

1.1.2 Variants of the Theory

Regarding the rule (β) as directed from left to right, the notion of *reduction* is obtained. This means, a term is rewritten according to the (β)-rule until it is of a certain syntactic form. E.g., if a term does not contain any (β)-*redex*, i.e., a position where the (β)-rule can be applied, it will be called a *normal form*. Most functional programming languages do not require reduction to a normal form but stop as soon as an abstraction is reached, which is said to be a *weak head normal form*. Abramsky calls this style of evaluation "lazy" in [Abr90].

This question of what has to be considered a value, is associated with the issue which form an argument must have in order to apply (β)-reduction. If the rule does not impose any restriction like in its original form above, we speak of *call-by-name*. It permits to implement normal-order as a reduction strategy.

This is contrary to *call-by-value*, where the term N in the (β)-rule above is demanded to be a value — however a value might be defined. The *applicative-order* evaluation strategy of strict functional languages corresponds closely to call-by-value. The "parametric lambda calculus" in [RDRP04] is an attempt to give a uniform account of these variants.

It is clear that there are situations where call-by-value requires more reduction steps to obtain a result than call-by-name. But the converse is also true, since by the substitution $M[N/x]$ in the rule (β) the term N may be duplicated and thus all the reductions necessary to evaluate N, too. In order to get "the best out of both worlds" call-by-name is combined with sharing, which yields *call-by-need* lambda calculi, e.g. [AFM+95, AF97, MOW98]. These usually employ a `let`-construct to represent the sharing of arguments explicitly, but it is also possible to do without such an extra syntax, cf. [AF97]. However, the reduction rules and the strategy have to be adapted in such a way that a function may be applied immediately but its argument is not copied until it represents a value. As an example consider the terms $\lambda x.(x + x)$ and $\lambda x.(2 * x)$ both representing functions which double their argument x. If the application to the argument $(5-3)$ was evaluated call-by-name using the (β)-rule, we would obtain $(5 - 3) + (5 - 3)$ and $2 * (5 - 3)$ respectively.

Obviously, the result of the term $(5 - 3)$ would be computed twice in the former case. According to referential transparency this seems redundant, as both occurrences of $(5 - 3)$ represent the same value. So in a call-by-need

calculus the application $(\lambda x.(x + x))\ (5 - 3)$ is usually transformed into a term of the form `let` $x = (5 - 3)$ `in` $(x + x)$ first. Nothing will be substituted for x in the body $x + x$ until for the term $(5 - 3)$ the result 2 has been computed. Thus duplication of reductions is avoided.

1.1.3 Equality

The equivalence of the classical **λ**-calculus is *convertibility*, i.e., two terms are equivalent when they can be transformed to each other according to the conversion rules of the calculus. As mentioned before, conversion is usually permitted inside arbitrary contexts, i.e. program fragments. Therefore convertibility allows to "replace equals by equals" which means that it is a congruence.

This is also the case for the call-by-need calculi named before. These papers emphasise that their call-by-need evaluation *implements* call-by-name correctly. This implies that all equivalences which hold in the call-by-name setting are also valid w.r.t. call-by-need evaluation. We believe that this should *not* be the case: Duplicating terms which have to be further reduced contradicts the idea of call-by-need. Particularly the original copying (β)-rule can therefore not be valid in general. On the other hand, Sands et al. show in [SGM02] for call-by-value calculi that convertibility can only relate terms of the same asymptotic complexity. Since this statement remains true in a call-by-need setting, an equational theory based on convertibility is too limited for proving useful optimisations correct.

So what about the situation with contextual equivalence? This notion is due to Morris, cf. [Mor68], and plays an important role for the correctness of program transformations. Unlike convertibility it does not depend directly on the reduction rules of a calculus. Rather, terms are discriminated by observing termination in all possible contexts. Thus with $\Downarrow$ denoting termination while C stands for program contexts, contextual equivalence $\simeq_c$ can be expressed as

$$ s \ \simeq_c \ t \ \Longleftrightarrow \ (\forall C : \ C[s] \Downarrow \Longleftrightarrow \ C[t] \Downarrow) $$

It is by definition a congruence, i.e., $s \simeq_c t \ \Longrightarrow \ C[s] \simeq_c C[t]$, and provides a justification for the reduction rules, if they preserve contextual equivalence.

However, the results of [MOW98, Theorem 32] and [AF97, Theorem 5.11] indicate that the contextual equivalences of call-by-name and call-by-need agree in their calculi. The reason for this is that we have considered the fact of termination only regardless of "how fast" it happens. Therefore an improvement theory like in [MS99] would probably lead to differing contextual equivalences.

This work opts for another possibility which is to examine a non-deterministic lambda calculus. If non-deterministic choices are copied, this will have a potential influence on termination. Moreover, non-deterministic lambda calculi have numerous applications in concurrency and parallelism as well as for modelling side-effecting I/O in non-strict functional programming languages. Some of these will be discussed in section 1.1.4. That's why the investigation of equality in such a call-by-need calculus is of particular interest: To gain a deeper understanding of sharing in a non-deterministic lambda calculus, a combination which is non-trivial, cf. [MSC99a]. This is one of the contributions of this work.

Similarity

Establishing contextual equivalence is rarely straightforward, thus sometimes a further equivalence is involved. What we mean here is a behavioural equivalence based on the technique of *bisimulation* due to Park [Par81] and Milner [Mil71]. Compared to contextual equivalence it provides a more stepwise approach for proving equations. The approach is generally applicable to *labelled transition systems*, e.g., automata. Also process calculi like CCS, cf. [Mil89], or the pi-calculus, even in a polymorphic setting, cf. [PS00], can be represented in this style. And this is the case for functional programming languages and lambda calculi as their operational semantics, too, cf. [Abr90, Gor99].

The equivalence may intuitively be described as follows: Whenever some action takes such a system from a state to certain descendant, this action must also be possible within the second system leading to an equivalent state. In this way bisimulation describes equivalence by always performing only a single action at a time and deferring to the next step the check, whether the results are equivalent or not. Note that this is not a cyclic definition because the equivalence can properly be constructed as the greatest fixed point of a suitable relation. So let P, P', Q, Q' stand for arbitrary states and $\xrightarrow{\alpha}$ denote the transition by some action α. Bisimulation equivalence $\sim$ may then be depicted as below.

$$P \sim Q \iff (\forall \alpha : \ P \xrightarrow{\alpha} P' \iff Q \xrightarrow{\alpha} Q' \ \wedge \ P' \sim Q')$$

This emphasises that two states are regarded equivalent as long as no difference is discovered by a (possibly infinite) series of actions. In functional programming languages or lambda calculi such an action involves the evaluation of a term, as will be discussed in section 2.4. Proofs of bisimulation equivalence may rely on coinduction, cf. [Gor94b, NNH99], a very powerful method. Roughly, it amounts to showing $\forall \alpha : \ P \xrightarrow{\alpha} P' \iff Q \xrightarrow{\alpha} Q' \ \wedge \ (P', Q') \in R$ for some relation R under the premise that $(P, Q) \in R$ holds. In some cases this precondition may

be so strong that the proof becomes nearly trivial whereas a direct proof for contextual equivalence is rather subtle.

However, in order to employ bisimulation as a tool for establishing contextual equivalences it has to be a congruence. This is vital, even if bisimulation is used on its own, e.g., for showing correctness of program transformations. But as, e.g., the work in [Abr90, How89, How96] demonstrates, proving bisimulation a congruence is in general a complex effort. In particular for non-deterministic call-by-need λ-calculi, up to now there has not been much research in this respect. Strictly speaking this work does not treat bisimulation but rather mutual similarity. This means that the greatest fixed point is taken of an operator which yields, instead of an equivalence relation, a preorder, namely similarity, see section 2.4. It will be shown that this preorder is respected by insertion into arbitrary contexts, i.e., similarity is a *precongruence*. Note that bisimulation and mutual similarity do not necessarily coincide in a non-deterministic environment, cf. [Gor99, LP00], whereas this is true for deterministic calculi.

For the "lazy" lambda calculus Abramsky shows in [Abr90] that applicative bisimulation is a congruence. Treating a deterministic call-by-name λ-calculus, his work employs domain theory usually known from denotational semantics. However, there are various purely operational approaches, some also devising rule formats so that congruence of bisimulation holds for all systems whose operational semantics is specified in this format.

So e.g., the GDSOS rule format of [San97] is deterministic, which ensures the orthogonality of the corresponding rewrite system, yet its strength lies in the proof principles it provides. Moreover, the *tyft/tyxt* format in [GV92] is too restricted to represent the calculus of section 3.1. On the other hand, the *promoted tyft/tyxt* format, cf. [Ber98], might be able to capture the normal-order reduction of our λ_{ND}-calculus. However, its proof obligations seem to amount to a direct precongruence proof in this case.

Even if the operational semantics of the λ_{ND}-calculus cannot directly be represented as a *Structured Evaluation System*, cf. [How96], the approach of Howe in [How89, How96] is trailblazing. First, it works for a wide range of languages called "lazy computation systems" while providing still some freedom for the definition of the evaluation relation. Secondly, the method permits non-determinism in a straightforward way. Thirdly, and likely most importantly, the approach has a clear potential for call-by-need, although it was not primarily designed to deal with sharing. The results presented in this work provide ample support for the universality of Howe's technique.

1.1.4   Extended Lambda Calculi

As described before, various extensions of the $\boldsymbol{\lambda}$-calculus have been the subject
of research. We have already quoted call-by-need calculi with or without a `let`-
construct. In particular worth mentioning is also PCF, cf. [Plo77], which is typed
and can be seen as a functional "core" language. There are numerous variants
containing numbers, bools or pairs and sometimes also lists. PCF is often used as
a vehicle for studying semantic properties of functional programming languages.

The extension with constructors and `case` discussed in section 6.1 could be
considered a PCF-like language in some respect. Apart from this, the calcu-
lus which is the subject of this work is rather different. First, PCF is typed
whereas the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus is untyped. This is no shortcoming because we view
the calculus $\boldsymbol{\lambda}_{\mathrm{ND}}$ as an intermediate product in the compilation of a high-level
language that has already been type-checked. However, regarded as the core of a
functional programming language, PCF lacks a representation of sharing which
is usually present in an implementation. On the other hand, the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus
has call-by-need evaluation.

Furthermore, by means of its non-deterministic choice it provides a foun-
dation for side-effecting I/O in non-strict functional programming languages as
the recent work in [Sab03, SS03a] demonstrates. Sharing is inevitable for such
an approach. Consider the example terms $\lambda x.(x + x)$ and $\lambda x.(2 * x)$ from sec-
tion 1.1.2 again. These should be considered equivalent as they both double
their argument. Now suppose a non-deterministic function `ASK`, which returns
for each call some user input. A similar construct has been implemented in
Natural Expert, cf. [HNSSH97]. The terms $(\text{ASK} + \text{ASK})$ and $(2 * \text{ASK})$ may yield
different results. For `let` $x = \text{ASK}$ `in` $(x + x)$ and $(2 * \text{ASK})$ this is not the case.

Non-deterministic Lambda-calculi

A number of questions have to be clarified when introducing a non-deterministic
construct into a programming language or, e.g., a lambda calculus. Apart from
the classification of non-determinism as e.g. in [SS92], we consider the decision
what kind of terms may be copied a major issue.

As the example above has shown, non-determinism in languages without
sharing, i.e. retaining a copying (β)-rule like [Ong93, San94, dP95, LP00],
is completely different from our work because it will distinguish $\lambda x.(x + x)$
from $\lambda x.(2 * x)$. Likewise is the situation with explicit substitution-calculi,
cf. [ACCL91], since substitutions are duplicated by distributing them over ap-
plications. In [Bou94] this is the case, too.

As mentioned before, the calculi in [AFM$^+$95, AF97, MOW98] realise explicit sharing and restrict copying to abstractions but are deterministic. Moreover, their equational theory is based on convertibility rather than on contextual equivalence. The calculi in [KSS98, MSC99a, SS03a, SSSS04] all provide a non-deterministic choice, sharing and contextual equivalence, albeit designed for various purposes. E.g., in [MSC99a] Moran et al. describe the semantics of "stream processors" which are the foundation of the Fudget combinators for graphical user interfaces in Haskell. The work [SSSS04] successfully employs a non-deterministic choice for representing set descriptions.

The last-named calculi roughly represent the direction of our investigations, though there are a few differences. Since these papers do not discuss congruence of bisimulation, it seems sensible to carry out our studies in a rather elementary calculus. Hence, like [KSS98, Kut99], the λ_{ND}-calculus has a non-recursive `let` only, whereas the remaining calculi provide recursive bindings. Furthermore, in contrast to [MSC99a, SS03a, SSSS04], the calculus λ_{ND} neither has a `case` nor data constructors.

1.2   Outline

The aim of this work is twofold. First it develops a general framework for proving similarity a precongruence in a call-by-need lambda calculus. Therefore in chapter 2 the pioneering method of Howe [How89, How96], which introduces a certain precongruence candidate relation, will be adapted to cope with sharing. In particular this concerns the way a relation over closed terms is continued on open terms. In [How89, How96] this is done using all closing substitutions, a view which is obviously incompatible with sharing. Because of this a general substitution lemma for the precongruence candidate like [How96, Lemma 3.2] is not possible. We will obtain specialised versions of this lemma but their proof has to be deferred. The reason is that we will parameterise the open extension of a relation and extract the notion of admissibility from Howe's original precongruence proof.

The second contribution of this work is to explore non-determinism. This is not only a vehicle in order to demonstrate proving similarity a precongruence in a highly non-trivial calculus but is also valuable to gain knowledge about the effect of sharing on the equational theory. So chapter 3 contains a thorough treatment of two non-deterministic call-by-need lambda calculi. In section 3.1 the λ_{ND}-calculus, which is the intrinsic subject of our study, will be introduced. The calculus has a `let`-construct to express sharing, a `seq`-operator for sequen-

tial evaluation and a non-deterministic choice called `pick`. In this calculus, the usual technique to define similarity, namely by reducing terms to a weak head normal form and applying these weak head normal forms to arbitrary fresh arguments, will not work, as we will see. But finding the right definition of similarity is not easy for a call-by-need calculus in the presence of non-determinism. The reason is that weak head normal forms in a call-by-need setting usually possess a `let`-environment which may contain unevaluated non-deterministic choices.

The solution proposed in this work is to represent each weak head normal form by a set of abstractions from which the `let`-environments have been eliminated. In order to do this, we must evaluate within the `let`-bindings — but when to stop? The answer is: At any point! This means it may be necessary to evaluate terms arbitrarily deep. We therefore introduce the $\boldsymbol{\lambda}_{\approx}$-calculus in section 3.2 which is equipped with the special constant $\odot$ that stops evaluation when substituted for a (sub-) term. This approach seems somewhat related to [Wad78] where the denotation of a term may be represented as the limit of its "approximate normal forms" though Wadsworth used the technique for the construction of $\boldsymbol{\lambda}$-calculus models.

The challenge of chapter 3 is to show that a $\boldsymbol{\lambda}_{\mathrm{ND}}$-term may be indeed "approximated" by terms of the $\boldsymbol{\lambda}_{\approx}$-calculus. This result will be achieved by the Approximation Theorem which says that termination of evaluation in both of the calculi coincides. Since weak head normal forms in the calculus $\boldsymbol{\lambda}_{\approx}$ are abstractions which do not carry around a `let`-environment, similarity can be defined in the usual manner. This is accomplished in chapter 4 which is probably the most central part of this work. Proposition 4.1.27 shows that the formulation of similarity conforms to the style of Abramsky's applicative bisimulation.

Figure 1.2 illustrates the dependencies of the essential proof steps up to the Precongruence Theorem which states that the open extension of similarity in the $\boldsymbol{\lambda}_{\approx}$-calculus is a precongruence. It is the significant achievement of chapter 4 and requires a series of supporting results. E.g., the aforementioned substitution lemmas are established in section 4.2.2. This is possible only after similarity, its open extension and the precongruence candidate in the $\boldsymbol{\lambda}_{\approx}$-calculus have been defined. Section 4.3.1 is worth mentioning, too. There, the precongruence candidate relation is shown to be stable under reduction. This is one of the major preconditions for the application of the results from chapter 2 in the proof of the Precongruence Theorem.

Chapter 5 covers the equational theory generated by contextual equivalence and depicts a denotational semantics for the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus. The Main Theorem in section 5.1.2 shows that the open extension of similarity implies contextual

preorder and thus establishes mutual open similarity, its symmetrisation, as a proof tool for contextual equivalence. This is true for the calculus $\boldsymbol{\lambda}_{\approx}$ as well as for $\boldsymbol{\lambda}_{\mathrm{ND}}$ because by virtue of the Approximation Theorem the respective contextual equivalences are shown to agree in section 5.1.1. However, mutual open similarity is strictly contained in, but not identical to, contextual equivalence.

This is the statement of Proposition 5.1.6 whose proof is established in section 5.2 involving the notion of *syntactic continuity*. By means of a counter-example it is shown that syntactic continuity holds for contextual preorder but fails for open similarity. This result is not easy to achieve as it involves several non-trivial properties of reductions from chapter 3.

Finally, chapter 6 discusses possible extensions of the base calculus such as recursive bindings or `case` and constructors.

Precongruence Theorem:
$(\lesssim_b)^o$ is a
precongruence

Theorem 2.2.13

$(\widehat{\lesssim_b})_0 \subseteq \lesssim_b$

$(\lesssim_b)^o$ is
admissible

$(\widehat{\lesssim_b})_0$ stable
under reduction

Substitution
Lemmas 4.2.7
and 4.2.8

Lemma 2.2.10

$(\cdot)^o$ is preorder-
preserving
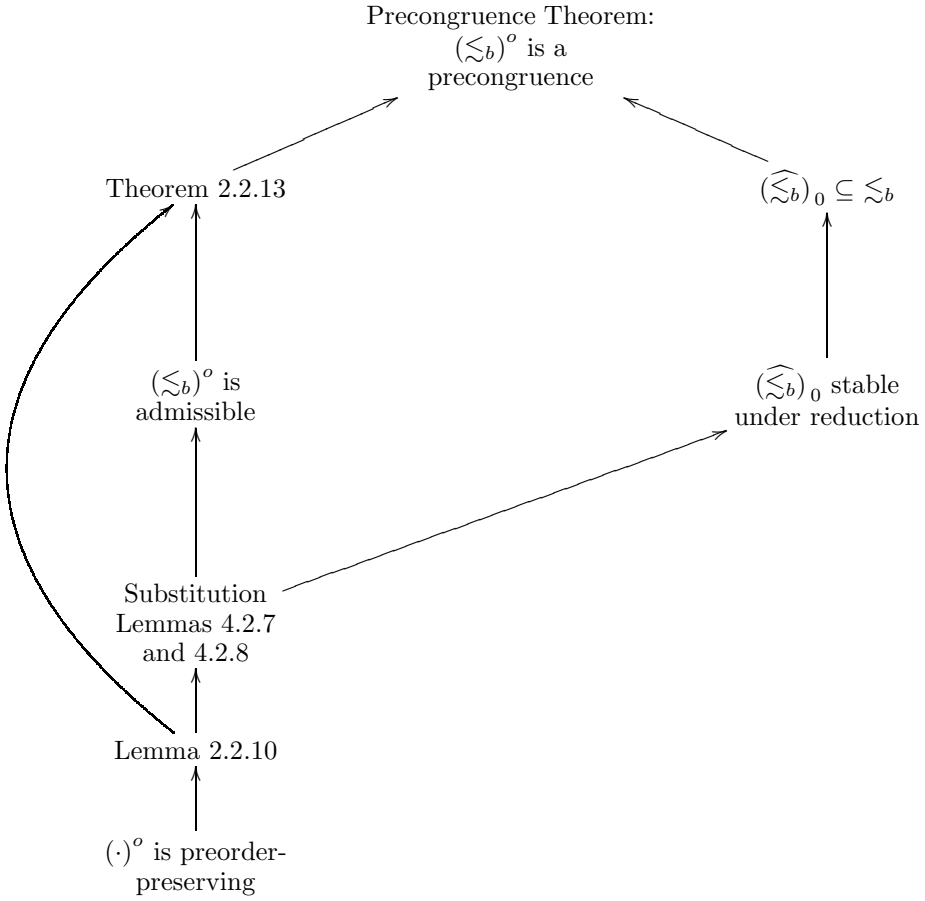
Figure 1.2: Structure of the precongruence proof

Chapter 2

Lazy Computation Systems

In this chapter, a general framework is being developed, under which a preorder can be shown to be compatible with contexts. This lays the foundation for the behavioural equivalence of the non-deterministic call-by-need lambda calculus in the remainder of this work.

Therefore, Howe's original work [How89, How96] on the so-called *lazy computation systems* will be extended. Though a priori it is capable to deal with non-determinism, it was not primarily intended for sharing which comes with call-by-need evaluation. By means of a lazy computation system the syntax and operational semantics of a language can be specified in an abstract way. This approach to higher-order syntax is fairly standard, cf. [KvOvR93, MN98] for combinatory reduction and higher-order rewrite systems. However, their focus is on confluence rather than behavioural equivalence.

For dealing with simulations and proving them a precongruence, certain rule formats, e.g. [Ber98, GV92, San97], for structural operational semantics have been proposed. As explained in section 1.1.3, these formats are all too limited for representing the calculus which is subject to this work.

Even though Howe originally allowed substitutions of arbitrary terms, his technique is flexible enough to cope with call-by-need. Sands has already demonstrated the extensibility of Howe's approach in [San91] by applying it to improvements, but without sharing. Moreover, Gordon [Gor94a, Gor99] makes use of the technique for typed programs.

This chapter treats the adoption of this method to non-deterministic call-by-need lambda calculi. Thus, it does not simply reproduce the corresponding notions and results but rather enhances them. Moreover, an attempt is made

to unravel some of the features of preorders and their impact on the proofs. We will therefore devise the notion of admissibility for the extension of a relation to open terms. It reveals its central role in the proof of theorem 2.2.13.

In section 2.1 we first introduce the concept of a lazy computation language, before section 2.2 addresses preorders and establishes sufficient conditions to make up a precongruence. In section 2.3 we then define reduction and evaluation, i.e. the operational semantics for lazy computation languages, thus yielding lazy computation systems. The notion of a simulation will be generally treated in section 2.4, where also coinductive proof principles are explained.

2.1   Language

The presentation of higher-order abstract syntax follows [How96] closely, i.e. the terms of the language are determined by the *operators* which impose a variable-binding structure.

Definition 2.1.1 (Lazy Computation Language). *Let $\mathcal{L} = (O, \alpha)$ be a signature such that $\alpha(\tau) \in \{\langle k_1, \ldots, k_n \rangle \mid \forall 1 \le i \le n \in \mathbb{N} : k_i \in \mathbb{N}_0\}$ for every $\tau \in O$ holds. Then $\mathcal{L}$ is called a* <u>*la*</u>*zy* <u>*c*</u>*omputation* <u>*l*</u>*anguage (lcl for short) and O its set of* operators *with α denoting their respective* arity*.*

The arity $\alpha(\tau)$ of an operator $\tau \in O$ is a sequence of non-negative integers in order to record the number of arguments as well as the number of variables, which become bound in every argument position. It is also possible to introduce separate *kinds* of variables, cf. [How96] and [San97], for e.g. mixing call-by-name and call-by-value evaluation. But since we are mainly interested in call-by-need evaluation here, we will not pursue such an approach. It seems straightforward to incorporate this though.

Definition 2.1.2 (Terms and Operands). *Let $\mathcal{L} = (O, \alpha)$ be a lazy computation language and V a countable set of variables. Then the sets ${}^i\mathcal{T}(\mathcal{L})$ are inductively defined as follows:*

- *$V \subseteq {}^0\mathcal{T}(\mathcal{L})$*

- *If $t \in {}^0\mathcal{T}(\mathcal{L})$ and $x_1, \ldots, x_n \in V$ are distinct then $x_1, \ldots, x_n.t \in {}^n\mathcal{T}(\mathcal{L})$*

- *If $\tau \in O$ with arity $\alpha(\tau) = \langle k_1, \ldots, k_n \rangle$ and $t_j \in {}^{k_j}\mathcal{T}(\mathcal{L})$ for $j \in \{1, \ldots, n\}$ then $\tau(t_1, \ldots, t_n) \in {}^0\mathcal{T}(\mathcal{L})$*

The terms *of the language $\mathcal{L}$ are given by the set $\mathcal{T}(\mathcal{L}) = {}^0\mathcal{T}(\mathcal{L})$ whereas the elements of the sets ${}^n\mathcal{T}(\mathcal{L})$ with $n > 0$ are called* operands*.*

If $x_1, \ldots, x_n.t \in {}^n\mathcal{T}(\mathcal{L})$ is an operand, the notation $\overline{x}.t$ will be used as a shorthand. Note that the constitution of an operand $\overline{x}.t$ binds all *free occurrences* of variables from $\overline{x}$ in t. With $\mathcal{FV}(t)$ the set of *free variables* of a term t is designated. As usual, a term t is *closed* if all of its variables are bound, i.e. $\mathcal{FV}(t) = \emptyset$, otherwise it is called *open*. The set of closed terms will be referred to as $\mathcal{T}_0(\mathcal{L})$. If the language $\mathcal{L}$ is clear from context we will simply write $\mathcal{T}$ and $\mathcal{T}_0$ respectively in the remainder of this chapter. In addition we will also frequently use the notation $\overline{s}$ for sequences, i.e. tuples, of terms and operands.

Definition 2.1.3. *Let $x \in V$ be a variable and $s, t \in \mathcal{T}$ terms. Then $s[t/x]$ denotes the capture-free substitution of t for all free occurrences of x in s.*

This extends to tuples of terms and variables in the obvious way, i.e. the expression $s[\overline{t}/\overline{x}]$ is a synonym for $s[t_1/x_1, \ldots, t_n/x_n]$ and stands for the term s in which all free occurrences of the variables $x_1, \ldots, x_n$ in s have been substituted by the respective terms $t_1, \ldots, t_n$ simultaneously.

The technical details of these notions are quite standard and can be found in the literature [Bar84, Dav89]. Furthermore, we will write $\{x \mapsto s\}$ for the substitution $[s/x]$ when it is convenient.

In the following, we use the symbol $\equiv$ to denote syntactic equivalence up to renaming of bound variables, i.e. terms and operands will be considered syntactically equal modulo alpha-renaming. In particular for operands, this means that $\overline{x}.s \equiv \overline{y}.t$ holds if there are fresh variables $\overline{z}$ such that the syntactic equivalence $s[\overline{z}/\overline{x}] \equiv t[\overline{z}/\overline{y}]$ is true.

It is well-known, cf. [Bar84], that variable renamings have to be consistent insofar as free variables should not become bound after a substitution. To avoid such a *variable capture* the following convention is widely accepted.

Variable Convention for Terms. Throughout this work, whenever some term t is referred to, its bound variables are chosen to be distinct from each other and the free variables. Furthermore, this convention extends to sets of terms as well as terms which result from other terms, e.g., by transformations.

Example 2.1.4. *The λ-language consisting of the operators $O = \{\lambda, @\}$ with its arities $\alpha(\lambda) = \langle 1 \rangle$ and $\alpha(@) = \langle 0, 0 \rangle$ forms a lazy computation language where e.g. $@(\lambda(x.s), y)$ is an open term which would usually be denoted by $(\lambda x.s)\ y$ and $\lambda(y.@(y, y))$ is a closed term, written $\lambda y.yy$ in the common notation.*

Example 2.1.5. *With $O = \{\lambda, \mathtt{let}, @\}$ a simple $\mathtt{let}$-language is declared. The arities of λ and $@$ are given in example 2.1.4, while $\alpha(\mathtt{let}) = \langle 1, 0 \rangle$ models the non-recursive $\mathtt{let}$ which binds one variable. Hence a term like $\mathtt{let}\ x = t\ \mathtt{in}\ s$ would be expressed as $\mathtt{let}(x.s, t)$.*

In the end of this section, the notion of a *context* will be introduced. Roughly, a context C is a term with a hole $[\ ]$ and $C[s]$ stands for the resulting term where s has been plugged into the hole of C. Contexts are defined analogously to definition 2.1.2 where $[\ ]$ denotes the empty context.

Definition 2.1.6 (Contexts). *Let $\mathcal{L}$ be a lcl and $\mathcal{T}(\mathcal{L})$ its set of terms. Then the set $\mathcal{C}(\mathcal{L}) = {}^0\mathcal{C}(\mathcal{L})$ of contexts over $\mathcal{L}$ is inductively defined as follows.*

- $[\ ] \in {}^0\mathcal{C}(\mathcal{L})$

- *If $t \in {}^0\mathcal{C}(\mathcal{L})$ and $x_1, \ldots, x_n \in V$ are distinct then $x_1, \ldots, x_n.t \in {}^n\mathcal{C}(\mathcal{L})$*

- *If $\tau \in O$ with arity $\alpha(\tau) = \langle k_1, \ldots, k_n \rangle$ and $C_i \in {}^{k_i}\mathcal{C}(\mathcal{L})$ for some $1 \leq i \leq n$ and $t_j \in {}^{k_j}\mathcal{T}(\mathcal{L})$ for $j \in \{1, \ldots, i-1, i+1, \ldots, n\}$ then $\tau(t_1, \ldots, t_{i-1}, C_i, t_{i+1}, \ldots, t_n) \in {}^0\mathcal{C}(\mathcal{L})$.*

We may omit the language $\mathcal{L}$ whenever there is no risk of confusion and write $\mathcal{C}$ for the set of all such single-hole contexts. We sometimes also consider *multi-contexts*, i.e. contexts with multiple, distinguishable holes. If C, D are contexts then $C[D]$ denotes the context resulting from inserting D into C's hole.

The essential feature of contexts is to possibly capture free variables of the term plugged into the hole. Therefore, contexts are *not* identified up-to renaming of bound variables.

Example 2.1.7. *Regard the lcl of example 2.1.5 again. Then $\mathtt{let}(x.[\ ], t)$ and $\mathtt{let}(y.[\ ], t)$ denote different contexts. Also $\lambda(x.@(s, [\ ]))$ is a context.*

The formation of terms and contexts is conveniently be specified by a context free grammar. Using this, we define the following compositions of contexts.

Definition 2.1.8. *Let $\mathcal{D}$ be denoting some set of contexts. Then the sets $\mathcal{D}^*$, $\mathcal{D}^+$, $\mathcal{D}^k$, $\mathcal{D}^{m \vee n}$ are defined by the corresponding symbols for every $D \in \mathcal{D}$:*

$$
\begin{aligned}
D^0 &::= [\ ] &\qquad D^{k+1} &::= D[D^k] \\
D^+ &::= D \ \mid \ D[D^+] &\qquad D^{m \vee n} &::= D^m \ \mid \ D^n \\
D^* &::= D^0 \ \mid \ D^+
\end{aligned}
$$

For ease of notation, we will write e.g. $D^{m \vee n}$ denoting a context $D \in \mathcal{D}^{m \vee n}$.

Definition 2.1.9. *Any subset $\mathcal{D} \subseteq \mathcal{C}$ of contexts is closed under composition if and only if for all $C, D \in \mathcal{D}$ also $C[D] \in \mathcal{D}$ holds.*

2.2 Preorders and the Precongruence Candidate

This section will introduce the method of Howe [How89, How96] for proving a preorder to be compatible with contexts, i.e. a precongruence. The key idea is to define a so-called "precongruence candidate" relation which is by definition reflexive and compatible with contexts, but not necessarily transitive. Once it has been shown that the precongruence candidate coincides with the underlying preorder, a reflexive, transitive and compatible relation is obtained.

So we first define these notions as well as how they extend to operands and tuples of terms and operands. Hence, for the remainder of this section we fix a lazy computation language $\mathcal{L}$ and its set $\mathcal{T}$ of terms.

Definition 2.2.1. *Let* $\eta \subseteq \mathcal{T} \times \mathcal{T}$ *be an arbitrary relation over terms. Then for operands* $\overline{x}.s, \overline{y}.t \in {}^n\mathcal{T}$ *we declare*

$$\overline{x}.s \; \eta \; \overline{y}.t \;\; \overset{def}{\iff} \;\; \exists \overline{z} : \; s[\overline{z}/\overline{x}] \; \eta \; t[\overline{z}/\overline{y}] \tag{2.2.1}$$

and for sequences of terms or operands furthermore

$$\overline{s_i} \; \eta \; \overline{t_i} \;\; \overset{def}{\iff} \;\; \forall i : \; s_i \; \eta \; t_i \tag{2.2.2}$$

For $\eta \subseteq \mathcal{T}^2$ *a relation,* $(\eta)_0 \overset{def}{=} \eta \cap \mathcal{T}_0^2$ *defines its* restriction to closed terms.

Note that by monotonicity of set-intersection the inclusion $(\eta_1)_0 \subseteq (\eta_2)_0$ follows from $\eta_1 \subseteq \eta_2$ immediately.

Definition 2.2.2 (Preorder). *A relation* $\eta \subseteq \mathcal{T}^2$ *is called a* preorder *if it is reflexive and transitive, i.e.,* $s \; \eta \; s$ *and* $r \; \eta \; s \; \wedge \; s \; \eta \; t \implies r \; \eta \; t$ *hold.*

A relation $\eta \subseteq \mathcal{T}^2$ is *compatible (with contexts)* if for every context $C \in \mathcal{C}$ from $s \; \eta \; t$ also $C[s] \; \eta \; C[t]$ is implied. A *precongruence* is a compatible preorder.

Definition 2.2.3 (Precongruence). *Let* $\eta \subseteq \mathcal{T}^2$ *be a preorder. Then* η *is called a* precongruence *if and only if for all terms* $s, t \in \mathcal{T}$ *the following holds:*

$$s \; \eta \; t \implies \forall C \in \mathcal{C} : \; C[s] \; \eta \; C[t] \tag{2.2.3}$$

If (2.2.3) *is valid and* η *additionally is symmetric, then* η *is a* congruence.

Example 2.2.4. *Apparently, the syntactic equivalence* $\equiv$ *is a congruence. Since a congruence, e.g. in [Bar84, p. 50], is also called* equality, *the syntactic equivalence* $\equiv$ *will usually be referred to as* syntactic equality.

By induction on the structure of contexts, it is easily shown that the following, more gradual definition yields an identical notion. We may therefore freely interchange the terms compatible and operator-respecting throughout this work.

Definition 2.2.5. *A relation $\eta \subseteq \mathcal{T}^2$ is said to be* operator-respecting *if and only if $\overline{a}_i \ \eta \ \overline{b}_i$ implies $\tau(\overline{a}_i) \ \eta \ \tau(\overline{b}_i)$ for all operands a_i, b_i and operators $\tau \in O$.*

As mentioned before, the precongruence candidate relation will be operator-respecting by construction. In order to define it, we need an idea of how a preorder $\eta \subseteq \mathcal{T}_0$ over closed terms should behave on open ones. The crucial point is that such a continuation of η on $\mathcal{T}$, the full set of terms, must in general *not* be carried out using all closing substitutions. The following example illustrates that this possibly violates sharing.

Example 2.2.6. *Consider the lcl from example 2.1.5, where* let *represents the explicit sharing of terms. Let $\Upsilon(\cdot)$ be a measure on terms that counts all (distinct) variables[1] and applications. Then relate all closed terms w.r.t. this measure, i.e. $\eta = \{(s,t) \in \mathcal{T}_0{}^2 \mid \Upsilon(s) \leq \Upsilon(t)\}$ is clearly reflexive and transitive.*

Now consider the open terms $@(x, @(x, y))$ and $@(x, @(y, y))$ for which obviously $\Upsilon(@(x, @(x, y))) = 4 = \Upsilon(@(x, @(y, y)))$ holds. After applying the substitution $\sigma = \{x \mapsto @(\lambda a.a, \lambda b.b), y \mapsto \lambda c.c\}$ to both of these terms, this relation is lost, i.e. $\Upsilon(\sigma(@(x, @(x, y)))) = 7$ but $\Upsilon(\sigma(@(x, @(y, y)))) = 6$.

On the other hand, note that for arbitrary closed terms $s, t \in \mathcal{T}_0$ we have

$$\Upsilon(\mathtt{let}(x.\mathtt{let}(y.@(x, @(x, y)), t), s)) = \Upsilon(\mathtt{let}(x.\mathtt{let}(y.@(x, @(y, y)), t), s))$$

Thus, the sharing has been broken by the substitution discussed above.

Closing terms by surrounding them with let-expressions is essentially what will be done in chapter 4 for the $\boldsymbol{\lambda}_{\approx}$-calculus. But for now we will not bother with concrete instances for a distinct lcl. Hence for a preorder $\eta \subseteq \mathcal{T}_0{}^2$ on closed terms its *extension η^o to open terms* will be defined rather conceptually. With the notion of *admissibility* we will soon impose additional restrictions:

Definition 2.2.7. *A map $(\cdot)^o : \mathcal{T}_0 \times \mathcal{T}_0 \to \mathcal{T} \times \mathcal{T}$ is said to* extend (relations) to open terms *and η^o is called the* extension of the relation η to open terms.

What follows is the definition of the precongruence candidate, which works without specific properties of $(\cdot)^o$ which extends $\eta \subseteq \mathcal{T}_0{}^2$ to open terms. The precongruence candidate can easily be seen to be compatible and reflexive but not necessarily transitive. Developing the criteria under which it coincides with the relation η^o is the purpose of this section.

---

[1]This is not the number of occurrences but the cardinality of the variables viewed as a set.

Definition 2.2.8 (Precongruence Candidate). *Let $\eta \subseteq \mathcal{T}_0 \times \mathcal{T}_0$ be a pre-order. Then define its* precongruence candidate $\widehat{\eta} \subseteq \mathcal{T} \times \mathcal{T}$ *inductively by*

- $x \, \widehat{\eta} \, b$ *if* $x \in V$ *is a variable and* $x \, \eta^o \, b$.

- $\tau(\overline{a}_i) \, \widehat{\eta} \, b$ *if there exists* $\overline{a}_i'$ *such that* $\overline{a}_i \, \widehat{\eta} \, \overline{a}_i'$ *and* $\tau(\overline{a}_i') \, \eta^o \, b$ *hold.*

The informal account in [How89, p. 201] is striking: *$a \, \widehat{\eta} \, b$ if b can be obtained from a via one bottom-up pass of replacements of subterms by terms that are larger under η^o.* So clearly for every nullary operator ζ, i.e., $\alpha(\zeta) = \langle \rangle$ and ζ has no operands, we obtain $\zeta \, \widehat{\eta} \, t \iff \zeta \, \eta^o \, t$ for $t \in \mathcal{T}$ be an arbitrary term.

Under certain conditions, it can be shown that no additional free variables have to be introduced for the intermediate operators $\overline{a}_i'$ in the recursive case of definition 2.2.8 above. Because of its interaction with the open extension $(\cdot)^o$ we cannot provide such a result in general here. But it is feasible in the context of the $\boldsymbol{\lambda}_{\approx}$-calculus, cf. lemma 4.2.3, where a term exists that may substituted for such a "spare" variable, i.e. a free variable in $\overline{a}_i'$ but not in $\overline{a}_i$ before.

Definition 2.2.9. *If $(\cdot)^o$ extends relations to open terms, it is called* preorder-preserving *if and only if for every relation $\eta \subseteq \mathcal{T}_0{}^2$ on closed terms the following holds: η is a preorder implies that η^o is a preorder, too.*

The notion above is sufficient for some of the most fundamental properties, like the following counterpart to [How96, Lemma 3.1].

Lemma 2.2.10. *Let $\eta \subseteq \mathcal{T}_0{}^2$ be a preorder and $(\cdot)^o$ preorder-preserving. Then the relation $\eta^o \subseteq \mathcal{T}^2$ is a preorder, too, and furthermore*

1. *$\widehat{\eta}$ is reflexive*

2. *$\widehat{\eta}$ and $\widehat{\eta}_0$ are operator-respecting*

3. *$\eta^o \subseteq \widehat{\eta}$*

4. *$\widehat{\eta} \circ \eta^o \subseteq \widehat{\eta}$*

Proof.   1. Let $a \in \mathcal{T}$ be an arbitrary but fixed term, then we show $a \, \widehat{\eta} \, a$ by induction on the structure of a:

- If $a \in V$ is a variable, we have $a \, \widehat{\eta} \, a$ from the reflexivity of η^o and the base case of definition 2.2.8.

- If $a \equiv \tau(\overline{a}_i)$ for some operator τ and operands a_i, we have $a_i \, \widehat{\eta} \, a_i$ from the induction hypothesis and $\tau(\overline{a}_i) \, \eta^o \, \tau(\overline{a}_i)$ from the reflexivity of η^o. So, by definition 2.2.8, we may compose this to $\tau(\overline{a}_i) \, \widehat{\eta} \, \tau(\overline{a}_i)$.

2. We assume $\overline{a}_i \; \widehat{\eta} \; \overline{b}_i$ and have to show $\tau(\overline{a}_i) \; \widehat{\eta} \; \tau(\overline{b}_i)$ for an arbitrary but fixed operator τ. By reflexivity of η^o we have $\tau(\overline{b}_i) \; \eta^o \; \tau(\overline{b}_i)$ and from definition 2.2.8 we conclude $\tau(\overline{a}_i) \; \widehat{\eta} \; \tau(\overline{b}_i)$.

   As a consequence if $\tau(\overline{a}_i), \tau(\overline{b}_i) \in \mathcal{T}_0$ are closed, we also have $\tau(\overline{a}_i) \; \widehat{\eta}_0 \; \tau(\overline{b}_i)$ from $\overline{a}_i \; \widehat{\eta} \; \overline{b}_i$, which implies that $\widehat{\eta}_0$ is operator-respecting too.

3. Assume $a \; \eta^o \; b$ for arbitrary but fixed $a, b \in \mathcal{T}$ and show $a \; \widehat{\eta} \; b$ by induction on the structure of a:

   - If $a \in V$ is a variable, we have $a \; \widehat{\eta} \; b$ directly from $a \; \eta^o \; b$ and the base case of definition 2.2.8.

   - If $a \equiv \tau(\overline{a}_i)$ for some operator τ and operands a_i, we have $a_i \; \widehat{\eta} \; a_i$ from property 1, the reflexivity of $\widehat{\eta}$. By definition 2.2.8 then, we may conclude $\tau(\overline{a}_i) \; \widehat{\eta} \; b$.

4. Assume $a \; \widehat{\eta} \; b$ and $b \; \eta^o \; c$, so according to definition 2.2.8 we have to distinguish the following two cases:

   - a is a variable, then for $a \; \widehat{\eta} \; b$ also $a \; \eta^o \; b$ must hold and thus the proposition by transitivity of η^o and property 3.

   - For a of the form $\tau(\overline{a}_i)$ there is $\tau(\overline{a}_i')$ with $\overline{a}_i \; \widehat{\eta} \; \overline{a}_i'$ and $\tau(\overline{a}_i') \; \eta^o \; b$. By transitivity of η^o we also have $\tau(\overline{a}_i') \; \eta^o \; c$ thus $\tau(\overline{a}_i) \; \widehat{\eta} \; c$. $\qquad\square$

Theorem 2.2.13 at the end of this section is vital in establishing the criteria under which η^o becomes a precongruence. In order to prove this theorem, we have to demand the following properties.

Definition 2.2.11. *Let $\eta \subseteq \mathcal{T}_0{}^2$ be a relation, then an extension $\eta^o \subseteq \mathcal{T}^2$ of η to open terms is* admissible *if only if all of the following conditions are met:*

1. *$(\cdot)^o$ is preorder-preserving*

2. *$(\eta^o)_0 = \eta$*

3. *$\forall \nu : \; \nu \subseteq \eta \implies \nu^o \subseteq \eta^o$*

4. *$\widehat{\eta} \; \subseteq (\widehat{\eta}_0)^o$*

Note that, because of the interdependency of η^o and $\widehat{\eta}$ above, the order of the definitions 2.2.7 and 2.2.8 is indeed significant.

Lemma 2.2.12. *Let $\eta \subseteq \mathcal{T}_0{}^2$ be a preorder on closed terms. Then every admissible extension η^o contains η, i.e. $\eta \subseteq \eta^o$.*

Proof. Assume a preorder $\eta \subseteq \mathcal{T}_0{}^2$ on closed terms such that η^o is admissible. Then $(\eta^o)_0 = \eta$ follows from property 2 of definition 2.2.11. Furthermore, for every relation $\nu \subseteq \mathcal{T}^2$ on terms, the inclusion $\nu_0 = \nu \cap \mathcal{T}_0{}^2 \subseteq \nu$ holds. Thus, with $\eta = (\eta^o)_0 = \eta^o \cap \mathcal{T}_0{}^2 \subseteq \eta^o$ the proposition is shown. □

In [How96, Theorem 3.1] the analogical formulation of the following theorem requires a substitution lemma which we cannot provide at this point. This is so, because until now only a few assumptions about η^o have been made.

On the other hand, the preceding definitions, particularly the notion of admissibility from definition 2.2.11, represent just the abstract conditions which are actually relevant. This is justified by the fact that they are strong enough for the proof of this section's main result.

Theorem 2.2.13. *Let $\eta \subseteq \mathcal{T}_0{}^2$ be a preorder and η^o its admissible extension to open terms. Then the following are equivalent.*

1. *η^o is a precongruence*

2. *$\widehat{\eta} \subseteq \eta^o$*

3. *$\widehat{\eta}_0 \subseteq \eta$*

Proof. The claim is shown by a chain of implications.

"1 $\implies$ 2": Assuming η^o to be a precongruence and $a \widehat{\eta} b$, we show $a \eta^o b$ by induction on the definition of $\widehat{\eta}$.

- If $a \in V$ is a variable, the only possibility is $a \eta^o b$.
- If $a \equiv \tau(\overline{a}_i)$ for some operator τ and operands a_i, there must have been operands a_i' such that $a_i \widehat{\eta} a_i'$ for every i and $\tau(\overline{a}_i') \eta^o b$. From the induction hypothesis we may conclude $\overline{a}_i \eta^o \overline{a}_i'$, which in turn means $\tau(\overline{a}_i) \eta^o \tau(\overline{a}_i')$ and furthermore $\tau(\overline{a}_i) \eta^o b$ since η^o is a precongruence.

"2 $\implies$ 3": From $\widehat{\eta} \subseteq \eta^o$ we have $\widehat{\eta}_0 \subseteq (\eta^o)_0 = \eta$ since η^o is admissible.

"3 $\implies$ 1": So it remains to show that η^o is a precongruence under the assumption $\widehat{\eta}_0 \subseteq \eta$. Since η^o is admissible, from $\widehat{\eta}_0 \subseteq \eta$ we have $(\widehat{\eta}_0)^o \subseteq \eta^o$ by monotonicity, i.e. property 3 of definition 2.2.11. In conjunction with property 4 there, this becomes $\widehat{\eta} \subseteq (\widehat{\eta}_0)^o \subseteq \eta^o$. Hence by property 3 of lemma 2.2.10 we have $\widehat{\eta} = \eta^o$, thus η^o is operator-respecting. □

The plan is to establish the last set inclusion $\widehat{\eta}_0 \subseteq \eta$ for which we will show the precongruence candidate $\widehat{\eta}$ to be stable under reduction. Thus, we now turn our attention to reduction and evaluation in lazy computation languages.

2.3   Reduction and Evaluation

In this section we shall investigate the operational semantics of a lazy computation language. In operational semantics, cf. [Hen90, Win93], it is common to distinguish between "big-step" and "small-step" operational semantics. Roughly, a big-step operational semantics is mainly concerned with the result of an *evaluation* whereas a small-step semantics emphasises the way which leads to this particular outcome. Therefore, big-step operational semantics seems somewhat related to denotational semantics which is covered in e.g. [Sch86, Mos90]. The small-step semantics for lambda calculi is *reduction*, which is a rewrite system on terms, cf. [Klo92, BN98, DP01, KBdV03], by its nature.

This section will depict reduction and evaluation to the extent applicable for the abstract notion of a lazy computation language. We begin with big-step operational semantics.

Definition 2.3.1 (Lazy Computation System). *A* lazy computation language $\mathcal{L}$ *together with a binary relation* $\Downarrow \subseteq \mathcal{T}^2$ *on terms is called a* lazy computation system *(lcs for short) if and only if* $\Downarrow$ *meets the following:*

$$a \Downarrow v \implies \forall v' : (v \Downarrow v' \iff v \equiv v') \tag{2.3.1}$$

We then call $\Downarrow$ evaluation *or* convergence. *The term* v *is the* answer *to which* a *is said to* converge *to. We simply write* $a \Downarrow$ *if there exists some* v *such that* a *converges to* v, *and* $a \not\Downarrow$ *if there is no such* v.

Note that (2.3.1) does not enforce determinism, as $a \Downarrow v$ may hold for different v. However, this condition causes the answers to be "end-points" of the evaluation in some sense. Frequently, small-step reduction is coupled to big-step evaluation when these end-points can be reached by finitely many reduction steps. In [How89] this is done by partitioning the set O of language operators into *canonical* and *non-canonical* ones. Terms whose top-level operator is canonical are then called *canonical terms* and regarded as answers. To indicate this, the operator θ instead of τ will be used sometimes.

Definition 2.3.2. *The* answer set *of a term* s *is defined by* $\mathtt{ans}(s) = \{\, t \mid s \Downarrow t \,\}$.

In [Bar84, p. 50], Barendregt requires a *reduction relation* to be compatible with contexts in general. Such a constraint will not be adopted, since it would be too restrictive for non-deterministic calculi. Another reason is that, as exposed in section 1.1 before, contrary to [Bar84, AFM+95], this work rests upon contextual equivalence instead of convertibility.

Definition 2.3.3 (Reduction). *A reduction relation is a binary relation on terms which may be specified by certain reduction rules.*

If $s, t \in \mathcal{T}$ are terms so that s reduces to t by some rule (a) we write $s \xrightarrow{a} t$ and speak of a top-level reduction. If $D \in \mathcal{C}$ is a context and s reduces to t at top-level by rule (a) then we may use rule (a) in the context D, which is denoted by $D[s] \xrightarrow{D,\,a} D[t]$.

The term s is referred to as an a-redex because it is a "reducible expression" and rule (a) may be applied to it. Furthermore, the term t is called the reduct *and $D[t]$ the* contractum. *If $\mathcal{D} \subseteq \mathcal{C}$ stands for some subset of contexts we will write $s \xrightarrow{\mathcal{D},\,a} t$ if and only if $s \xrightarrow{D,\,a} t$ holds for some $D \in \mathcal{D}$. When (a) is not explicitly specified it may be any of the rules from the respective calculus.*

We will not go into the details of how single rewrite steps may be defined by means of reduction rules, since this is outside the scope of this work. We rather postulate the rules to be specified in a framework like [KvOvR93, San97].

Writing $s \xrightarrow{[\,],\,a} t$ we shall emphasise a top-level reduction. As is customary, the transitive and reflexive-transitive closure of $\rightarrow$ will be denoted by the symbols $\rightarrow^{+}$ and $\rightarrow^{*}$ respectively. Furthermore, we write $\rightarrow^{k}$ and $\rightarrow^{<k}$ for a reduction of exactly and less than k steps, whereas e.g. $\rightarrow^{k \geq m}$ indicates a reduction of $k \geq m$ steps. Further labels may be attached to the symbol $\rightarrow$ in order to specify certain properties of the reduction. A word which comprises all the labels of an arrow will usually be designated by lowercase Greek letters.

Reduction rules might also be used in the opposite direction. Instead of a *contraction* we then speak of an *expansion*. A *conversion* is either an contraction or an expansion.

Definition 2.3.4 (Reduction and Conversion Sequences). *Let $n \in \mathbb{N}$ be a non-negative integer, $t_1, \ldots, t_{n+1} \in \mathcal{T}$ be terms, $C_{a_1}, \ldots, C_{a_n}$ be contexts and $a_1, \ldots, a_n$ be reduction rules (with possibly further labels attached).*

Then a sequence $(\langle t_i, a_i, C_{a_i} \rangle)_i$ is a reduction sequence (of length n) *if and only if $t_i \xrightarrow{C_{a_i},\,a_i} t_{i+1}$ for every $1 \leq i \leq n$ holds.*

If for every $1 \leq i \leq n$, we have either $t_i \xrightarrow{C_{a_i},\,a_i} t_{i+1}$ or $t_i \xleftarrow{C_{a_i},\,a_i} t_{i+1}$, then we call $(\langle t_i, a_i, C_{a_i} \rangle)_i$ a conversion sequence (of length n).

2.3.1 Convertibility and Confluence

Quite naturally, conversion gives rise to the following equivalence relation, which is the reflexive-symmetric-transitive closure of $\rightarrow$ in fact.

Definition 2.3.5 (Convertibility). *Two terms $r, s \in \mathcal{T}$ are* convertible *if and only if there is a conversion sequence $(\langle t_i, a_i, C_{a_i} \rangle)_i$ such that $r \equiv t_1$ and $t_{n+1} \equiv s$ holds. We then write $r \overset{*}{\leftrightarrow} t$ while the relation $\overset{*}{\leftrightarrow}$ is called* convertibility.

Note that in *equational reasoning*, e.g. [HO80, Pla93], it is common to define conversion first and then derive reduction rules by giving an orientation to the equations. But equational reasoning is based on the basic principle that "equals may be replaced by equals" and thus requires the equivalence to be *substitutive*, i.e. compatible. This is why conversion is usually permitted inside arbitrary contexts, which makes convertibility a congruence.

However, in this way, equality and reduction are heavily intertwined, since the reduction rules, viewed as *program transformations*, are trivially correct w.r.t. convertibility. Hence the notion of *confluence* frequently serves as a separate justification, e.g. [Bar84, MOW98], that a certain calculus behaves well.

Definition 2.3.6 (Confluence). *A reduction relation $\rightarrow \subseteq \mathcal{T}^2$ is* confluent *if for all terms $p, q, r \in \mathcal{T}$ satisfying $p \overset{*}{\leftarrow} q \rightarrow^* r$ there is a further term $t \in \mathcal{T}$ such that $p \rightarrow^* t \overset{*}{\leftarrow} r$ holds.*

Figure 2.1 depicts confluence and its weaker companion local confluence. Confluence and techniques to prove it have been addressed at length, cf. [Hue80].

(a)   (b)
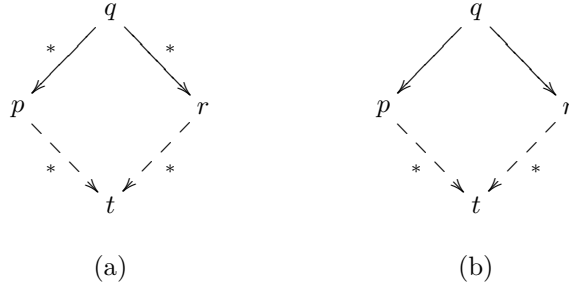
Figure 2.1: Confluence (a) and local confluence (b)

One particular result worth mentioning is *Newman's Lemma*, i.e. that every terminating rewrite system is confluent whenever it is locally confluent. As a consequence, for the confluence of a terminating rewrite system it is sufficient to show that all its *critical pairs* are joinable. This notion has successfully been transferred to the higher-order case in [MN98]. Roughly, a critical pair represents the overlap of two reductions which leads to a non-trivial *peak*, i.e. a

situation where $p \leftarrow q \rightarrow r$ and the existence of a term t with $p \rightarrow^* t \; {}^* \leftarrow r$ is not obvious. We will see later how this motivates our notions of complete sets of forking and commuting diagrams.

Note that local confluence in general does *not* imply confluence as the following ubiquitous example, cf. e.g. [BN98, p. 29], illustrates.

Example 2.3.7. *Let the reduction relation $\rightarrow$ be specified by*

$$a \longleftarrow b \; \substack{\displaystyle\curvearrowright \\ \displaystyle\curvearrowleft} \; c \longrightarrow d$$

Then clearly $\rightarrow$ is locally confluent but not confluent.

2.3.2 Contextual Equivalence

In operational semantics, contextual equivalence, cf. [Mor68, Plo75], is widely used and can be considered *the* standard semantic equivalence. Section 1.1.3 gives several reasons why we consider it more significant than convertibility.

Contextual equivalence equates two terms if they exhibit the same converging behaviour in all possible contexts.

Definition 2.3.8. *The* contextual equivalence $\simeq_c \subseteq \mathcal{T} \times \mathcal{T}$ *is defined by*

$$s \simeq_c t \; \stackrel{def}{\iff} \; (\forall C \in \mathcal{C} : \; C[s] \Downarrow \iff C[t] \Downarrow) \tag{2.3.2}$$

Contrary to convertibility, it forms a congruence independently of where reduction is permitted. Therefore, contextual equivalence provides a reasonable alternative to confluence for the validation of a calculus.

However, proving that two terms are contextually equivalent requires to take infinitely many contexts into account. Switching to another equality which implies contextual equivalence may circumvent this. As mentioned before, this is one of the main reasons for the treatment of similarity and its precongruence proof in this work. But for a while, we will raise the issue of how to directly establish the reduction rules to be correct program transformations w.r.t. contextual equivalence.

Definition 2.3.9 (Program Transformation). *A program transformation is a binary relation on terms. Let $\Psi \subseteq \mathcal{T}^2$ be a program transformation. Then Ψ is* correct *(w.r.t. contextual equivalence) if and only if for all terms $s, t \in \mathcal{T}$ the implication $s \; \Psi \; t \implies s \simeq_c t$ holds.*

We do not require a program transformation to be compatible with contexts, though program transformations shall typically be applied at arbitrary locations of a program. However, for every correct program transformation its compatible closure is again a correct program transformation: For $s\ \Psi\ t$, correctness of Ψ implies $s \simeq_c t$ and furthermore $C[s] \simeq_c C[t]$ since $\simeq_c$ is a congruence.

Now suppose that evaluation is defined by finite $\xrightarrow{\alpha}$-reduction sequences to an answer. Then it is in general not obvious, whether the reduction rules itself constitute correct program transformations. But we shall adopt techniques which are, to some extent, related to the ones in confluence proofs.
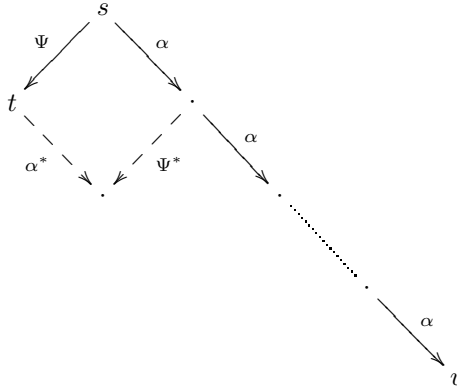
Figure 2.2: Application of a Program Transformation Ψ

First note, that non-terminating reduction sequences are ruled out in the definition of contextual equivalence. Therefore, induction becomes applicable like for the proof of Newman's Lemma. Now regard the illustration in figure 2.2 where $\xrightarrow{\Psi}$ indicates an application of the program transformation Ψ to a term while the $\xrightarrow{\alpha}$-reduction sequence to the right is assumed to end in an answer. If the dashed relations exist, α and Ψ are called to *commute weakly* in [BKvO98].

As can be seen, for t a finite sequence of $\xrightarrow{\alpha}$-reductions could, under certain conditions, be constructed from an argument of induction. This sequence ends in a term which may be reached from v by several program transformations. If the notion of answer is invariant w.r.t. the transformation Ψ, the $\xrightarrow{\alpha}$-reduction sequence for t might produce an answer too, i.e., t converges.

In this manner the implication $C[s] \Downarrow \implies C[t] \Downarrow$ of the contextual equiv-

alence $s \simeq_c t$ could be established. Of course, this is just one example from numerous possibilities, but it demonstrates well the main idea behind the notions of the next section. Also does the aforementioned article [BKvO98] discuss several confluence patterns in addition to the one in figure 2.2.

2.3.3 Reduction Diagrams

The method presented in this section will be used in significant parts of later proofs. After it was successfully applied in showing program transformations correct w.r.t. contextual equivalence repeatedly [KSS98, SSSS04], we consider it a well-established technique.

In this work we will employ it as a general framework for transformations on reduction sequences in multiple ways. Instead of diagrams like the one in figure 2.2, a more concise notation will be introduced.

Definition 2.3.10 (Transformation Rule). *A* transformation *of a conversion sequence $(\langle s_i, a_i, C_{a_i} \rangle)_i$ of length m consists of a conversion sequence $(\langle t_j, b_j, C_{b_j} \rangle)_j$ of length n such that $s_1 \equiv t_1$ and $s_{m+1} \equiv t_{n+1}$ holds.*

A transformation rule *describes a set of possible transformations of conversion sequences in an abstract manner using a notation like*

$$\xrightarrow{C_{a_1},\, a_1} \cdot \xleftarrow{C_{a_2},\, a_2} \cdot \ldots \cdot \xleftarrow{C_{a_{m-1}},\, a_{m-1}} \cdot \xrightarrow{C_{a_m},\, a_m} \rightsquigarrow$$

$$\xleftarrow{C_{b_1},\, b_1} \cdot \xrightarrow{C_{b_2},\, b_2} \cdot \ldots \cdot \xrightarrow{C_{b_{n-1}},\, b_{n-1}} \cdot \xleftarrow{C_{b_n},\, b_n}$$

A transformation rule of the above form is called applicable to a prefix (suffix) *of a conversion sequence $(\langle s_i, a_i, C_{a_i} \rangle)_i$ of length $k \geq m$ if there are terms $t_1, \ldots, t_{n+1} \in \mathcal{T}$ such that $(\langle t_j, b_j, C_{b_j} \rangle)_j$ is a transformation of the prefix-sequence $(\langle s_i, a_i, C_{a_i} \rangle)_{i \leq m}$ (suffix-sequence $(\langle s_i, a_i, C_{a_i} \rangle)_{i > k-m}$ respectively).*

Now suppose that, similar to the case in figure 2.2, we have a terminating reduction sequence for some term. Moreover, assume that for every reduction which overlaps with the first of the given sequence, there is a possible transformation to join the peak. Then we obtain the following notion.

Definition 2.3.11 (Complete Set of Forking Diagrams). *A* set of forking diagrams *for a reduction $\xrightarrow{\mathcal{D},\, a}$ w.r.t. two sets B, O of reductions, is a set of*

transformation rules of the form

$$\xleftarrow{C_{b_1},\ b_1} \cdot \ldots \cdot \xleftarrow{C_{b_l},\ b_l} \cdot \xrightarrow{\mathcal{D},\ a} \rightsquigarrow$$

$$\xrightarrow{C_{a_1},\ a_1} \cdot \ldots \cdot \xrightarrow{C_{a_n},\ a_n} \cdot \xleftarrow{C_{b'_1},\ b'_1} \cdot \ldots \cdot \xleftarrow{C_{b'_m},\ b'_m}$$

for conversion sequences, where the following conditions are met:

1. $\xrightarrow{C_{b_i},\ b_i} \subseteq B$ *for every* $1 \le i \le l$,

2. $\xrightarrow{C_{b'_i},\ b'_i} \subseteq B$ *for every* $1 \le i \le m$ *and*

3. $\xrightarrow{C_{a_j},\ a_j} \subseteq O \cup \xrightarrow{\mathcal{D},\ a}$ *for every* $1 \le j \le n$

A set of forking diagrams is called complete *if and only if for every conversion sequence of the form*

$$s_0 \xleftarrow{C_{b_1},\ b_1} \ldots \xleftarrow{C_{b_{l-1}},\ b_{l-1}} s_{l-1} \xleftarrow{C_{b_l},\ b_l} s_l \xrightarrow{\mathcal{D},\ a} s_{l+1}$$

such that $l > 0$ *and* s_0 *is an answer but* s_l *is not, there is one transformation rule applicable to a suffix of the sequence.*

The existence of complete sets of forking diagrams for some reduction may guarantee that arbitrary applications of this reduction do not compromise the termination behaviour. This is how complete sets of forking diagrams will typically be used in this work, although they are equally well suited for the correctness of program transformations.

However, as the preceding section has shown, from complete set of forking diagrams only one of the implications necessary for contextual equivalence could be established. So assume $s\ \Psi\ t$ where t rather than s has a finite reduction to an answer. If the inverse program transformation Ψ^{-1} of Ψ is considered, a complete set of forking diagrams for $\xrightarrow{\Psi^{-1}}$ will achieve the desired effect. Hence the corresponding concept for the original transformation Ψ is that of a complete set of commuting diagrams.

Definition 2.3.12 (Complete Set of Commuting Diagrams). *A* set of commuting diagrams *for a reduction* $\xrightarrow{\mathcal{D},\ a}$ *w.r.t. two sets* B, O *of reductions, is*

a set of transformation rules of the form

$$\xrightarrow{\mathcal{D},\ a} \cdot \xrightarrow{C_{b_1},\ b_1} \cdot \ldots \cdot \xrightarrow{C_{b_l},\ b_l} \leadsto$$

$$\xrightarrow{C'_{b_1},\ b'_1} \cdot \ldots \cdot \xrightarrow{C_{b'_m},\ b'_m} \cdot \xrightarrow{C_{a_1},\ a_1} \cdot \ldots \cdot \xrightarrow{C_{a_n},\ a_n}$$

for reduction sequences, where the following conditions are met:

1. $\xrightarrow{C_{b_i},\ b_i} \subseteq B$ *for every* $1 \leq i \leq l$,

2. $\xrightarrow{C_{b'_i},\ b'_i} \subseteq B$ *for every* $1 \leq i \leq m$ *and*

3. $\xrightarrow{C_{a_j},\ a_j} \subseteq O \cup \xrightarrow{\mathcal{D},\ a}$ *for every* $1 \leq j \leq n$

A set of commuting diagrams is called complete *if and only if for every reduction sequence of the form*

$$s_0 \xrightarrow{\mathcal{D},\ a} s_1 \xrightarrow{C_{b_1},\ b_1} \ldots \xrightarrow{C_{b_l},\ b_l} s_{l+1}$$

such that $l > 0$ *and* s_{l+1} *is an answer but* s_0 *is not, there is one transformation rule applicable to a prefix of the sequence.*

By means of a complete set of commuting diagrams for $\xrightarrow{\mathcal{D},\ a}$ it can be shown that every $\xrightarrow{\mathcal{D},\ a}$-reduction may be moved to the end of any terminating reduction sequence consisting only of reductions from B. During this process auxiliary reductions from O might possibly be introduced.

In this work, complete sets of commuting diagrams will chiefly be of the form $\xrightarrow{\mathcal{D},\ a} \cdot \xrightarrow{\mathcal{D},\ b} \leadsto \xrightarrow{\mathcal{D},\ b}{}^{*} \cdot \xrightarrow{\mathcal{D},\ a}$ whereas in e.g. [SSSS04] also patterns like $\xrightarrow{\mathcal{D},\ a} \cdot \xrightarrow{\mathcal{D},\ b}{}^{*} \leadsto \xrightarrow{\mathcal{D},\ b}{}^{*} \cdot \xrightarrow{\mathcal{D},\ a}$ occur sometime.

Both, complete sets of commuting and forking diagrams, should adhere to the condition that the composition of diagrams from the set terminates. Apart from their limitation to terminating reduction sequences, this is another prerequisite to make induction applicable.

If $O = \emptyset$ in the definitions above, it is omitted and we simply speak of a set of forking (commuting) diagrams w.r.t. B.

Independent Reductions

While diagrams are a very useful tool to prove commuting reductions, there
are already some simple cases which may be shown in a generic manner. We
therefore introduce the notion of *disjoint* contexts.

Definition 2.3.13. *Two contexts $C_a, C_b \in \mathcal{C}$ are called* disjoint *if there is no
other context $C \in \mathcal{C}$ such that $C_a \equiv C_b[C]$ or $C_b \equiv C_a[C]$ holds.*

Lemma 2.3.14. *Let $C_a, C_b \in \mathcal{C}$ be two disjoint contexts. Then for all terms
$s_0, s_1, s_2 \in \mathcal{T}$ and all reductions a, b the following holds:*

$$s_0 \xrightarrow{C_a,\, a} s_1 \xrightarrow{C_b,\, b} s_2 \implies \exists s_2', C_a', C_b' : \ s_0 \xrightarrow{C_b',\, b} s_2' \xrightarrow{C_a',\, a} s_2$$

$$s_1 \xleftarrow{C_a,\, a} s_0 \xrightarrow{C_b,\, b} s_2 \implies \exists s_3, C_a', C_b' : \ s_1 \xrightarrow{C_b',\, b} s_3 \xleftarrow{C_a',\, a} s_2$$

Proof. Assume $C_a, C_b \in \mathcal{C}$ to be disjoint contexts, i.e. neither $C_a \equiv C_b[C]$ nor
$C_b \equiv C_a[C]$ for some further context $C \in \mathcal{C}$ holds. We recall that $s_0 \xrightarrow{C_a,\, a} s_1$
means that s_0 and s_1 are of the respective forms $s_0 \equiv C_a[t_1]$ and $s_1 \equiv C_a[t_1']$
such that $t_1 \xrightarrow{[\,],\, a} t_1'$ is a top-level reduction. The same holds for C_b, i.e. in the
forking case which we will treat first, we have $s_0 \equiv C_b[t_2]$ and $s_2 \equiv C_b[t_2']$ such
that $t_2 \xrightarrow{[\,],\, b} t_2'$ is a top-level reduction. Since C_a and C_b are disjoint, there is a
two-hole-context $C[[\ ]_1, [\ ]_2] \in \mathcal{C}$ such that $C_a \equiv C[[\ ]_1, t_2]$ and $C_b \equiv C[t_1, [\ ]_2]$.
Thus $C_a' \equiv C[[\ ]_1, t_2']$ and $C_b' \equiv C[t_1', [\ ]_2]$ represent the desired contexts, which
clearly permit the reductions $s_1 \xrightarrow{C_a',\, a} s_3$ and $s_2 \xrightarrow{C_b',\, b} s_3$.

For the commuting case we have $s_1 \equiv C_b[t_2]$ and $s_2 \equiv C_b[t_2']$ such that the
reduction $t_2 \xrightarrow{[\,],\, b} t_2'$ is at top-level. Since C_a and C_b are disjoint, there is a
two-hole-context $C[[\ ]_1, [\ ]_2] \in \mathcal{C}$ such that $C_a \equiv C[[\ ]_1, t_2]$ and $C_b \equiv C[t_1', [\ ]]$
hold. Thus $C_a' \equiv C[[\ ]_1, t_2']$ and $C_b' \equiv C[t_1, [\ ]_2]$ are contexts, which permit the
reductions $s_0 \equiv C[t_1, t_2] \xrightarrow{C_b',\, b} C[t_1, t_2'] \xrightarrow{C_a',\, a} C[t_1', t_2'] \equiv s_2$ as desired.   □

Constructing Complete sets of Diagrams

In the following, we will develop criteria on how to establish complete sets of
commuting and forking diagrams respectively. The key idea is that, for reduc-
tions which may be performed inside contexts that are closed under composition,
it is sufficient to analyse the empty context in a few base cases. The next lemma
demonstrates this for the commutation of two reductions.

Lemma 2.3.15. *Let* $s_0 \xrightarrow{C_a,\ a} s_1 \xrightarrow{C_b,\ b} s_2$ *be a reduction sequence for arbitrary terms* $s_0, s_1, s_2 \in \mathcal{T}$. *If* $C_b \equiv C_a[C]$ *with some context* $C \in \mathcal{C}$ *then the following holds: If for every reduction sequence* $t_0 \xrightarrow{[\ ],\ a} t_1 \xrightarrow{C,\ b} t_2$ *there exists a term* $t_1' \in \mathcal{T}$ *and a context* C' *such that* $t_0 \xrightarrow{C',\ b} t_1' \xrightarrow{[\ ],\ a} t_2$ *then there is also a term* $s_1' \in \mathcal{T}$ *and a context* $C_a' \in \mathcal{C}$ *with* $s_0 \xrightarrow{C_b,\ b} s_1' \xrightarrow{C_a',\ a} s_2$.

Proof. Assume $s_0 \xrightarrow{C_a,\ a} s_1 \xrightarrow{C_b,\ b} s_2$ then s_0 and s_1 must be of the respective forms $s_0 \equiv C_a[t_0]$ and $s_1 \equiv C_a[t_1]$ for some terms $t_0, t_1 \in \mathcal{T}$ and a top-level reduction $t_0 \xrightarrow{[\ ],\ a} t_1$. Since $C_b \equiv C_a[C]$ we have $s_2 \equiv C_a[t_2]$ and $t_1 \xrightarrow{C,\ b} t_2$ from $s_1 \xrightarrow{C_b,\ b} s_2$, hence there are t_1' and C' such that $t_0 \xrightarrow{C',\ b} t_1' \xrightarrow{[\ ],\ a} t_2$ by the premise. This reduction can also be performed inside C_a, thus the claim. $\square$

The last step of the proof using composition of contexts is remarkable. Hence we may adopt the technique to reductions inside a class of contexts which is closed under composition. Also the claim could be extended to reduction and conversion sequences of a length greater than two by similar arguments, hence the summary in the following corollary.

Corollary 2.3.16. *Let* $(\langle s_i, a_i, C_{a_i} \rangle)_i$ *be a reduction (conversion) sequence of length* m *and* $1 \le k \le m$ *an index such that for every* $1 \le i \le m$ *there are contexts* C_{a_i}' *with* $C_{a_i} \equiv C_{a_k}[C_{a_i}']$ *for* $i \ne k$ *and* $C_{a_k}' \equiv [\ ]$. *Furthermore, presume that for every conversion sequence* $(\langle p_i, a_i, C_{a_i}' \rangle)_i$ *there is a transformation* $(\langle q_j, b_j, C_{b_j} \rangle)_j$. *Then there exists also a transformation* $(\langle t_j, b_j, C_k[C_{b_j}] \rangle)_j$ *of the original reduction (conversion) sequence.*

It is important to note that one serious restriction applies, when making use of lemma 2.3.15 and the above corollary respectively. I.e., in this case it is not valid to consider converging reduction sequences only.

Usually this is permissible since a set of reduction diagrams is complete if it contains one applicable diagram for every converging reduction. But when some part of the enclosing context is disregarded, also non-converging terms have to be taken into account, for obvious reasons: Even if some term t itself does not converge, for some compound term $C[t]$ this will clearly be possible.

2.4   Simulations

The preceding section addressed evaluation and reduction in lazy computation languages thus defining lazy computation systems. Moreover, two forms of

equality, namely convertibility and contextual equivalence, have been discussed there. We have seen how contextual equivalence provides a meaningful approach for the correctness of program transformations.

This section will now develop a further view on equivalence which yields towards a more stepwise procedure to discover the behaviour of terms compared to contextual equivalence. It is based on an, not necessarily effective, *experiment*.

Definition 2.4.1. *Let $\mathcal{L}$ be a lazy computation system and $\eta \subseteq \mathcal{T} \times \mathcal{T}$ be a preorder. Then the* experiment $[\cdot]$ *with* $[\eta] \subseteq \mathcal{T_0}^2$ *is given by*

$$s \ [\eta] \ t \ \overset{def}{\iff} \ (\forall \theta(\overline{s}_i): \ s \Downarrow \theta(\overline{s}_i) \implies (\exists \theta(\overline{t}_i): \ t \Downarrow \theta(\overline{t}_i) \ \wedge \ \overline{s}_i \ \eta \ \overline{t}_i)) \quad (2.4.1)$$

The rationale behind an experiment is to determine whether t converges at least as often as s to a term with the same top-level operator. And if it does so, to leave the examination of the respective operands to the specified preorder. Hence the application of an experiment could simply stop at this depth, i.e. if for η the whole relation $\eta = (\mathcal{T_0} \times \mathcal{T_0})^o$ is chosen. But it could also be performed one step further by taking $\eta = [(\mathcal{T_0} \times \mathcal{T_0})^o]^o$ and so on.

In this manner, two terms will be identified as long as no difference in their *behaviour* is encountered while this process is continued arbitrarily deep. In this way, the infinite intersection $\bigcap_i [(\mathcal{T_0} \times \mathcal{T_0})^o]^i$ is computed. It represents the greatest fixed point of the compound $[\cdot^o]$-operator, since $[\cdot^o]$ is monotone on relations considered as sets and sets ordered by inclusion form a complete lattice, cf. [DP92].

Definition 2.4.2 (Simulation). *A preorder $\eta \subseteq \mathcal{T_0}^2$ is called a* simulation *if and only if $\eta \subseteq [\eta^o]$ holds. Similarity $\lesssim_b$ is defined to be the largest simulation.*

To see that $[\cdot^o]$ is in fact monotone, consider two preorders $\nu, \eta \subseteq \mathcal{T_0}^2$ on closed terms such that $\nu \subseteq \eta$ holds. Since $(\cdot)^o$ is admissible $\nu^o \subseteq \eta^o$ follows from property 3 of definition 2.2.11 immediately. Assuming $s[\nu^o]t$ we have $s[\eta^o]t$ too. Thus $[\cdot^o]$ is monotone and its greatest fixed point actually exists.

By $\eta \subseteq [\eta^o]$ a simulation constitutes a *post-fixed point* of the $[\cdot^o]$-operator. A relation η which meets this property is called $[\cdot^o]$-*dense* in [Gor94b]. There, a quite perspicuous account[2] is given that similarity coincides with both the greatest fixed point of $[\cdot^o]$ and the union $\bigcup \{\, \eta \mid \eta \subseteq [\eta^o] \,\}$ of all $[\cdot^o]$-dense sets. The latter gives rise to the proof principle of *co-induction*, cf. [Gor99, NNH99]. In order to establish $s \lesssim_b t$ it suffices to give any simulation η such that $(s, t) \in \eta$

---

[2]in a more general setting concerning monotone operators on sets

is satisfied. Then $(s, t) \in \lesssim_b$ holds, too, because $\lesssim_b$ is the largest simulation which contains all $[\cdot^o]$-dense sets.

This technique is due to the work of Milner [Mil71] and Park [Par81] who had applied it to transition systems originally. Meanwhile it has also established as a standard approach, cf. [Abr90, Las98b, Gor99], for functional programming and lambda calculi. Since two programs are regarded equivalent as long as they show the same behaviour, it is sometimes called *behavioural equivalence* in contrast to the term *observational equivalence* which is frequently used for contextual equivalence. However, based on simulations, behavioural equivalence may be defined in (at least) two ways.

Definition 2.4.3 (Bisimulation). *An equivalence relation η is a* bisimulation *if $\eta \subseteq [\eta^o]$ holds. Then* bisimilarity $\sim_b$ *is the largest bisimulation.*

Note that a simulation is a bisimulation whenever it is symmetric and thus bisimilarity is the largest symmetric simulation. The demand for symmetry before taking the greatest fixed point causes a tight coupling between the terms related by bisimilarity. During every step in applying the experiment, uniform convergent behaviour is required.

An alternative is to regard terms as equivalent if they simulate each other. This means to flip the order of "symmetrisation" and taking the fixed point.

Definition 2.4.4 (Mutual Similarity). *The largest equivalence relation contained in $\lesssim_b$ is defined by $\simeq_b \stackrel{def}{=} \lesssim_b \cap \gtrsim_b$ and called* mutual similarity.

See [Gor99, p. 19] and in particular [Las98b, p. 92] why mutual similarity generally strictly contains bisimilarity in a non-deterministic scenario. Since the aim of this dissertation is to establish contextual equivalence via behavioural equivalence, we opt for mutual similarity. Once similarity has shown to form a precongruence, its inclusion in contextual preorder is relatively straightforward.

Definition 2.4.5 (Contextual Preorder). *The relation $\lesssim_c \subseteq \mathcal{T} \times \mathcal{T}$ is referred to as* contextual preorder *and defined by*

$$s \lesssim_c t \stackrel{def}{\iff} (\forall C \in \mathcal{C} : \ C[s] \Downarrow \implies C[t] \Downarrow) \tag{2.4.2}$$

Obviously, contextual preorder is a precongruence and thus also called *contextual precongruence* frequently. It is clear that $s \simeq_c t$ is valid if and only if $s \lesssim_c t$ and $s \gtrsim_c t$ hold. So mutual similarity implies contextual equivalence whenever similarity implies contextual preorder. The latter holds if similarity extended to open terms is a precongruence and the extension $(\cdot)^o$ *qualifies for preservation of convergence.*

Definition 2.4.6. *An extension* $(\cdot)^o$ *of relations from closed to open terms is said to* qualify for preservation of convergence *if it meets the following condition.*

$$(\forall s, t \in \mathcal{T}_0 :\ s\ \eta\ t \implies (s \Downarrow \implies t \Downarrow)) \implies$$
$$(\forall s', t' \in \mathcal{T} :\ s'\ \eta^o\ t' \implies (s' \Downarrow \implies t' \Downarrow)) \quad (2.4.3)$$

Roughly, an extension $(\cdot)^o$ qualifies for preservation of convergence if it transfers the implication $s \Downarrow \implies t \Downarrow$ from the relation on closed terms to the one over open terms. Note how the premise $s\ \eta\ t \implies (s \Downarrow \implies t \Downarrow)$ in (2.4.3) is satisfied for every simulation: From $s\ \eta\ t$ we have $s\ [\eta^o]\ t$ since η is a simulation, i.e., $\eta \subseteq [\eta^o]$ holds. Then $s \Downarrow \implies t \Downarrow$ is a direct consequence from definition 2.4.1 of the experiment.

Theorem 2.4.7. *Let* $\eta \subseteq \mathcal{T}_0 \times \mathcal{T}_0$ *be a simulation such that* η^o *is a precongruence and* $(\cdot)^o$ *qualifies for preservation of convergence. Then* $\eta^o \subseteq \lesssim_c$ *is true.*

Proof. Assume terms $s, t \in \mathcal{T}$ such that $s\ \eta^o\ t$ holds. Then $\forall C :\ C[s]\ \eta^o\ C[t]$ because η^o is a precongruence. As explained before, η matches the premise of (2.4.3) since it is a simulation. Thus, from $\forall C :\ C[s]\ \eta^o\ C[t]$ we may also infer $\forall C :\ C[s] \Downarrow \implies C[t] \Downarrow$ which establishes the claim. $\qquad \square$

2.4.1   Proving Similarity a Precongruence

The essential challenge is the proof that similarity is a precongruence which has its own right, e.g. for the application of mutual similarity to equational reasoning. However, it is not possible to adopt definition 2.4.1 of the experiment directly for the non-deterministic call-by-need calculus $\boldsymbol{\lambda}_{\mathrm{ND}}$ in section 3.1. The reason is that answers in $\boldsymbol{\lambda}_{\mathrm{ND}}$ might contain unevaluated non-deterministic choices that are explicitly shared using `let`. Thus section 3.2 will introduce the $\boldsymbol{\lambda}_{\approx}$-calculus in which sharing is eliminated from answers by collecting all possible outcomes.

Additionally, in this way the distinction from [How89] between *canonical* and *non-canonical* operators of the language remains intact for the $\boldsymbol{\lambda}_{\approx}$-calculus, i.e. only λ-terms form answers. In contrast, such a distinction is impossible in the calculus $\boldsymbol{\lambda}_{\mathrm{ND}}$, since a `let`-term might either be an answer or constitute a redex, so `let` is neither a canonical nor a non-canonical operator. The underlying difficulty would persist with [How96] even if small-step reduction to canonical terms is abandoned in favour of a big-step evaluation semantics. In definition 2.4.1, this can be seen from the decomposition of the answers.

However, the notion of a simulation derived from Howe's work is general enough to deal with non-deterministic evaluation. As mentioned before, we will apply theorem 2.2.13 in order to prove similarity a precongruence. For this purpose, the inclusion $(\widehat{\lesssim_b})_0 \subseteq \lesssim_b$ is to be shown. Therefore, note that $\lesssim_b$ is defined as the greatest fixed point of the operator $[\cdot^o]$, i.e., contains all $[\cdot^o]$-dense sets. Recall that a relation η is $[\cdot^o]$-dense iff $\eta \subseteq [\eta^o]$ holds. By coinduction, the inclusion $(\widehat{\lesssim_b})_0 \subseteq \left[(\widehat{\lesssim_b})_0{}^o\right]$ suffices. It can be attained by the concept below.

Definition 2.4.8. *Let $\eta \subseteq \mathcal{T}_0{}^2$ be a relation on closed terms. Then* evaluation respects η *if for all closed terms $s, t, \theta(\overline{s}_i) \in \mathcal{T}_0$ such that $s \Downarrow \theta(\overline{s}_i)$ and $s \ \eta \ t$ hold, there exists $\theta(\overline{t}_i)$ such that $t \Downarrow \theta(\overline{t}_i)$ and $\theta(\overline{s}_i) \ [\eta^o] \ \theta(\overline{t}_i)$ are true.*

Figure 2.3 illustrates the situation for a relation η to be respected by evaluation. The immediate consequence is that every such relation is a simulation.
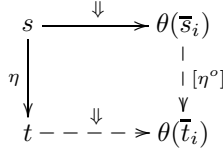
Figure 2.3: Evaluation respects a relation η

Corollary 2.4.9. *Let $\eta \subseteq \mathcal{T}_0{}^2$ be a relation on closed terms that is respected by evaluation. Then η is a simulation, i.e. $\eta \subseteq [\eta^o]$ holds.*

Proof. Assume terms $s, t, \theta(\overline{s}_i)$ such that $s \ \eta \ t$ and $s \Downarrow \theta(\overline{s}_i)$ hold. Since evaluation respects η there exists a term $\theta(\overline{t}_i)$ such that $t \Downarrow \theta(\overline{t}_i)$ and $\theta(\overline{s}_i) \ [\eta^o] \ \theta(\overline{t}_i)$ is satisfied. Since only $\theta(\overline{s}_i) \Downarrow \theta(\overline{s}_i)$ and $\theta(\overline{t}_i) \Downarrow \theta(\overline{t}_i)$ are possible by definition 2.3.1 of the evaluation, this implies $\overline{s}_i \ \eta^o \ \overline{t}_i$ which proves the claim. $\square$

The notion of evaluation *respecting a relation on subterms* in [How96] aims at big-step operational semantics defined in terms of structural rules whereas the *operator extensionality* from [How89] provides a way to show the precongruence candidate to be respected by evaluation on a per-operator basis.

In this work, evaluation will be defined by a finite small-step reduction sequence to answers with a certain syntactical structure. Although this resembles the use of canonical operators, contrary to [How89] the proof will be undertaken for every reduction *rule* instead of every language *operator*. Therefore, we introduce the notion of a relation be stable under reduction.

Definition 2.4.10. *Let $\eta \subseteq \mathcal{T}_0{}^2$ be a relation on closed terms. Then η is said to be* stable under reduction *if for all closed terms $s, s', t, \in \mathcal{T}_0$ such that $s \, \eta \, t$ and $s \to s'$ hold, the relation $s' \, \eta \, t$ is valid as well.*

As convergence for the $\boldsymbol{\lambda}_{\approx}$-calculus is a reduction to an answer, what is to be established by induction is actually the condition (2.4.4) below. In section 4.3 we will be able to prove how it implies that evaluation respects the restriction of the precongruence candidate to closed terms.

$$(s \, \widehat{\eta}_0 \, t \ \wedge \ s \Downarrow \theta(\overline{s}_i)) \implies \theta(\overline{s}_i) \, \widehat{\eta}_0 \, t \tag{2.4.4}$$

Then it will be evident that the precongruence candidate is contained in similarity thus making its open extension a precongruence. This not only enables equational reasoning but also causes mutual similarity to be a correct program transformation w.r.t. contextual equivalence.

2.4.2   Simulation up to

When proving a relation $[\,\cdot^o]$-dense, i.e. that it forms a simulation, sometimes a stronger hypothesis would be helpful. This means to modify the definition of the experiment such that $\overline{s}_i \ (\!|\eta|\!) \ \overline{t}_i$ rather than $\overline{s}_i \, \eta \, \overline{t}_i$ suffices to be shown for some suitable modification $(\!|\eta|\!)$ of the original relation. It is the aim of this section to explore what "suitable" could mean.

One possibility is to consider the operands $\overline{s}_i$ and $\overline{t}_i$ respectively modulo an equivalence relation. This resembles the "bisimulation up to equivalence" of [Mil89, p. 92] and, with $\sim$ denoting some equivalence, looks like

$$s \ [\eta] \ t \iff (\forall \theta(\overline{s}_i) : \ s \Downarrow \theta(\overline{s}_i) \implies (\exists \theta(\overline{t}_i) : \ t \Downarrow \theta(\overline{t}_i) \ \wedge \ \overline{s}_i \, (\sim \eta \sim) \, \overline{t}_i))$$

where $(\!|\eta|\!) = (\sim \eta \sim)$ and $(\nu \, \eta \, \upsilon)$ stands for the composition of relations below.

$$s \ (\nu \, \eta \, \upsilon) \ t \ \overset{\text{def}}{\iff} \ \exists s', t' : \ s \, \nu \, s' \ \wedge \ s' \, \eta \, t' \ \wedge \ t' \, \upsilon \, t$$

Reasoning within a simulation up to equivalence agrees with similarity as long as the equivalence is contained in the extension of similarity to open terms. The requirement of an equivalence relation can be relaxed further to a preorder, thus even $(\!|\eta|\!) = (\precsim_b \eta \precsim_b)$ is conceivable.

Apart from the "bisimulation up to equivalence" of [Mil89] already mentioned, the notion of "simulation up to" has appeared sometimes in the literature. So, e.g. Lassen [Las98a] discusses simulation "up to similarity" and

simulation "up to context" while in [San98] simulation "up to improvement and context" is treated by Sands.

Subsequently, we will try to shed some light on the general principle behind simulation up to and why it is correct. Therefore, consider the operator $(\!|\cdot|\!)$ on relations over closed terms again.

Definition 2.4.11. *An operator $(\!|\cdot|\!)$ is said to* support the experiment $[\cdot]$ *whenever $(\!|\eta|\!) \subseteq \mathcal{T}_0{}^2$ for every relation $\eta \subseteq \mathcal{T}_0{}^2$ and the following is true:*

$$\forall s, t: \ s \ (\!|\eta|\!) \ t \implies (\exists s', t': \ s' \ \eta \ t' \ \wedge \ (s' \ [\,(\!|\eta|\!)^o\,] \ t' \implies s \ [\,(\!|\eta|\!)^o\,] \ t))$$

Then η is called a simulation up to experiment support *if $\eta \subseteq [\,(\!|\eta|\!)^o\,]$ holds.*

We will also speak of a *simulation up to* $(\!|\cdot|\!)$ for a simulation up to experiment support. It is then to show that $\eta \subseteq [\,(\!|\eta|\!)^o\,]$ suffices for $\eta \subseteq \lesssim_b$, i.e. that η is included in similarity. The following points out that the notion of $(\!|\cdot|\!)$ to support the experiment $[\cdot]$ is appropriate.

Lemma 2.4.12. *If η is a simulation up to $(\!|\cdot|\!)$ then $(\!|\eta|\!)$ is a simulation.*

Proof. For a simulation $\eta \subseteq \mathcal{T}_0{}^2$ up to $(\!|\cdot|\!)$ we will show that $(\!|\eta|\!) \subseteq [\,(\!|\eta|\!)^o\,]$ is valid. So we assume closed terms $s, t \in \mathcal{T}_0$ such that $s \ (\!|\eta|\!) \ t$ holds. Since $(\!|\cdot|\!)$ supports the experiment $[\cdot]$ there exist closed terms $s', t' \in \mathcal{T}_0$ such that $s' \ \eta \ t'$ as well as the implication $s' \ [\,(\!|\eta|\!)^o\,] \ t' \implies s \ [\,(\!|\eta|\!)^o\,] \ t$ is true. Because of $\eta \subseteq [\,(\!|\eta|\!)^o\,]$ the premise of this implication is met. Thus $s \ [\,(\!|\eta|\!)^o\,] \ t$ proves the claim. $\square$

Although the previous lemma did not, nor could it because that would require the condition $\eta \subseteq (\!|\eta|\!)$, establish that every simulation up to $(\!|\cdot|\!)$ is also a simulation, reasoning based on simulation up to experiment support is sensible.

Theorem 2.4.13. *Similarity contains all simulations up to experiment support.*

Proof. Lemma 2.4.12 shows that if η is a simulation up to $(\!|\cdot|\!)$ then $(\!|\eta|\!)$ is a simulation. Hence $(\!|\eta|\!) \subseteq [\,(\!|\eta|\!)^o\,]$ and thus $[(\!|\eta|\!)^o] \subseteq [\,[\,(\!|\eta|\!)^o\,]^o\,]$ holds, since $[\cdot^o]$ is monotone. So $[\,(\!|\eta|\!)^o\,]$ is $[\cdot^o]$-dense and thus contained in similarity. $\square$

Example 2.4.14. *We will demonstrate that simulation up to similarity is contained in similarity, i.e. that $(\!|\eta|\!) = (\lesssim_b \ \eta \ \lesssim_b)$ supports the experiment. Then simulation up to $(\!|\cdot|\!)$ results in the following proof obligation.*

$$s \ \eta \ t \implies \big(\forall \theta(\overline{s}_i): \ s \Downarrow \theta(\overline{s}_i) \implies (\exists \theta(\overline{t}_i): \ t \Downarrow \theta(\overline{t}_i) \ \wedge \ \overline{s}_i \ (\lesssim_b \ \eta \ \lesssim_b)^o \ \overline{t}_i)\big)$$

In the course of this example suppose that the extension $(\cdot)^o$ to open terms is given by $s \ \eta^o \ t \iff \forall \sigma : \ \sigma(s) \ \eta \ \sigma(t)$ for simplicity, i.e. the common way over all closing substitutions. Assume $s \ (\lesssim_b \ \eta \ \lesssim_b) \ t$ in order to show that the operator $(\!|\cdot|\!)$ supports the experiment. This means that there are terms s', t' such that $s \ \lesssim_b \ s' \ \wedge \ s' \ \eta \ t' \ \wedge \ t' \ \lesssim_b \ t$ holds. So furthermore assume $s' \ [\,(\!|\eta|\!)^o\,] \ t'$ for showing $s \ [\,(\!|\eta|\!)^o\,] \ t$ as follows:

$$s \Downarrow \theta(\overline{s}_i) \implies s' \Downarrow \theta(\overline{s}'_i) \ \wedge \ \overline{s}_i \lesssim_b^o \overline{s}'_i \qquad\qquad\qquad (\text{by } s \ \lesssim_b \ s')$$

$$s' \Downarrow \theta(\overline{s}'_i) \implies t' \Downarrow \theta(\overline{t}'_i) \ \wedge \ \overline{s}'_i \ (\!|\eta|\!)^o \ \overline{t}'_i \qquad\qquad (\text{by } s' \ [\,(\!|\eta|\!)^o\,] \ t')$$

$$\forall \sigma : \ \sigma(\overline{s}'_i) \ (\!|\eta|\!) \ \sigma(\overline{t}'_i) \qquad\qquad\qquad\qquad\qquad (\text{by } \overline{s}'_i \ (\!|\eta|\!)^o \ \overline{t}'_i)$$

$$\exists \overline{s}''_i, \overline{t}''_i : \ \sigma(\overline{s}'_i) \ \lesssim_b \ \overline{s}''_i \ \eta \ \overline{t}''_i \ \lesssim_b \ \sigma(\overline{t}'_i) \qquad\qquad (\text{by } \sigma(\overline{s}'_i) \ (\!|\eta|\!) \ \sigma(\overline{t}'_i))$$

$$t' \Downarrow \theta(\overline{t}'_i) \implies t \Downarrow \theta(\overline{t}_i) \ \wedge \ \overline{t}'_i \lesssim_b^o \overline{t}_i \qquad\qquad\qquad (\text{by } t' \ \lesssim_b \ t)$$

$$\forall \sigma : \ \sigma(\overline{s}_i) \ \lesssim_b \ \overline{s}''_i \ \eta \ \overline{t}''_i \ \lesssim_b \ \sigma(\overline{t}_i) \qquad (\text{by def. of } (\cdot)^o \text{ and trans. of } \lesssim_b)$$

$$\overline{s}_i \ (\!|\eta|\!)^o \ \overline{t}_i \qquad\qquad\qquad\qquad\qquad (\text{by def. of } (\cdot)^o \text{ and } (\!|\cdot|\!))$$

$$s \ [\,(\!|\eta|\!)^o\,] \ t \qquad\qquad (\text{by } s \Downarrow \theta(\overline{s}_i) \ \wedge \ t \Downarrow \theta(\overline{t}_i) \text{ and the def. of } [\cdot])$$

As an aside, note that the proof principle which emerges from $(\lesssim_b \ \eta \ \lesssim_b)^o$ is usually stronger than its counterpart $(\lesssim_b^o \ \eta^o \ \lesssim_b^o)$ in [Las98a, p. 107]. This is so because $s \ (\lesssim_b^o \ \eta^o \ \lesssim_b^o) \ t$ implies $s \ (\lesssim_b \ \eta \ \lesssim_b)^o \ t$ for $(\cdot)^o$ like above as well as its restricted variant in the $\boldsymbol{\lambda}_{\approx}$-calculus.

2.5   Future Work

Since the framework is greatly abstract already, there are rather few possibilities for further enhancements. Though, it will be discussed shortly whether and how the use of the $\Downarrow$-relation in the experiment could be further relaxed. So consider definition 2.4.1 where $\Downarrow$ has been replaced by υ as an additional parameter.

$$s \ [\eta]_{\upsilon} \ t \iff (\forall \theta(\overline{s}_i) : \ s \ \upsilon \ \theta(\overline{s}_i) \implies (\exists \theta(\overline{t}_i) : \ t \ \upsilon \ \theta(\overline{t}_i) \ \wedge \ \overline{s}_i \ \eta \ \overline{t}_i))$$

E.g. choosing $\upsilon = \gtrsim_c$ could be worthwhile for some situations. But what conditions on υ make the precongruence proof to go through? At least υ has to be a preorder for this and the below seems also sensible.

$$s \ \upsilon \ t \implies (t \Downarrow \implies s \Downarrow)$$

$$s \ \upsilon \ t \implies (t \ \left[\upsilon \cap \upsilon^T\right]_{\upsilon} \ s)$$

While $\Downarrow$ and $\succsim_c$ trivially meet the former, the latter requires a closer look. It essentially states that $v^T \subseteq [v \cap v^T]$ holds, i.e., whenever two terms s, t are related by v then they must be contained in the experiment of the equivalence generated by v too.

However, instead of reiterating the whole proof with this parameterised version of experiment it seems more appealing to ascertain general properties. So among others, the following conditions can be found to be of relevance for the precongruence proof.

$$s \not\Downarrow \implies \forall t: \ s \ [\eta] \ t$$
$$s \ [\eta] \ t \implies (\forall s': \ s \Downarrow s' \implies (\exists t': \ t \Downarrow t' \ \wedge \ s' \ [\eta] \ t'))$$

By the first, a non-converging term is considered smaller than anything else, a common property. The second demands that the answers itself will be contained in the experiment if the originating terms are.

It is then to be shown that the notions of the precongruence candidate being respected by evaluation and stable under reduction respectively, guarantee its inclusion in similarity for an experiment characterised in this abstract way. Thereby note that the results of section 2.2, in particular theorem 2.2.13, remain applicable as long as the extension $(\cdot)^o$ of relations to open terms is admissible.

Chapter 3

Non-deterministic Lambda-calculi

The previous chapter describes a general framework for (pre-) congruence proofs in lazy computation systems. There, the work of Howe [How89, How96] has been adapted, in order to cope with call-by-need evaluation. The feasibility of this method will be set out in the remainder of this work.

As an exhaustive example we therefore present a non-deterministic call-by-need lambda calculus in this chapter. The reason for non-determinism is to demonstrate the effect of sharing on the equations valid w.r.t. contextual equivalence. The results of [MOW98, Theorem 32] and [AF97, Theorem 5.11] indicate that the operational theories of call-by-name and call-by-need do not differ much in a deterministic scenario. At least this seems to be true as long as only the fact of convergence or divergence is observed whereas the number of reductions is not taken into account. Hence another interesting application would be improvement theory like in [San98, MS99], though it will not be treated here.

However, the motivation for our non-deterministic call-by-need lambda calculus is not merely to serve as an example. Rather, subsequent chapters of this work contain a thorough treatment of operational equivalence and possible tools to prove it. Many further aspects are covered, e.g. computation of fixed points and denotational semantics in section 5.3 and 5.4 respectively.

So this chapter is organised as follows. After the non-deterministic call-by-need lambda calculus $\boldsymbol{\lambda}_{\mathrm{ND}}$ is introduced in section 3.1, a way to prune the evaluation in environments at an arbitrary, finite depth is developed. This is

the approximation calculus $\boldsymbol{\lambda}_{\approx}$ of section 3.2, which provides a basis for the substantial proof in chapter 4 that similarity in $\boldsymbol{\lambda}_{\approx}$ forms a precongruence. In section 3.3 the link between the $\boldsymbol{\lambda}_{\mathrm{ND}}$- and the $\boldsymbol{\lambda}_{\approx}$-calculus is established by showing that the respective notions of convergence coincide. The $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus deliberately has an elementary structure in order to lay a foundation for further studies. Thus, chapter 6 will discuss possible extensions.

3.1   The Call-by-need Calculus $\boldsymbol{\lambda}_{\mathrm{ND}}$

In this section the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus will be discussed. It closely resembles the one of [KSS98, Kut99], apart from a few differences. First, the erratic nondeterministic choice is modelled by the syntactic construct `pick` rather than a constant. While this is mainly of technical nature, basing the contextual equivalence solely on the observation of converging behaviour, has a deeper impact. Disregarding divergence in its definition leads to more equations which are valid w.r.t. contextual equivalence.

On the other hand, the language construct `seq` will increase the discriminating power of contexts. Therefore, it will cause contextual preorder to be closer to similarity than this was possible in [Man04].

Having specified the language, we will define a normal-order reduction which respects sharing in that only abstractions may be copied and non-deterministic choices will not be duplicated. We will prove some fundamental properties of the normal-order reduction and then demonstrate why a definition of simulation in the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus working directly with `let`-environments fails. This is the reason for the approximation calculus $\boldsymbol{\lambda}_{\approx}$ of section 3.2.

3.1.1   Language

The language Λ_{ND} is given by the symbol E of the grammar in figure 3.1, where the non-terminal V stands for a countable set of variables. I.e., the terms are either variables, applications or formed by one of the language operators λ, `let`, `pick` and `seq`.

Since the symbol $=$ is part of the `let`-construct, we use $\equiv$ for syntactic equality up to renaming of bound variables. While the intention of abstraction, application and explicit sharing via `let` should be clear from usual call-by-need calculi, the intuitive meaning of `pick` and `seq` will be explained in a few words.

The operator `pick` represents erratic non-determinism, i.e., may unconditionally reduce to either of its arguments. In contrast to the constant `choice`

$$E ::= V \ \mid \ (\lambda V.E) \ \mid \ (E \ E) \ \mid \ (\texttt{let } V = E \texttt{ in } E)$$
$$\mid \ (\texttt{pick } E \ E) \ \mid \ (E \texttt{ seq } E)$$

Figure 3.1: Syntax for expressions in the language Λ_{ND}

in [Kut99], it is modelled as a syntactic construct, hence the different name. The constant $\texttt{choice}$ can be implemented in the $\boldsymbol{\lambda}_{\approx}$-calculus, though its representation is distinguished from $\lambda x.(\lambda y.\texttt{pick } x \ y)$ by contexts. See section 3.4.1 for a more in-depth treatment.

The construct $\texttt{seq}$ enables sequential evaluation. Given a term of the form $s \texttt{ seq } t$, the subterm s has to be evaluated first. Using strict application, i.e. argument evaluation precedes function application, is on a par with $\texttt{seq}$ because both are mutually expressible by each other. Suppose $⑤$ to denote strict application, then $\texttt{seq}$ and $⑤$ could be defined by the equations $s \texttt{ seq } t \stackrel{\text{def}}{=} ⑤(\mathbf{K} \, t, \, s)$ and $⑤(s, \, t) \stackrel{\text{def}}{=} \texttt{let } x = t \texttt{ in } x \texttt{ seq } s \, x$ respectively. So it is merely a matter of taste which one to implement. Since it is easier to interface with the calculus of [SSSS04], $\texttt{seq}$ will be set up in this work.

Like in [Man04] before, the (set) inclusion of similarity in contextual preorder will be strict. One merit of $\texttt{seq}$ in this work is to reduce that existing gap. Roughly, we could show that s is smaller than t w.r.t. similarity if s is contextually smaller than t and the answer set $\texttt{ans}(t)$ of t is finite.

Obviously, λ and $\texttt{let}$ are the only operators which *bind* variables, so Λ_{ND} forms a lazy computation language with the following arities, where @ stands for application.

$$\alpha(@) \stackrel{\text{def}}{=} \langle 0, 0 \rangle \qquad\qquad \alpha(\lambda) \stackrel{\text{def}}{=} \langle 1 \rangle$$
$$\alpha(\texttt{pick}) \stackrel{\text{def}}{=} \langle 0, 0 \rangle \qquad\qquad \alpha(\texttt{let}) \stackrel{\text{def}}{=} \langle 1, 0 \rangle$$
$$\alpha(\texttt{seq}) \stackrel{\text{def}}{=} \langle 0, 0 \rangle$$

The definition of *free* and *bound* variables is as usual and will not be repeated here. $\mathcal{FV}(s)$ will denote the set of free variables of the term s. A term which does not contain free variables is called *closed*, otherwise *open*. We frequently will refer to closed terms as *combinators*. Writing e.g. $\lambda xy.s$ as a shortcut for

$\lambda x.(\lambda y.s)$, the following combinators will be used throughout this work.

$$
\begin{array}{llll}
\mathbf{S} & \overset{\text{def}}{=} & \lambda pqr.p\,r\,(q\,r) & \qquad \mathbf{K} & \overset{\text{def}}{=} & \lambda xy.x \\[2mm]
\mathbf{I} & \overset{\text{def}}{=} & \lambda x.x & \qquad \mathbf{K2} & \overset{\text{def}}{=} & \lambda xy.y \\[2mm]
\mathbf{Y} & \overset{\text{def}}{=} & \lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x)) & \qquad \mathbf{\Omega} & \overset{\text{def}}{=} & (\lambda x.x\,x)\,(\lambda x.x\,x)
\end{array}
$$

The notion of a *context* has formally been declared in section 2.1 already. Contexts are terms with some holes, which are denoted by $[\,]$, and are *not* considered equivalent modulo renaming of bound variables: Capturing free variables is one of their major features.

Definition 3.1.1. *Let e be a term. Define the following sets of contexts:*

$$
\begin{array}{lll}
A_L \equiv [\,]\,e & L_L \equiv \texttt{let } x = [\,] \texttt{ in } e & W_L \equiv [\,] \texttt{ seq } e \\[1mm]
A_R \equiv e\,[\,] & L_R \equiv \texttt{let } x = e \texttt{ in } [\,] & W_R \equiv e \texttt{ seq } [\,]
\end{array}
$$

Some of the contexts of this definition will be used to explain reduction and evaluation in the following section. For this purpose, the common notation from section 2.1, as e.g. L_R^* for the Kleene closure of L_R-contexts, will be used.

3.1.2   Reduction and Evaluation

In this section we will provide an operational semantics for the Λ_{ND}-language by means of a small-step reduction relation. Evaluation is "lazy" in the sense that reductions do not take place under an abstraction, cf. [Abr90]. But since the $\boldsymbol{\lambda}_{\text{ND}}$-calculus respects sharing, the notion of a *weak head normal form* has to be adapted accordingly. Hence the following definition reflects what is called an *answer* in [AFM+95, AF97, MOW98].

Definition 3.1.2 (Weak Head Normal Form). *A term $L_R^*[\lambda x.s] \in \Lambda_{\text{ND}}$ is called a <u>w</u>eak <u>h</u>ead <u>n</u>ormal <u>f</u>orm (WHNF for short). A term $t \in \Lambda_{\text{ND}}$ is in WHNF iff $t \equiv L_R^*[\lambda x.s]$ for some context L_R^* and term $s \in \Lambda_{\text{ND}}$ holds.*

Contexts of the form L_R^* play an essential role in the previous definition and thus will be called *environment contexts* or *environments* for short. The goal of reducing a term is to bring it in WHNF. Therefore, the calculus $\boldsymbol{\lambda}_{\text{ND}}$ embodies the reduction rules of figure 3.2 which should be understood as templates. I.e., for the symbols $s, t, \ldots$ all kinds of terms, for $x, y, \ldots$ arbitrary variables and for D any context may be substituted.

Furthermore, we adopt the distinct variable convention, i.e. suppose all bound variables to be distinct from each other and the free variables. Since we implicitly assume this convention to take effect after every reduction step, notably the duplicate occurrence of the term $\lambda y.r$ in the specification of the (cp)-rule below does not pose any problem.

$$\mathtt{let}\ x = (\mathtt{let}\ y = t_y\ \mathtt{in}\ t_x)\ \mathtt{in}\ s \xrightarrow{\text{llet}} \mathtt{let}\ y = t_y\ \mathtt{in}\ (\mathtt{let}\ x = t_x\ \mathtt{in}\ s) \quad \text{(llet)}$$

$$(\mathtt{let}\ x = t_x\ \mathtt{in}\ s)\ t \xrightarrow{\text{lapp}} \mathtt{let}\ x = t_x\ \mathtt{in}\ (s\,t) \quad \text{(lapp)}$$

$$(\lambda x.s)\ t \xrightarrow{\text{lbeta}} \mathtt{let}\ x = t\ \mathtt{in}\ s \quad \text{(lbeta)}$$

$$(\mathtt{let}\ x = r\ \mathtt{in}\ s)\ \mathtt{seq}\ t \xrightarrow{\text{lseq}} \mathtt{let}\ x = r\ \mathtt{in}\ (s\ \mathtt{seq}\ t) \quad \text{(lseq)}$$

$$(\lambda x.s)\ \mathtt{seq}\ t \xrightarrow{\text{eseq}} t \quad \text{(eseq)}$$

$$\mathtt{pick}\ s\ t \xrightarrow{\text{nd, left}} s \quad \text{(nd, left)}$$

$$\mathtt{pick}\ s\ t \xrightarrow{\text{nd, right}} t \quad \text{(nd, right)}$$

$$\mathtt{let}\ x = \lambda y.r\ \mathtt{in}\ D[x] \xrightarrow{\text{cp}} \mathtt{let}\ x = \lambda y.r\ \mathtt{in}\ D[\lambda y.r] \quad \text{(cp)}$$

Figure 3.2: The reduction rules of the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus

The purpose of (llet), (lapp) and (lseq) is mainly to rearrange $\mathtt{let}$-bindings for subsequent reductions. The ordinary (β)-rule is superseded by (lbeta) which just creates a $\mathtt{let}$-binding. The rules (nd, left) and (nd, right) implement the non-deterministic choice while (eseq) returns the second argument of a $\mathtt{seq}$ once the first has been evaluated to an abstraction.

With (cp) one occurrence of a variable bound to an abstraction may be replaced with a *copy* of this abstraction, hence the name of the rule. Note that an abstraction is copied to *exactly one* location at a time. This conforms to the earlier work [AFM⁺95, AF97, MOW98] on call-by-need $\boldsymbol{\lambda}$-calculi and is also closer to the implementation of lazy functional programming languages than a simultaneous substitution would be.

Definition 3.1.3 (Reduction). *Figure 3.2 defines the reduction rules of the calculus* $\boldsymbol{\lambda}_{\mathrm{ND}}$*, which induce a reduction relation as follows. For terms* $s, t \in \Lambda_{\mathrm{ND}}$

we write $s \xrightarrow{C,\ a} t$ if there exists a context $C \in \mathcal{C}$ and terms $s', t' \in \Lambda_{\mathrm{ND}}$ such that $s \equiv C[s'] \ \wedge \ t \equiv C[t']$ and $s' \xrightarrow{a} t'$ holds by some (a) of these rules. Thereby, s' is called redex[1] *or a-redex more precisely, because rule (a) is used. The term t' is the* reduct *and $C[t']$ is called* contractum *sometimes. In addition, let $\mathcal{D} \subseteq \mathcal{C}$ denote a subset of contexts. We then write $s \xrightarrow{\mathcal{D},\ a} t$ if $s \xrightarrow{D,\ a} t$ for some $D \in \mathcal{D}$ is true. Whenever (a) is not explicitly specified it may be any of the rules.*

As usual, the symbol $\rightarrow^+$ denotes the transitive and $\rightarrow^*$ the reflexive-transitive closure of a reduction relation. Here, the arrow may also be decorated by one or several rules and we declare the following unions of reductions.

$$\xrightarrow{\text{nd}} \ \overset{\text{def}}{=\!=} \ \xrightarrow{\text{nd, left}} \ \cup \ \xrightarrow{\text{nd, right}} \tag{nd}$$

$$\xrightarrow{\text{lll}} \ \overset{\text{def}}{=\!=} \ \xrightarrow{\text{llet}} \ \cup \ \xrightarrow{\text{lapp}} \ \cup \ \xrightarrow{\text{lseq}} \tag{lll}$$

The notion $\xrightarrow{\mathcal{D},\ a}$ of a reduction (a) only to take place within certain contexts belonging to the subset $\mathcal{D} \subseteq \mathcal{C}$ will be used frequently. In particular the *normal-order reduction* will be restricted to *reduction contexts*, although compared to other $\boldsymbol{\lambda}$-calculi, as e.g. [Gor99, San98, Sch00], this is not a sufficient condition.

Since the `seq`-construct has to be incorporated into reduction contexts, the definition becomes slightly more complicated than in [Kut99, Man05]. Note that reduction contexts are *not* closed under composition.

Definition 3.1.4 (Reduction Contexts). *The class $\mathcal{R}$ of reduction contexts is inductively defined by the following rule for the symbol R while the symbol R^- denotes* weak reduction contexts*:*

$$R^- ::= [\,]\ \mid\ R^-[A_L]\ \mid\ R^-[W_L]$$
$$R\ \ ::= L_R^*[R^-]\ \mid\ L_R^*[\texttt{let } x = R^- \texttt{ in } R[x]]$$

Let $t \in \Lambda_{\mathrm{ND}}$ be a term. Then a weak reduction context R^- is called maximal *for $R^-[t]$ if and only if there is no other term $t' \in \Lambda_{\mathrm{ND}}$ and no weak reduction context $R^{-\prime}$ such that $R^{-\prime}[t'] \equiv R^-[t]$ holds and t' is a subterm of t.*

The notion of *normal-order reduction* may then intuitively be described as follows. Descend into contexts of the form L_R and subsequently R^-, until one of the rules (lapp), (lbeta), (lseq), (eseq) or (nd) becomes applicable, the case 1 of definition 3.1.5. If during this process a variable is encountered, its binding is

---

[1]which is an acronym for "reducible expression"

examined. Whenever possible, perform (cp) or (llet) for the variable in question, i.e., cases 3 and 4 respectively. Otherwise, in case 2, if the variable is bound to an application, descend into the R^--context as far as necessary in order to apply (lapp), (lbeta), (lseq), (eseq) or (nd).

Definition 3.1.5 (Normal-Order Reduction). *A reduction $s \xrightarrow{\mathcal{R},\, a} t$ is called* normal-order *and depicted by $s \xrightarrow{n,\, a} t$ if and only if it matches one of the following.*

1. *If $s \equiv L_R^*[R^-[r]]$ with a weak reduction context R^- and rule* (lapp), (lbeta), (lseq), (eseq), (nd, left) *or* (nd, right) *is applied to r.*

2. *If $s \equiv L_R^*[\mathtt{let}\ x = R^-[r]\ \mathtt{in}\ R[x]]$ with R a reduction context and R^- a weak reduction context such that rule* (lapp), (lbeta), (lseq), (eseq), (nd, left) *or* (nd, right) *is applied to r.*

3. *If $s \equiv L_R^*[\mathtt{let}\ x = \lambda y.r\ \mathtt{in}\ R[x]] \xrightarrow{n,\, cp} L_R^*[\mathtt{let}\ x = \lambda y.r\ \mathtt{in}\ R[\lambda y.r]] \equiv t$ by rule* (cp) *for some reduction context R.*

4. *If rule* (llet) *is applied as follows:*
$$s \equiv L_R^*[\mathtt{let}\ x = (\mathtt{let}\ y = t_y\ \mathtt{in}\ t_x)\ \mathtt{in}\ R[x]] \xrightarrow{n,\, llet}$$
$$L_R^*[\mathtt{let}\ y = t_y\ \mathtt{in}\ (\mathtt{let}\ x = t_x\ \mathtt{in}\ R[x])] \equiv t$$

Note that, for $R^-[r]$ in the cases 1 and 2 of the above definition, the weak reduction context R^- is *not* maximal. Rather, it is the one directly "above" the maximal weak reduction context, i.e. the maximal weak reduction context extends into the outermost application or $\mathtt{seq}$-operator of the redex r. It is easy to see that none of the rules permitted in the cases 1 and 2 is applicable within a smaller weak reduction context. Hence the normal-order redex is unique and, except for the non-deterministic rules, also the reduction itself.

The definition of the normal-order reduction is well-suited for the aforementioned purpose to determine weak head normal forms of terms. I.e., we state without proof, that the normal-order reduction is "standard" in that whenever for some term t there is a reduction to WHNF then t has also a normal-order reduction to the same WHNF.

Therefore the notion of *convergence* in the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus is defined by a normal-order reduction sequence to a term of the form $L_R^*[\lambda x.t]$, i.e. a WHNF.

Definition 3.1.6 (Convergence). *Let $s, t \in \Lambda_{\mathrm{ND}}$ be terms. Then s converges to t, denoted by $s \Downarrow t$, if and only if $s \xrightarrow{n}{}^* t$ and t is a WHNF. We write $s \Downarrow$ if there exists a term $t \in \Lambda_{\mathrm{ND}}$ such that $s \Downarrow t$ and $s \nDownarrow$ if not.*

From the definition of the normal-order it can easily be seen how the conditions of definition 2.3.1 are satisfied and $\boldsymbol{\lambda}_{\mathrm{ND}}$ forms a lazy computation system.

Below, a converging normal-order reduction is illustrated briefly as well as the possible cases to reach a weak head normal in one normal-order step.

Example 3.1.7. *The normal-order reduction of $(\lambda x.x)\,\big((\lambda y.(\lambda z.q))\,r\big)$ is*

$$(\lambda x.x)\,\big((\lambda y.(\lambda z.q))\,r\big) \xrightarrow{n,\ lbeta} \mathtt{let}\ x = ((\lambda y.(\lambda z.q))\,r)\ \mathtt{in}\ x$$

$$\xrightarrow{n,\ lbeta} \mathtt{let}\ x = (\mathtt{let}\ y = r\ \mathtt{in}\ (\lambda z.q))\ \mathtt{in}\ x$$

$$\xrightarrow{n,\ llet} \mathtt{let}\ y = r\ \mathtt{in}\ (\mathtt{let}\ x = (\lambda z.q)\ \mathtt{in}\ x)$$

$$\xrightarrow{n,\ cp} \mathtt{let}\ y = r\ \mathtt{in}\ (\mathtt{let}\ x = (\lambda z.q)\ \mathtt{in}\ (\lambda z.q))$$

Thus $(\lambda x.x)\,\big((\lambda y.(\lambda z.q))\,r\big) \Downarrow \mathtt{let}\ y = r\ \mathtt{in}\ (\mathtt{let}\ x = (\lambda z.q)\ \mathtt{in}\ (\lambda z.q))$ holds.

Remark 3.1.8. *If $s \xrightarrow{n} t$ for two terms $s, t \in \Lambda_{\mathrm{ND}}$ such that t is a weak head normal form but s is not, then s must be of one of the following forms where the respective reduction rule is used:*

- $s \equiv L_R^*[(\lambda y.L_R'^*[\lambda z.q])\,r] \xrightarrow{n,\ lbeta} L_R^*[\mathtt{let}\ y = r\ \mathtt{in}\ L_R'^*[\lambda z.q]]$

- $s \equiv L_R^*[(\lambda y.p)\ \mathtt{seq}\ (L_R'^*[\lambda z.q])] \xrightarrow{n,\ eseq} L_R^*[L_R'^*[\lambda z.q]]$

- $s \equiv L_R^*[\mathtt{pick}\ (L_R'^*[\lambda z.q])\ r] \xrightarrow{n,\ nd,\ left} L_R^*[L_R'^*[\lambda z.q]]$

- $s \equiv L_R^*[\mathtt{pick}\ r\ (L_R'^*[\lambda z.q])] \xrightarrow{n,\ nd,\ right} L_R^*[L_R'^*[\lambda z.q]]$

- $s \equiv L_R^*[\mathtt{let}\ y = \lambda z.q\ \mathtt{in}\ L_R'^*[y]] \xrightarrow{n,\ cp} L_R^*[\mathtt{let}\ y = \lambda z.q\ \mathtt{in}\ L_R'^*[\lambda z.q]]$

Internal Reductions

Contrary to calculi like [AF97, MOW98, Sch00], the normal-order reduction in the calculus $\boldsymbol{\lambda}_{\mathrm{ND}}$ cannot simply be specified as the closure of the reduction rules w.r.t. reduction contexts. Therefore, we now examine reductions which are non-normal-order, in particular those within reduction contexts.

Definition 3.1.9. *A reduction by a rule (a) within some context $C \in \mathcal{C}$ is called internal and depicted by $\xrightarrow{i,\ C,\ a}$ (or $\xrightarrow{iC,\ a}$ for short) if it is not normal-order.*

This notion extends to subsets of contexts in the obvious way. It is then straightforward to make the following observations.

Lemma 3.1.10. *There is no $\xrightarrow{i\mathcal{R},\ a}$-reduction for $a \in \{\ lapp,\ lbeta,\ lseq,\ eseq,\ nd\ \}$*

Proof. By inspecting the structure of reduction contexts. □

Lemma 3.1.11. *A reduction inside a reduction context $L_R^*[R^-]$ is internal if* (llet) *or* (cp) *is applied to the subterm* let $x = r$ in s *of* $L_R^*[R^-[\mathtt{let}\ x = r\ \mathtt{in}\ s]]$ *and either there is no reduction context R such that $s \equiv R[x]$ holds or R^- is not the empty context.*

Lemma 3.1.12. *A reduction for a term $L_R^*[\mathtt{let}\ x = R^-[\mathtt{let}\ y = r\ \mathtt{in}\ s]\ \mathtt{in}\ R[x]]$ is internal within a reduction context if R is a reduction context and the rule* (llet) *or* (cp) *is applied to* let $y = r$ in s.

Proof of Lemma 3.1.11 and 3.1.12. By the definition of normal-order. □

As an analysis shows, a $(i\mathcal{R},\ \text{llet})$-reduction may *only* be of a form as in the previous two lemmas. The target location of the copy operation does indeed matter, hence for $(i\mathcal{R},\ \text{cp})$ there is one additional possibility.

Corollary 3.1.13. *Let $s, t \in \Lambda_{\mathrm{ND}}$ be terms. Then $s \xrightarrow{i\mathcal{R}} t$ if and only if rule* (llet) *or* (cp) *is applied to the subterm s' and one of the following holds:*

1. *$s \equiv L_R^*[\mathtt{let}\ x = q\ \mathtt{in}\ C[x]]$ where $s' \equiv \mathtt{let}\ x = q\ \mathtt{in}\ C[x]$ and C is not a reduction context.*

2. *$s \equiv L_R^*[R^-[s']]$ where $s' \equiv \mathtt{let}\ y = q\ \mathtt{in}\ r$ and R^- is not the empty context or there is no reduction context R such that $s \equiv R[y]$.*

3. *$s \equiv L_R^*[\mathtt{let}\ x = R^-[\mathtt{let}\ y = q\ \mathtt{in}\ r]\ \mathtt{in}\ R[x]]$ where R is a reduction context and $s' \equiv \mathtt{let}\ y = q\ \mathtt{in}\ r$.*

It easily can be seen that the property of a weak head normal form is invariant w.r.t. internal reductions within reduction contexts.

Lemma 3.1.14. *Let $s, t \in \Lambda_{\mathrm{ND}}$ be terms such that $s \xrightarrow{i\mathcal{R}} t$ holds. Then t is a WHNF only if s is a WHNF.*

Proof. By case analysis on the contraposition of the claim. □

3.1.3   Contextual (Pre-) Congruence

The way it has been defined in the previous section, convergence exhibits the so-called *may convergence*, i.e. $s \Downarrow$ holds if there is *any* normal-order reduction sequence starting with s and leading to a WHNF. For a non-deterministic calculus, like e.g. [Las98b], the complementary notion of *must convergence* also makes sense, i.e., *all* normal-order reduction sequences starting with s result in a WHNF. Since *may divergence*, i.e., the existence of infinite normal-order reduction sequences, is the logical negation of must convergence, it may equally well be used, cf. [MSC99a, Kut99, SS03a], instead of must convergence. However, for reasons of simplicity this work only regards may convergence.

Definition 3.1.15. *The* contextual preorder $\lesssim_{\Lambda_{\mathrm{ND}}, c}$ *is defined by*

$$s \lesssim_{\Lambda_{\mathrm{ND}}, c} t \iff \forall C: \ C[s] \Downarrow \implies C[t] \Downarrow$$

and contextual equivalence $\simeq_{\Lambda_{\mathrm{ND}}, c}$ *by*

$$s \simeq_{\Lambda_{\mathrm{ND}}, c} t \iff s \lesssim_{\Lambda_{\mathrm{ND}}, c} t \ \wedge \ t \lesssim_{\Lambda_{\mathrm{ND}}, c} s$$

Obviously, these relations are reflexive, transitive, compatible with contexts and, in the case of $\simeq_c$, symmetric. Therefore $\simeq_c$ ($\lesssim_c$) is a (pre-) congruence. Establishing contextual equivalence is in general not straightforward, since its definition quantifies over all possible contexts. The following lemma considerably reduces the "amount" of contexts which have to be taken into account.

Lemma 3.1.16 (Context Lemma). *Let $s \in \Lambda_{\mathrm{ND}}$ be a term and $T \subseteq \Lambda_{\mathrm{ND}}$ a set of terms satisfying the following: For every reduction context $R \in \mathcal{R}$ with $R[s] \Downarrow$ there is a term $t \in T$ such that $R[t] \Downarrow$ holds.*
Then this property is also valid for general contexts, i.e. for every context $C \in \mathcal{C}$ with $C[s] \Downarrow$ there exists $t \in T$ such that $C[t] \Downarrow$ is true.

Proof. For $1 \leq i \leq k$ let $s_i \in \Lambda_{\mathrm{ND}}$ be terms and $T_i \subseteq \Lambda_{\mathrm{ND}}$ sets of terms. We will then show the following claim:

> If for every reduction context $R \in \mathcal{R}$ the following property holds: $R[s_i] \Downarrow$ implies that there is a $t_i \in T_i$ such that $R[t_i] \Downarrow$ too. Then for every multi-context $C \in \mathcal{C}$ with k holes, $C[s_1, \ldots, s_k] \Downarrow$ implies $\exists (t_1, \ldots, t_k) \in T_1 \times \cdots \times T_k: \ C[t_1, \ldots, t_k] \Downarrow$.

The proof is by induction on the lexicographical ordering consisting of the length of a normal order reduction $C[s_1, \ldots, s_k] \xrightarrow{n}{}^{*} p$ to some weak head normal form

p and the number k of C's holes. For the induction base consider the case for a context C with only one hole, where $C[s_i]$ already is a weak head normal form. Then either $C[t_i]$ for some $t_i \in T_i$ is a weak head normal form too, or the hole is in a reduction context and the precondition proves the claim.

So for the induction step we assume the proposition to hold for all terms s_i, all sets of terms T_i and all contexts smaller than C w.r.t. the given ordering. Then there are the following possibilities:

- In the case the first reduction of the sequence $C[s_1, \ldots, s_k] \xrightarrow{n}{}^* p$ is of the form $C[s_1, \ldots, s_k] \xrightarrow{n} C'[s_1, \ldots, s_k]$ the induction hypothesis applied to the context C' proves the claim.

  Note that this applies to the rules (llet), (lapp), (lbeta) and (lseq) but for the remaining reduction rules only if none of the subterms s_j is affected.

- For reductions of the form $C[s_1, \ldots, s_k] \xrightarrow{n} C'[s'_1, \ldots, s'_m]$ only the following rules have to be taken into account:

  - An application of rule (nd) may "pick" a subset of the s_i. I.e., $m \leq k$ holds and for every $1 \leq j \leq m$ there is a $1 \leq i \leq k$ such that $s'_j \equiv s_i$ is valid.
    This selection emerges for the reduction $C[t_1, \ldots, t_k] \xrightarrow{n} C'[t'_1, \ldots, t'_m]$ as well and therefore the induction hypothesis may be applied since the length of the normal order reduction sequence to a weak head normal form has been decreased.

  - If the first argument of a `seq` has been discarded by rule (eseq), the argument is the same as in the previous case.

  - For (cp), when a hole is within the copied expression, the corresponding s_i has been duplicated, i.e. there is a $1 \leq j \leq m$ and a variable renaming σ such that $s_j \equiv \sigma(s_i)$ holds. Note that, by the variable convention, σ only renames free variables of s_i which become bound by C. Therefore the precondition $R[s_j] \Downarrow \implies \exists t_j \in T_j : R[t_j] \Downarrow$ remains valid for s_j, too. Thus the proposition is shown by the induction hypothesis, since the length of the normal order reduction sequence to a weak head normal form has been decreased.

- The remaining possibility is that some hole $\cdot_i$ of $C[\cdot_1, \ldots, \cdot_k]$ is found in a reduction context. From the definition of a reduction context it is easy to show that this case is independent of the terms filled in the holes. Now let $C' \equiv C[[\ ]_1, \ldots, [\ ]_{i-1}, s_i, [\ ]_{i+1}, \ldots, [\ ]_k]$ then the terms

$C'[s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_k]$ and $C[s_1, \ldots, s_k]$ have the same normal order reduction because $C'[s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_k] \equiv C[s_1, \ldots, s_k]$. Since C' is smaller than C, i.e. has $k-1$ holes, the induction hypothesis applies, so there are $(t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_k) \in T_1 \times \cdots \times T_{i-1} \times T_{i+1} \times \cdots \times T_k$ such that $C'[t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_k]$ may converge, too. From the above, we have that $R \equiv C[t_1, \ldots, t_{i-1}, [\,], t_{i+1}, \ldots, t_k]$ is also a reduction context and therefore $R[s_i] \implies \exists t_i \in T_i : R[t_i] \Downarrow$ by the precondition. Thus by construction of R there exist $(t_1, \ldots, t_k) \in T_1 \times \cdots \times T_k$ such that $C[t_1, \ldots, t_k]$ converges. $\qquad\qquad\square$

The lemma is especially useful in the case $T = \{t\}$, i.e. where T is a singleton, from which the following corollary arises.

Corollary 3.1.17. *Let $s, t \in \Lambda_{\mathrm{ND}}$ be terms such that $R[s] \Downarrow$ implies $R[t] \Downarrow$ for every reduction context $R \in \mathcal{R}$. Then $s \lesssim_c t$ is true.*

For proving contextual equivalence, a case analysis may now be limited to reduction contexts instead of arbitrary ones, although their number is still infinite. Hence there is good reason for a more stepwise method in proving equations.

So the ambition is to define similarity in such a manner that it implies contextual preorder. However, the following example makes clear that in the calculus $\boldsymbol{\lambda}_{\mathrm{ND}}$ it is impossible to apply the usual approach "reduce to weak head normal form and apply to fresh arguments" like e.g. in [Abr90].

Example 3.1.18. *The so-called* "shifting `let` over λ" *transformation, is in general not correct w.r.t. contextual equivalence. Therefore consider the closed terms $s \equiv$ `let` $v =$ `pick` **K K2** `in` $\lambda w.v$ and $t \equiv \lambda w.$`let` $v =$ `pick` **K K2** `in` v which are* not *contextually equivalent. Rather, they are distinguished by the context $C \equiv$ `let` $f = [\,]$ `in` $((f\,\mathbf{K})\,(f\,\mathbf{K})\,\Omega\,\Omega\,\mathbf{K})$ in the following way. Concerning t we may construct a normal-order reduction sequence $C[t] \xrightarrow{n}{}^{*} L_R^*[\mathbf{K}]$ whereas there is no converging normal-order reduction sequence for $C[s]$ since v is shared.*

*Obviously, the terms s and t are weak head normal forms and if applied to an arbitrary (dummy) argument both may either yield **K** or **K2**. Hence s and t could not be distinguished by application to an argument.*

The example suggests that, because of the `let`-environments, weak head normal forms do not carry enough information in order to be distinguished solely by application to arguments. Therefore the calculus $\boldsymbol{\lambda}_{\approx}$ of the subsequent section eliminates environments from the weak head normal forms.

This has the additional benefit that proving the precongruence candidate stable under the rule (llet) becomes obsolete, a task which seemed to be infeasible. The next example illustrates that in the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus the rule (llet) is generally necessary to find a WHNF.

Example 3.1.19. *Consider the term $s \equiv \mathtt{let}\ x = (\mathtt{let}\ y = t_y\ \mathtt{in}\ \lambda z.t)\ \mathtt{in}\ x$ which obviously has a WHNF by the following normal-order reduction:*

$$\mathtt{let}\ x = (\mathtt{let}\ y = t_y\ \mathtt{in}\ \lambda z.t)\ \mathtt{in}\ x$$

$$\xrightarrow{n,\ llet} \mathtt{let}\ y = t_y\ \mathtt{in}\ (\mathtt{let}\ x = \lambda z.t\ \mathtt{in}\ x)$$

$$\xrightarrow{n,\ cp} \mathtt{let}\ y = t_y\ \mathtt{in}\ (\mathtt{let}\ x = \lambda z.t\ \mathtt{in}\ \lambda z.t)$$

Apparently, the effect of (llet) *cannot be accomplished without it. E.g., making a copy of the whole environment $\mathtt{let}\ y = t_y\ \mathtt{in}\ \lambda z.t$ is obviously not an option.*

3.2 The Approximation Calculus $\boldsymbol{\lambda}_{\approx}$

This section introduces the $\boldsymbol{\lambda}_{\approx}$-calculus. It will enable us to define a sensible notion of simulation for the calculus $\boldsymbol{\lambda}_{\mathrm{ND}}$ from the preceding section. Example 3.1.18 shows that a direct definition in the calculus $\boldsymbol{\lambda}_{\mathrm{ND}}$ is not easily possible. As we have already argued there, the difficulty results from the environments in a weak head normal form.

Therefore, the reduction in the $\boldsymbol{\lambda}_{\approx}$-calculus has the ability to eliminate these environments from weak head normal forms. This sometimes requires to cut off evaluations which otherwise could continue arbitrarily deep. Hence we speak of the "approximation calculus", since this process permits to come as close as desired to the original terms.

Section 3.4.2 explains how this relates, to some extent, to call-by-value evaluation. However, we will not pursue such an approach because calculi which represent $\mathtt{let}$ implicitly by applications of the form $(\lambda x.s)\, t$, are not capable of a recursive $\mathtt{letrec}$, a potential subject of further research.

So weak head normal forms of the calculus $\boldsymbol{\lambda}_{\approx}$ are simply abstractions. This facilitates the definition of a simulation and makes its precongruence proof accessible for Howe's method from [How89, How96].

First, in the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus, a distinction between the notion *canonical* and *non-canonical* was not possible for the $\mathtt{let}$-operator, since a $\mathtt{let}$-term could be either a WHNF or a redex. Secondly, the reduction rule (llet) becomes obsolete in the $\boldsymbol{\lambda}_{\approx}$-calculus. Proving the precongruence candidate stable under reduction w.r.t. this rule seemed intractable in the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus.

3.2.1   Language

As figure 3.3 shows, a special constant $\odot$ (pronounced "stop") is added to the language which is now designated by $\Lambda_{\approx}$. The reduction rules of the $\boldsymbol{\lambda}_{\approx}$-calculus

$$E ::= V \mid \odot \mid (\lambda x.E) \mid (E\ E) \mid (\texttt{let } x = E \texttt{ in } E) \mid (\texttt{pick } E\ E)$$

Figure 3.3: Syntax for expressions in the language $\Lambda_{\approx}$

in figure 3.4 evolve from the ones in $\boldsymbol{\lambda}_{\mathrm{ND}}$ as follows. First, by the rule (stop)

$$(\texttt{let } x = t_x \texttt{ in } s)\, t \xrightarrow{\text{lapp}}_{\boldsymbol{\lambda}_{\approx}} \texttt{let } x = t_x \texttt{ in } (s\, t) \tag{lapp}$$

$$(\lambda x.s)\, t \xrightarrow{\text{lbeta}}_{\boldsymbol{\lambda}_{\approx}} \texttt{let } x = t \texttt{ in } s \tag{lbeta}$$

$$(\texttt{let } x = r \texttt{ in } s)\ \texttt{seq}\ t \xrightarrow{\text{lseq}}_{\boldsymbol{\lambda}_{\approx}} \texttt{let } x = r \texttt{ in } (s\ \texttt{seq}\ t) \tag{lseq}$$

$$(\lambda x.s)\ \texttt{seq}\ t \xrightarrow{\text{eseq}}_{\boldsymbol{\lambda}_{\approx}} t \tag{eseq}$$

$$\texttt{pick}\ s\ t \xrightarrow{\text{nd, left}}_{\boldsymbol{\lambda}_{\approx}} s \tag{nd, left}$$

$$\texttt{pick}\ s\ t \xrightarrow{\text{nd, right}}_{\boldsymbol{\lambda}_{\approx}} t \tag{nd, right}$$

$$\texttt{let } x = s \texttt{ in } t \xrightarrow{\text{cpa}}_{\boldsymbol{\lambda}_{\approx}} t[s/x] \tag{cpa}$$
$$\text{where } s \equiv \lambda z.q \text{ or } s \equiv \odot$$

$$s \xrightarrow{\text{stop}}_{\boldsymbol{\lambda}_{\approx}} \odot \quad \text{if } s \not\equiv \odot \tag{stop}$$

Figure 3.4: The reduction rules of the $\boldsymbol{\lambda}_{\approx}$-calculus

which may reduce every non-$\odot$ term to $\odot$, a further level of non-determinism is introduced. As there is no rule for $\odot$, this delimits the reduction, i.e. evaluation is pruned underneath. Along with the existing non-determinism of the calculus, we will utilise rule (stop) in order to represent every term by, so to speak, a set of terms which have been evaluated to varying depth.

Since it is our goal to eliminate top-level environments, it is natural to completely substitute terms that could not be reduced further, namely $\odot$ and abstractions, and remove their bindings with the rule (cpa) in parallel. If (cp) was extended such that also $\odot$ can be copied, the rule (cpa) could also be regarded as a combination of certain, mostly internal reductions of type (cp), followed by a garbage collection, i.e., a removal of unnecessary `let`-bindings. So the only reduction rule actually new is (stop), and it is no surprise, that in section 3.3 we can show the original (cp)-rule to become obsolete.

All these reductions will be permitted inside arbitrary surface contexts, which are denoted by the symbol $\mathcal{S}$ and defined as follows.

Definition 3.2.1 (Surface Contexts). *The class $\mathcal{S}$ of* surface contexts *is given by the following rules for the symbol S where e means any expression:*

$$
\begin{aligned}
S ::= \ & [\,] \ \mid \ S \, e \ \mid \ e \, S \ \mid \\
& \texttt{let } x = e \texttt{ in } S \ \mid \ \texttt{let } x = S \texttt{ in } e \ \mid \\
& \texttt{pick } S \, e \ \mid \ \texttt{pick } e \, S \ \mid \\
& S \ \texttt{seq } e \ \mid \ e \ \texttt{seq } S
\end{aligned}
$$

It is easy to see that surface contexts are closed under composition, thus the results of lemma 2.3.14 and corollary 2.3.16 apply. Moreover, they obey the significant property that a hole does not occur under an abstraction. This closely relates to the idea that only abstractions may be copied.

Definition 3.2.2 (Approximation Reduction). *The reduction rules of the calculus $\boldsymbol{\lambda}_{\approx}$ in figure 3.4 define an* approximation reduction *as follows. For terms $s, t \in \Lambda_{\approx}$ we write $s \xrightarrow{S,\ a}_{\boldsymbol{\lambda}_{\approx}} t$ if there exists a surface context $S \in \mathcal{S}$ and terms $s', t' \in \Lambda_{\approx}$ such that $s \equiv S[s'] \ \wedge \ t \equiv S[t']$ and $s' \xrightarrow{a}_{\boldsymbol{\lambda}_{\approx}} t'$ hold. If not explicitly specified, (a) may be any of the declared rules.*

The notions of redex, reduct *and* contractum *are used accordingly. Furthermore, we write $s \xrightarrow{S,\ a}_{\boldsymbol{\lambda}_{\approx}} t$ if $s \xrightarrow{S,\ a}_{\boldsymbol{\lambda}_{\approx}} t$ for some $S \in \mathcal{S}$ is true. If it is clear from context, the subscript $\boldsymbol{\lambda}_{\approx}$ might be omitted, in particular when one of the rules* (cpa) *or* (stop) *is used.*

It is remarkable that, beyond the reductions (nd) and (stop), in the $\boldsymbol{\lambda}_{\approx}$-calculus an additional degree of non-determinism arises from the choice between several possible surface contexts where reduction could be performed.

Again, for a reduction relation, the symbols $\rightarrow^{+}$ and $\rightarrow^{*}$ denote the transitive and reflexive-transitive closure respectively. Also, the arrow may be decorated by one or several rules. As before, the rules (nd, left) and (nd, right) are

combined into the following union.

$$\xrightarrow{\text{nd}}_{\boldsymbol{\lambda}_\approx} \stackrel{\text{def}}{\equiv} \xrightarrow{\text{nd, left}}_{\boldsymbol{\lambda}_\approx} \cup \xrightarrow{\text{nd, right}}_{\boldsymbol{\lambda}_\approx} \tag{nd}$$

While [How96] aims mainly at big-step evaluation rules, the notion of *operator extensionality* from [How89, sec. 3.1] is well-suited for a small-step reduction semantics as in our $\boldsymbol{\lambda}_\approx$-calculus. Therefore, we will need a set of evaluation relations $\Downarrow_k$ which fulfils the conditions that

- $a \Downarrow_0 b$ if and only if $a \equiv b$ is an abstraction.

- If $a \Downarrow_k b$ for some $k > 0$ then a is not an abstraction but b is.

and write $a \Downarrow b$ if there is some $k \geq 0$ such that $a \Downarrow_k b$ holds.

Definition 3.2.3 (Evaluation). *For $k \geq 0$ let the relation $\Downarrow_{\boldsymbol{\lambda}_\approx,k}$ defined by*

$$s \Downarrow_{\boldsymbol{\lambda}_\approx,k} t \;\; \stackrel{def}{\Longleftrightarrow} \;\; \exists \lambda x.t' : \; s \xrightarrow{\mathcal{S}}{}^{k}_{\boldsymbol{\lambda}_\approx} t \;\wedge\; t \equiv \lambda x.t' \tag{3.2.1}$$

We say that s evaluates *or* converges *to t in k steps. If $\Downarrow_{\boldsymbol{\lambda}_\approx,k}$ is valid for some k, the symbol $\Downarrow_{\boldsymbol{\lambda}_\approx}$ will be used. Moreover, we simply write $s \Downarrow_{\boldsymbol{\lambda}_\approx}$ if there exists a term $t \in \Lambda_\approx$ satisfying $s \Downarrow_{\boldsymbol{\lambda}_\approx} t$, and $s \not\Downarrow_{\boldsymbol{\lambda}_\approx}$ if no such terms exists.*

Again, the criteria of definition 2.3.1 are obviously fulfilled, i.e. $\boldsymbol{\lambda}_\approx$ together with $\Downarrow_{\boldsymbol{\lambda}_\approx}$ is a lazy computation system. Like with reduction we will omit the subscript $\boldsymbol{\lambda}_\approx$ throughout this section, if it is clear from context. Note that the term $\odot$ is a *constant*, i.e., there is no reduction rule for it, since $\odot$ is explicitely excluded from the (stop)-rule. Thus $\odot \not\Downarrow$ is clear.

Note that in contrast to the convergence in $\boldsymbol{\lambda}_{\text{ND}}$, we demand an approximation reduction that ends in an abstraction. On the other hand, all these reductions are permitted within arbitrary surface contexts, which seems to provide considerably more freedom than normal-order reductions. However, section 3.3 will show that convergence in $\boldsymbol{\lambda}_{\text{ND}}$ and $\boldsymbol{\lambda}_\approx$ indeed coincides.

Regarding the previous comment on emulating the rule (cpa) with internal (cp)-reductions followed by a garbage collection, it can now be seen that the rule (stop) is essential: Just like in example 3.1.19, finding a weak head normal form is impossible without the rule (llet), all converging approximation reductions of the term let $x = (\text{let } y = \boldsymbol{\Omega} \text{ in } \lambda z.t) \text{ in } x$ require the (stop)-rule.

Example 3.2.4. *The term* let $x = ($let $y = \boldsymbol{\Omega}$ in $\lambda z.t)$ in x *has the following, converging approximation reduction*

$$\text{let } x = (\text{let } y = \boldsymbol{\Omega} \text{ in } \lambda z.t) \text{ in } x$$

$$\xrightarrow{\text{let } x=(\text{let } y=[\ ] \text{ in } \lambda z.q) \text{ in } x,\ stop} \text{let } x = (\text{let } y = \circledcirc \text{ in } \lambda z.t) \text{ in } x$$

$$\xrightarrow{\text{let } x=[\ ] \text{ in } x,\ cpa} \text{let } x = \lambda z.t[\circledcirc/y] \text{ in } x$$

$$\xrightarrow{[\ ],\ cpa} \lambda z.t[\circledcirc/y]$$

Thus let $x = ($let $y = \boldsymbol{\Omega}$ in $\lambda z.t)$ in $x \Downarrow_{\boldsymbol{\lambda}_{\approx}} \lambda z.t[\circledcirc/y]$ *holds. Another possible approximation reduction, which does not lead to an abstraction, is*

$$\text{let } x = (\text{let } y = \boldsymbol{\Omega} \text{ in } \lambda z.t) \text{ in } x$$

$$\xrightarrow{\text{let } x=[\ ] \text{ in } x,\ stop} \text{let } x = \circledcirc \text{ in } x$$

$$\xrightarrow{[\ ],\ cpa} \circledcirc$$

Since the evaluation relation is highly non-deterministic, it is convenient to gather all possible abstraction a term converges to.

Definition 3.2.5. *We define the* answer set $\mathtt{ans}(s) \subseteq \Lambda_{\approx}$ *of a term s as follows.*

$$\mathtt{ans}(s) \stackrel{def}{=} \{t \mid s \Downarrow t\}$$

and augment $\mathtt{ans}(\cdot)$ *to sets of terms by* $\mathtt{ans}(S) \stackrel{def}{=} \{t \mid s \in S \ \wedge \ t \in \mathtt{ans}(s)\}$.

Closed terms possess only closed answers, since reduction does not introduce free variables, which is the statement of the following corollary.

Corollary 3.2.6. *Let $s \in \Lambda_{\approx}^0$ be a closed term. Then* $\mathtt{ans}(s) \subseteq \Lambda_{\approx}^0$.

It is interesting to note that only some of the above rules may yield an abstraction directly in one step.

Remark 3.2.7. *If $s \xrightarrow{[\ ]}_{\boldsymbol{\lambda}_{\approx}} \lambda x.t$ for two terms $s, \lambda x.t \in \Lambda_{\approx}$, then s must be of one of the following forms where the respective reduction rule is used:*

- $s \equiv (\lambda z.q) \text{ seq } (\lambda x.t) \xrightarrow{[\ ],\ eseq} \lambda x.t$

- $s \equiv \text{pick } r \ (\lambda x.t) \xrightarrow{[\ ],\ nd,\ right} \lambda x.t$

- $s \equiv \texttt{pick}\ (\lambda x.t)\ r \xrightarrow{\ [\ ],\ nd,\ left\ } \lambda x.t$

- $s \equiv \texttt{let}\ y = \lambda x.t\ \texttt{in}\ y \xrightarrow{\ [\ ],\ cpa\ } \lambda x.t$ *as a special case of*

- $s \equiv \texttt{let}\ y = q\ \texttt{in}\ r \xrightarrow{\ [\ ],\ cpa\ } r[q/y] \equiv \lambda x.t$

Section 3.3 will establish the approximation of $\boldsymbol{\lambda}_{\mathrm{ND}}$-terms in the $\boldsymbol{\lambda}_{\approx}$-calculus. Therefore we need some elementary properties about reductions in $\boldsymbol{\lambda}_{\approx}$ which are the topic of the following subsections.

3.2.2   The (cpa)-reduction

Reduction by the rule (cpa) is the key to eliminate environments. Often we will use the fact that it does not change convergent behaviour. In order to prove this, we need to establish a complete set of forking diagrams for $\xrightarrow{\mathrm{cpa}}$ w.r.t. all the remaining $\xrightarrow{\mathcal{S}}_{\boldsymbol{\lambda}_{\approx}}$-reductions. In the following we will only consider the cases according to corollary 2.3.16, where a surrounding surface context already is abandoned, since surface contexts are closed under composition. Now we inspect all possible (cpa)-redexes.

If (cpa) is applied in the empty context, the whole expression must be of the form $\texttt{let}\ x = s\ \texttt{in}\ t$ with $s \equiv \odot$ or $s \equiv \lambda z.q$. Hence a (lapp)-reduction is possible in the surface context $\texttt{let}\ x = e\ \texttt{in}\ S$ only, which results in the first diagram. If (lapp) is top-level on the other hand, a reduction by (cpa) may take place in every non-empty surface context, which could be commuted easily in the following three diagrams. The last represents the case where (cpa) makes the top-level (lapp)-reduction superfluous.

$$\xleftarrow{\ \texttt{let}\ x=e\ \texttt{in}\ S,\ \mathrm{lapp}\ } \cdot \xrightarrow{\ [\ ],\ \mathrm{cpa}\ } \quad\rightsquigarrow\quad \xrightarrow{\ [\ ],\ \mathrm{cpa}\ } \cdot \xleftarrow{\ S,\ \mathrm{lapp}\ }$$

$$\xleftarrow{\ [\ ],\ \mathrm{lapp}\ } \cdot \xrightarrow{\ (\texttt{let}\ x=t_x\ \texttt{in}\ s)\,S,\ \mathrm{cpa}\ } \quad\rightsquigarrow\quad \xrightarrow{\ \texttt{let}\ x=t_x\ \texttt{in}\ (s\,S),\ \mathrm{cpa}\ } \cdot \xleftarrow{\ [\ ],\ \mathrm{lapp}\ }$$

$$\xleftarrow{\ [\ ],\ \mathrm{lapp}\ } \cdot \xrightarrow{\ (\texttt{let}\ x=S\ \texttt{in}\ s)\,t,\ \mathrm{cpa}\ } \quad\rightsquigarrow\quad \xrightarrow{\ \texttt{let}\ x=S\ \texttt{in}\ (s\,t),\ \mathrm{cpa}\ } \cdot \xleftarrow{\ [\ ],\ \mathrm{lapp}\ }$$

$$\xleftarrow{\ [\ ],\ \mathrm{lapp}\ } \cdot \xrightarrow{\ (\texttt{let}\ x=t_x\ \texttt{in}\ S)\,t,\ \mathrm{cpa}\ } \quad\rightsquigarrow\quad \xrightarrow{\ \texttt{let}\ x=t_x\ \texttt{in}\ (S\,t),\ \mathrm{cpa}\ } \cdot \xleftarrow{\ [\ ],\ \mathrm{lapp}\ }$$

$$\xleftarrow{\ [\ ],\ \mathrm{lapp}\ } \cdot \xrightarrow{\ [\ ]\,t,\ \mathrm{cpa}\ } \quad\rightsquigarrow\quad \xrightarrow{\ [\ ],\ \mathrm{cpa}\ }$$

For the rule (lseq) we have exactly the same cases as before.

$$\xleftarrow{\texttt{let } x=e \texttt{ in } S,\ \mathrm{lseq}} \cdot \xrightarrow{[\ ],\ \mathrm{cpa}} \ \rightsquigarrow\ \xrightarrow{[\ ],\ \mathrm{cpa}} \cdot \xleftarrow{S,\ \mathrm{lseq}}$$

$$\xleftarrow{[\ ],\ \mathrm{lseq}} \cdot \xrightarrow{(\texttt{let } x=t_x \texttt{ in } s)\ \texttt{seq}\ S,\ \mathrm{cpa}} \ \rightsquigarrow\ \xrightarrow{\texttt{let } x=t_x \texttt{ in } (s\ \texttt{seq}\ S),\ \mathrm{cpa}} \cdot \xleftarrow{[\ ],\ \mathrm{lseq}}$$

$$\xleftarrow{[\ ],\ \mathrm{lseq}} \cdot \xrightarrow{(\texttt{let } x=S \texttt{ in } s)\ \texttt{seq}\ t,\ \mathrm{cpa}} \ \rightsquigarrow\ \xrightarrow{\texttt{let } x=S \texttt{ in } (s\ \texttt{seq}\ t),\ \mathrm{cpa}} \cdot \xleftarrow{[\ ],\ \mathrm{lseq}}$$

$$\xleftarrow{[\ ],\ \mathrm{lseq}} \cdot \xrightarrow{(\texttt{let } x=t_x \texttt{ in } S)\ \texttt{seq}\ t,\ \mathrm{cpa}} \ \rightsquigarrow\ \xrightarrow{\texttt{let } x=t_x \texttt{ in } (S\ \texttt{seq}\ t),\ \mathrm{cpa}} \cdot \xleftarrow{[\ ],\ \mathrm{lseq}}$$

$$\xleftarrow{[\ ],\ \mathrm{lseq}} \cdot \xrightarrow{[\ ]\ \texttt{seq}\ t,\ \mathrm{cpa}} \ \rightsquigarrow\ \xrightarrow{[\ ],\ \mathrm{cpa}}$$

Again, the first case for (lbeta) is the same as for (lapp) and (lseq), whereas for a top-level application of (lbeta), the situation is slightly different since (cpa) is possible only in surface contexts of the form $(\lambda x.e)\,S$ now.

$$\xleftarrow{\texttt{let } x=e \texttt{ in } S,\ \mathrm{lbeta}} \cdot \xrightarrow{[\ ],\ \mathrm{cpa}} \ \rightsquigarrow\ \xrightarrow{[\ ],\ \mathrm{cpa}} \cdot \xleftarrow{S,\ \mathrm{lbeta}}$$

$$\xleftarrow{[\ ],\ \mathrm{lbeta}} \cdot \xrightarrow{(\lambda x.e)\ S,\ \mathrm{cpa}} \ \rightsquigarrow\ \xrightarrow{\texttt{let } x=S \texttt{ in } e,\ \mathrm{cpa}} \cdot \xleftarrow{[\ ],\ \mathrm{lbeta}}$$

For (eseq) the situation is similar:

$$\xleftarrow{\texttt{let } x=e \texttt{ in } S,\ \mathrm{eseq}} \cdot \xrightarrow{[\ ],\ \mathrm{cpa}} \ \rightsquigarrow\ \xrightarrow{[\ ],\ \mathrm{cpa}} \cdot \xleftarrow{S,\ \mathrm{eseq}}$$

$$\xleftarrow{[\ ],\ \mathrm{eseq}} \cdot \xrightarrow{(\lambda x.e)\ \texttt{seq}\ S,\ \mathrm{cpa}} \ \rightsquigarrow\ \xrightarrow{S,\ \mathrm{cpa}} \cdot \xleftarrow{[\ ],\ \mathrm{eseq}}$$

The rules (nd, left) and (nd, right) do not cause any problem even if the target of the copy operation lies inside the `pick` in question. Consequently, the following two diagrams ensue.

$$\xrightarrow{\texttt{let } x=e \texttt{ in } S,\ \mathrm{nd},\ \mathrm{left}} \cdot \xrightarrow{[\ ],\ \mathrm{cpa}} \ \rightsquigarrow\ \xrightarrow{[\ ],\ \mathrm{cpa}} \cdot \xleftarrow{S,\ \mathrm{nd},\ \mathrm{left}}$$

$$\xrightarrow{\texttt{let } x=e \texttt{ in } S,\ \mathrm{nd},\ \mathrm{right}} \cdot \xrightarrow{[\ ],\ \mathrm{cpa}} \ \rightsquigarrow\ \xrightarrow{[\ ],\ \mathrm{cpa}} \cdot \xleftarrow{S,\ \mathrm{nd},\ \mathrm{right}}$$

In the converse case, i.e. if (cpa) is applied inside the `pick`, it may be eliminated.

$$\xleftarrow{[\ ],\ \mathrm{nd},\ \mathrm{left}} \cdot \xrightarrow{\texttt{pick}\ S\ e,\ \mathrm{cpa}} \ \rightsquigarrow\ \xrightarrow{S,\ \mathrm{cpa}} \cdot \xleftarrow{[\ ],\ \mathrm{nd},\ \mathrm{left}}$$

$$\xleftarrow{[\ ],\ \mathrm{nd},\ \mathrm{right}} \cdot \xrightarrow{\texttt{pick}\ S\ e,\ \mathrm{cpa}} \ \rightsquigarrow\ \xleftarrow{[\ ],\ \mathrm{nd},\ \mathrm{right}}$$

$$\xleftarrow{[\ ],\ \mathrm{nd},\ \mathrm{right}} \cdot \xrightarrow{\texttt{pick}\ e\ S,\ \mathrm{cpa}} \ \rightsquigarrow\ \xrightarrow{S,\ \mathrm{cpa}} \cdot \xleftarrow{[\ ],\ \mathrm{nd},\ \mathrm{right}}$$

$$\xleftarrow{[\ ],\ \mathrm{nd},\ \mathrm{left}} \cdot \xrightarrow{\texttt{pick}\ e\ S,\ \mathrm{cpa}} \ \rightsquigarrow\ \xleftarrow{[\ ],\ \mathrm{nd},\ \mathrm{left}}$$

The cases for the rule (cpa) overlapping itself conceal that a "top-down" strategy seems capable to minimise the number of target locations for (cpa)-reductions.

$$\xleftarrow{\texttt{let } x{=}e \texttt{ in } S,\ \text{cpa}} \cdot \xrightarrow{[\ ],\ \text{cpa}} \quad \rightsquigarrow \quad \xrightarrow{[\ ],\ \text{cpa}} \cdot \xleftarrow{S,\ \text{cpa}}$$

$$\xleftarrow{[\ ],\ \text{cpa}} \cdot \xrightarrow{\texttt{let } x{=}e \texttt{ in } S,\ \text{cpa}} \quad \rightsquigarrow \quad \xrightarrow{S,\ \text{cpa}} \cdot \xleftarrow{[\ ],\ \text{cpa}}$$

If rule (stop) is applied to the term e to be copied by (cpa), then it has to be compensated by $\#_x(e)$ subsequent (stop)-reductions, where $\#_x(e)$ is the number of occurrences of the variable x in the term e. Note that these (stop)-reductions have to be admitted within arbitrary contexts. Another possibility is that (stop) may be applied to a target of (cpa) or a superterm of this target, or does not interfere with the copy procedure. Hence the second diagram arises. The last diagram covers the case where (stop) is applied in the empty context, while substituting the whole term with ⊙ and thus deleting every surface (cpa)-redex.

$$\xleftarrow{\texttt{let } x{=}[\ ] \texttt{ in } e,\ \text{stop}} \cdot \xrightarrow{[\ ],\ \text{cpa}} \quad \rightsquigarrow \quad \xrightarrow{[\ ],\ \text{cpa}} \cdot \xleftarrow{\mathcal{C},\ \text{stop}}{}^{\#_x(e)}$$

$$\xleftarrow{\texttt{let } x{=}e \texttt{ in } S,\ \text{stop}} \cdot \xrightarrow{[\ ],\ \text{cpa}} \quad \rightsquigarrow \quad \xrightarrow{[\ ],\ \text{cpa}} \cdot \xleftarrow{S,\ \text{stop}}$$

$$\xleftarrow{[\ ],\ \text{stop}} \cdot \xrightarrow{S,\ \text{cpa}} \quad \rightsquigarrow \quad \xleftarrow{[\ ],\ \text{stop}}$$

The observations above end up in the following lemma.

Lemma 3.2.8. *A complete set of forking diagrams for $\xrightarrow{S,\ cpa}$ w.r.t. the two sets $\xrightarrow{S}{}_{\lambda_\approx}$ and $\xrightarrow{\mathcal{C},\ stop}$ of reductions is:*

$$\xleftarrow{S,\ a} \cdot \xrightarrow{S,\ cpa} \quad \rightsquigarrow \quad \xrightarrow{S,\ cpa} \cdot \xleftarrow{S,\ a} \qquad (3.2.2)$$

$$\xleftarrow{[\ ],\ \{lapp,\ lseq\}} \cdot \xrightarrow{[\ ]t,\ cpa} \quad \rightsquigarrow \quad \xrightarrow{[\ ],\ cpa} \qquad (3.2.3)$$

$$\xleftarrow{S,\ nd} \cdot \xrightarrow{S,\ cpa} \quad \rightsquigarrow \quad \xleftarrow{S,\ nd} \qquad (3.2.4)$$

$$\xleftarrow{S,\ stop} \cdot \xrightarrow{[\ ],\ cpa} \quad \rightsquigarrow \quad \xrightarrow{[\ ],\ cpa} \cdot \xleftarrow{\mathcal{C},\ stop}{}^{*} \qquad (3.2.5)$$

$$\xleftarrow{[\ ],\ stop} \cdot \xrightarrow{S,\ cpa} \quad \rightsquigarrow \quad \xleftarrow{[\ ],\ stop} \qquad (3.2.6)$$

Proof. By corollary 2.3.16, the cases discussed above are exhausting. □

An analysis of the reduction rules shows, that the diagrams for an overlap of (cpa) with (lapp) and (lbeta) are very similar to the ones with (lseq) and (eseq)

respectively. For a better understanding, assume that the operator τ stands for @, i.e., application, or `seq`. Then the respective left hand side of the rules (lapp), (lseq) and (lbeta), (eseq) obey the format $\tau(\texttt{let } x = s_x \texttt{ in } s, t)$ and $\tau(\lambda x.s, t)$ respectively. Hence in the following we will not treat these analogue cases in detail again.

From this complete set of forking diagrams we wish to conclude that the convergent behaviour is preserved after an application of the (cpa)-rule. Therefore, we first have to examine the $\xrightarrow{\mathcal{C}, \; stop}$-reductions which are introduced in the diagram (3.2.5).

3.2.3   Internal (stop)-reductions

Since in the diagrams for the reduction rule (cpa) there occur (stop)-reductions which are to be carried out in arbitrary rather than surface contexts, we have to show that they are safe for an approximation reduction to reach an abstraction. More precisely, complete sets of commuting diagrams for such *internal* reductions will be established in this section.

Definition 3.2.9. *A* $\xrightarrow{C, \; stop}$*-reduction is called* internal*, depicted by* $\xrightarrow{i, \; C, \; stop}$*, if $C \in \mathcal{C}$ is a context which is not a surface context, i.e. $C \notin \mathcal{S}$.*

Lemma 3.2.10. *Below is a complete set of commuting diagrams for* $\xrightarrow{i, \; C, \; stop}$ *w.r.t. approximation reductions:*

$$\xrightarrow{i, \; C, \; stop} \cdot \xrightarrow{\mathcal{S}, \; a} \;\; \leadsto \;\; \xrightarrow{\mathcal{S}, \; a} \cdot \xrightarrow{i, \; C, \; stop}$$

$$\xrightarrow{i, \; C, \; stop} \cdot \xrightarrow{\mathcal{S}, \; a} \;\; \leadsto \;\; \xrightarrow{\mathcal{S}, \; a}$$

$$\xrightarrow{i, \; \texttt{let } x=\lambda y.C \texttt{ in } t, \; stop} \cdot \xrightarrow{[\;], \; cpa} \;\; \leadsto \;\; \xrightarrow{[\;], \; cpa} \cdot \xrightarrow{i, \; C, \; stop \; \#_x(t)}$$

$$\xrightarrow{i, \; S_1[(\lambda x.S_2)\, t], \; stop} \cdot \xrightarrow{S_1, \; lbeta} \;\; \leadsto \;\; \xrightarrow{S_1, \; lbeta} \cdot \xrightarrow{S_1[\texttt{let } x=t \texttt{ in } S_2], \; stop}$$

Proof. Note that — apart from the empty one — the contexts in which approximation and internal (stop)-reductions respectively take place, are by definition disjoint. Hence the first diagram is an easy application of lemma 2.3.14. Apart from the (cpa)-rule, it also covers approximation reductions that are performed inside the empty context.

An internal (stop)-reduction within a subexpression of a `seq` or `pick` may be eliminated as the second diagram shows. This happens when the mentioned subexpression is dropped by rule (eseq), (nd, left) or (nd, right).

The third diagram just treats the case, where the internal (stop)-reduction takes place inside the term to be copied by (cpa), which therefore has to be made up by as many internal (stop)-reductions as the variable x occurs in the target t of the copy operation. As the last diagram shows, the internal (stop)-reduction may also be turned into an ordinary one, since a (lbeta)-reduction can bring the corresponding context to the surface.   ◻

It is easily seen that the application of the above commuting diagrams for internal (stop)-reductions terminates. Consider the weight of a reduction sequence given by the multi-set containing the number of non-(stop) approximation reductions which follow each internal (stop)-reduction. Applying one of the commuting diagrams for internal (stop)-reductions strictly decreases this weight w.r.t. the usual multi-set ordering. Hence by induction on this measure of such a sequence, we may draw the conclusion that for every reduction sequence consisting of $\xrightarrow{\mathcal{S}}_{\boldsymbol{\lambda}_{\approx}}$- and $\xrightarrow{i,\ stop}$-reductions leading to an abstraction, there is also an approximation reduction sequence to an abstraction that only differs by internal (stop)-reductions.

Lemma 3.2.11. *Let $s, \lambda x.t \in \Lambda_{\approx}$ be terms with $s\ (\xrightarrow{\mathcal{S}}_{\boldsymbol{\lambda}_{\approx}} \cup \xrightarrow{i,\ stop})^{*}\ \lambda x.t$. Then there is also a reduction $s \xrightarrow{\mathcal{S}}^{*}_{\boldsymbol{\lambda}_{\approx}} \lambda x.t'$ such that $\lambda x.t' \xrightarrow{i,\ stop}^{*} \lambda x.t$ holds.*

Such mixed $(\xrightarrow{\mathcal{S}}_{\boldsymbol{\lambda}_{\approx}} \cup \xrightarrow{i,\ stop})$-reductions may evolve from the application of forking diagrams for (cpa) within some $\xrightarrow{\mathcal{S}}_{\boldsymbol{\lambda}_{\approx}}$-reduction sequence. Therefore, the previous lemma is necessary to show that a (cpa)-reduction preserves the convergent behaviour of terms up to internal (stop)-reductions.

Lemma 3.2.12. *Let s, t be terms such that $s \xrightarrow{\mathcal{S},\ cpa} t$. Then whenever s has an approximation reduction to an abstraction $\lambda x.s'$, there is also an approximation reduction starting from t leading to an abstraction $\lambda x.t'$ such that $\lambda x.s'$ differs from $\lambda x.t'$ only by internal (stop)-reductions, i.e. $\lambda x.t' \xrightarrow{i,\ stop}^{*} \lambda x.s'$ holds.*

Proof. Assuming $s \xrightarrow{\mathcal{S}}^{k}_{\boldsymbol{\lambda}_{\approx}} \lambda x.s'$ we show

$$\exists \lambda x.t' :\ t \xrightarrow{\mathcal{S}}^{*}_{\boldsymbol{\lambda}_{\approx}} \lambda x.t' \ \wedge\ \lambda x.t' \xrightarrow{i,\ stop}^{*} \lambda x.s'$$

by an induction on the length k of the reduction sequence.

- If $k = 1$ then according to remark 3.2.7, only (eseq), (nd) or (cpa) are possible within the empty context. In these cases we obviously have

$t \xrightarrow{\mathcal{S}}{}^{0\vee1}_{\pmb{\lambda}_{\approx}} \lambda x.s'$ by the two diagrams (3.2.2) and (3.2.4) of lemma 3.2.8 that come into question.

- For the induction step assume the claim to be valid for sequences of length smaller than k. Then the sequence $s \xrightarrow{\mathcal{S}}{}^{k}_{\pmb{\lambda}_{\approx}} \lambda x.s'$ may be divided as follows:

$$s \xrightarrow{\mathcal{S}, a}_{\pmb{\lambda}_{\approx}} s_1 \xrightarrow{\mathcal{S}}{}^{k-1}_{\pmb{\lambda}_{\approx}} \lambda x.s'$$

Therefore, $s_1 \xleftarrow{\mathcal{S}, a}_{\pmb{\lambda}_{\approx}} s \xrightarrow{\mathcal{S}, \text{cpa}} t$ represents a peak. Since s_1 has a converging approximation reduction, at least one forking diagram from the complete set of lemma 3.2.8 must be applicable.

From (3.2.2) a term t_1 satisfying $s_1 \xrightarrow{\mathcal{S}, \text{cpa}} t_1 \xleftarrow{\mathcal{S}, a}_{\pmb{\lambda}_{\approx}} t$ is obtained. By the induction hypothesis, there is a converging approximation reduction sequence $t_1 \xrightarrow{\mathcal{S}}{}^{*}_{\pmb{\lambda}_{\approx}} \lambda x.t'$ for which $\lambda x.t' \xrightarrow{i, \text{stop}}{}^{*} \lambda x.s'$ is valid. Therefore with $t \xrightarrow{\mathcal{S}, a}_{\pmb{\lambda}_{\approx}} t_1$ the proposition is shown.

For the diagram in (3.2.3) the situation is $s_1 \xrightarrow{\mathcal{S}, \text{cpa}} t$ and thus slightly simpler. Here, the induction hypothesis provides an approximation reduction sequence $t \xrightarrow{\mathcal{S}}{}^{*}_{\pmb{\lambda}_{\approx}} \lambda x.t'$ such that $\lambda x.t' \xrightarrow{i, \text{stop}}{}^{*} \lambda x.s'$ holds.

In the case of diagram (3.2.4) it is not necessary to use the induction hypothesis, since $t \xrightarrow{\mathcal{S}, \text{nd}} t_1 \equiv s_1 \xrightarrow{\mathcal{S}}{}^{k-1}_{\pmb{\lambda}_{\approx}} \lambda x.s'$ directly.

An application of the diagram (3.2.5) may cause internal (stop)-reductions, i.e. $s_1 \xrightarrow{\mathcal{S}, \text{cpa}} t_1 \xleftarrow{\mathcal{C}, \text{stop}}{}^{*} t$ ensues. Again, from $s_1 \xrightarrow{\mathcal{S}, \text{cpa}} t_1$ we have, by the induction hypothesis, an approximation reduction $t_1 \xrightarrow{\mathcal{S}}{}^{*}_{\pmb{\lambda}_{\approx}} \lambda x.t'$ that meets the condition $\lambda x.t' \xrightarrow{i, \text{stop}}{}^{*} \lambda x.s'$. Now, the whole reduction sequence is of the form $t \xrightarrow{\mathcal{C}, \text{stop}}{}^{*} t_1 \xrightarrow{\mathcal{S}}{}^{*}_{\pmb{\lambda}_{\approx}} \lambda x.t' \xrightarrow{i, \text{stop}}{}^{*} \lambda x.s'$ and lemma 3.2.11 proves the claim.

The last diagram (3.2.6) is not applicable here, because otherwise s_1 would reduce to the term $\odot$ instead of an abstraction. $\qquad\blacksquare$

Note that the induction argument in the above lemma is valid because of the special structure of the forking diagrams for (cpa), i.e. there is at most one (cpa)-reduction necessary to clean up the forking situation.

3.2.4 The (lbeta)-reduction

For reductions by rule (lbeta) we will devise a complete set of forking diagrams w.r.t. $\xrightarrow{\mathcal{S}}_{\lambda_{\approx}}$-reductions in this section.

If (lbeta) is applied in the empty context, the whole expression must be of the form $(\lambda x.t)\, s$. Hence a surface reduction may take place inside s or within the context $[\,]\, s$ only. For the latter possibility solely rule (stop) is applicable:

$$\xleftarrow{[\,]\, s,\ \text{stop}}\ .\ \xrightarrow{[\,],\ \text{lbeta}}\ \rightsquigarrow\ \xrightarrow{[\,],\ \text{stop}}\ .\ \xleftarrow{[\,],\ \text{stop}}$$

In the former case, any approximation reduction may obviously be commuted with a (lbeta)-reduction; thus the diagram below.

$$\xleftarrow{(\lambda x.t)\, S,\ a}\ .\ \xrightarrow{[\,],\ \text{lbeta}}\ \rightsquigarrow\ \xrightarrow{[\,],\ \text{lbeta}}\ .\ \xleftarrow{\texttt{let}\ x{=}S\ \texttt{in}\ t,\ a}$$

If rule (lbeta) is applied inside a non-empty surface context $S \in \mathcal{S}$ and (lapp) is top-level, the following three diagrams arise. The subsequent diagrams for a top-level (lseq) are analogous.

$$\xleftarrow{[\,],\ \text{lapp}}\ .\ \xrightarrow{(\texttt{let}\ x{=}s\ \texttt{in}\ t)\, S,\ \text{lbeta}}\ \rightsquigarrow\ \xrightarrow{\texttt{let}\ x{=}s\ \texttt{in}\ (t\, S),\ \text{lbeta}}\ .\ \xleftarrow{[\,],\ \text{lapp}}$$

$$\xleftarrow{[\,],\ \text{lapp}}\ .\ \xrightarrow{(\texttt{let}\ x{=}s\ \texttt{in}\ S)\, t,\ \text{lbeta}}\ \rightsquigarrow\ \xrightarrow{\texttt{let}\ x{=}s\ \texttt{in}\ (S\, t),\ \text{lbeta}}\ .\ \xleftarrow{[\,],\ \text{lapp}}$$

$$\xleftarrow{[\,],\ \text{lapp}}\ .\ \xrightarrow{(\texttt{let}\ x{=}S\ \texttt{in}\ s)\, t,\ \text{lbeta}}\ \rightsquigarrow\ \xrightarrow{\texttt{let}\ x{=}S\ \texttt{in}\ (s\, t),\ \text{lbeta}}\ .\ \xleftarrow{[\,],\ \text{lapp}}$$

$$\xleftarrow{[\,],\ \text{lseq}}\ .\ \xrightarrow{(\texttt{let}\ x{=}s\ \texttt{in}\ t)\ \texttt{seq}\ S,\ \text{lbeta}}\ \rightsquigarrow\ \xrightarrow{\texttt{let}\ x{=}s\ \texttt{in}\ (t\ \texttt{seq}\ S),\ \text{lbeta}}\ .\ \xleftarrow{[\,],\ \text{lseq}}$$

$$\xleftarrow{[\,],\ \text{lseq}}\ .\ \xrightarrow{(\texttt{let}\ x{=}s\ \texttt{in}\ S)\ \texttt{seq}\ t,\ \text{lbeta}}\ \rightsquigarrow\ \xrightarrow{\texttt{let}\ x{=}s\ \texttt{in}\ (S\ \texttt{seq}\ t),\ \text{lbeta}}\ .\ \xleftarrow{[\,],\ \text{lseq}}$$

$$\xleftarrow{[\,],\ \text{lseq}}\ .\ \xrightarrow{(\texttt{let}\ x{=}S\ \texttt{in}\ s)\ \texttt{seq}\ t,\ \text{lbeta}}\ \rightsquigarrow\ \xrightarrow{\texttt{let}\ x{=}S\ \texttt{in}\ (s\ \texttt{seq}\ t),\ \text{lbeta}}\ .\ \xleftarrow{[\,],\ \text{lseq}}$$

The next diagram covers the situation of (stop) being applied in the empty context eliminating the (lbeta)-reduction while the following handles the function argument of an (lbeta)-redex being canceled by (stop).

$$\xleftarrow{[\,],\ \text{stop}}\ .\ \xrightarrow{\mathcal{S},\ \text{lbeta}}\ \rightsquigarrow\ \xleftarrow{[\,],\ \text{stop}}$$

$$\xleftarrow{S[[\,]\, s],\ \text{stop}}\ .\ \xrightarrow{S,\ \text{lbeta}}\ \rightsquigarrow\ \xrightarrow{S,\ \text{stop}}\ .\ \xleftarrow{S,\ \text{stop}}$$

For rule (eseq) we obtain a single diagram

$$\xleftarrow{[\,],\ \text{eseq}}\ .\ \xrightarrow{(\lambda x.t)\ \texttt{seq}\ S,\ \text{lbeta}}\ \rightsquigarrow\ \xrightarrow{S,\ \text{lbeta}}\ .\ \xleftarrow{[\,],\ \text{eseq}}$$

If the non-deterministic rules (nd) are applied at top-level, the (lbeta)-reduction may be eliminated as the following diagrams illustrate.

$$\xleftarrow{[\,], \text{ nd, left}} \cdot \xrightarrow{\texttt{pick } S \ e, \text{ lbeta}} \quad \rightsquigarrow \quad \xrightarrow{S, \text{ lbeta}} \cdot \xleftarrow{[\,], \text{ nd, left}}$$

$$\xleftarrow{[\,], \text{ nd, right}} \cdot \xrightarrow{\texttt{pick } S \ e, \text{ lbeta}} \quad \rightsquigarrow \quad \xleftarrow{[\,], \text{ nd, right}}$$

$$\xleftarrow{[\,], \text{ nd, right}} \cdot \xrightarrow{\texttt{pick } e \ S, \text{ lbeta}} \quad \rightsquigarrow \quad \xrightarrow{S, \text{ lbeta}} \cdot \xleftarrow{[\,], \text{ nd, right}}$$

$$\xleftarrow{[\,], \text{ nd, left}} \cdot \xrightarrow{\texttt{pick } e \ S, \text{ lbeta}} \quad \rightsquigarrow \quad \xleftarrow{[\,], \text{ nd, left}}$$

Overlapping (lbeta) with itself modifies only the surface context where the reduction takes place.

$$\xleftarrow{(\lambda x.t)\, S, \text{ lbeta}} \cdot \xrightarrow{[\,], \text{ lbeta}} \quad \rightsquigarrow \quad \xrightarrow{[\,], \text{ lbeta}} \cdot \xleftarrow{\texttt{let } x=S \texttt{ in } t, \text{ lbeta}}$$

$$\xleftarrow{[\,], \text{ lbeta}} \cdot \xrightarrow{(\lambda x.t)\, S, \text{ lbeta}} \quad \rightsquigarrow \quad \xrightarrow{\texttt{let } x=S \texttt{ in } t, \text{ lbeta}} \cdot \xleftarrow{[\,], \text{ lbeta}}$$

So let us consider the remaining case, where rule (cpa) is applied in the empty context, i.e. the term has to be of the form $\texttt{let } x = \lambda z.q \texttt{ in } t$ or $\texttt{let } x = \circledcirc \texttt{ in } t$ and hence a surface reduction is possible inside t only.

$$\xleftarrow{[\,], \text{ cpa}} \cdot \xrightarrow{\texttt{let } x=\lambda z.q \texttt{ in } S, \text{ lbeta}} \quad \rightsquigarrow \quad \xrightarrow{S, \text{ lbeta}} \cdot \xleftarrow{[\,], \text{ cpa}}$$

$$\xleftarrow{[\,], \text{ cpa}} \cdot \xrightarrow{\texttt{let } x=\circledcirc \texttt{ in } S, \text{ lbeta}} \quad \rightsquigarrow \quad \xrightarrow{S, \text{ lbeta}} \cdot \xleftarrow{[\,], \text{ cpa}}$$

Referring to corollary 2.3.16 we collect these diagrams in a separate lemma.

Lemma 3.2.13. *The following is a complete set of forking diagrams for* $\xrightarrow{S, \text{ lbeta}}$ *w.r.t.* $\xrightarrow{S}_{\lambda_{\approx}}$ *-reductions:*

$$\xleftarrow{S, \ a} \cdot \xrightarrow{S, \ lbeta} \quad \rightsquigarrow \quad \xrightarrow{S, \ lbeta} \cdot \xleftarrow{S, \ a}$$

$$\xleftarrow{S, \ nd} \cdot \xrightarrow{S, \ lbeta} \quad \rightsquigarrow \quad \xleftarrow{S, \ nd}$$

$$\xleftarrow{[\,], \ stop} \cdot \xrightarrow{S, \ lbeta} \quad \rightsquigarrow \quad \xleftarrow{[\,], \ stop}$$

$$\xleftarrow{S[[\,]\,s], \ stop} \cdot \xrightarrow{S, \ lbeta} \quad \rightsquigarrow \quad \xrightarrow{S, \ stop} \cdot \xleftarrow{S, \ stop}$$

Before we can show that also reduction by rule (lbeta) preserves the approximation reduction to an abstraction, we need the following result about the correspondence of $\circledcirc$ and its application within surface contexts.

Lemma 3.2.14. *Let $t \in \Lambda_\approx$ be a term and $S \in \mathcal{S}$ a surface context such that $S[\circledcirc\, t] \xrightarrow{\mathcal{S}}{}^{*}_{\boldsymbol{\lambda}_\approx} \lambda z.q$. Then also $S[\circledcirc] \xrightarrow{\mathcal{S}}{}^{*}_{\boldsymbol{\lambda}_\approx} \lambda z.q$ holds.*

Proof. W.l.o.g. we may require S to be non-empty, i.e. $S \not\equiv [\ ]$, because $\circledcirc\, t$ obviously has no approximation reduction to an abstraction: The rules (lapp) and lbeta are the only possibilities to get rid of the top-level @-operator.

Now assume $S[\circledcirc\, t] \xrightarrow{\mathcal{S}}{}^{k}_{\boldsymbol{\lambda}_\approx} \lambda z.q$ for an induction on the length k of the reduction sequence.

- In the case $S[\circledcirc\, t] \xrightarrow{\mathcal{S}}_{\boldsymbol{\lambda}_\approx} \lambda z.q$ for $k = 1$ it is easily checked that only the rules (cpa) or (nd) could have been used to produce an abstraction. Since the term $\circledcirc\, t$ in question is located in a surface context, i.e. not under an abstraction, it must be discarded by (nd), whereas (cpa) is not possible.

- For $S[\circledcirc\, t] \xrightarrow{\mathcal{S}}_{\boldsymbol{\lambda}_\approx} s \xrightarrow{\mathcal{S}}{}^{k}_{\boldsymbol{\lambda}_\approx} \lambda z.q$ the proposition will obviously be true, if the application $\circledcirc\, t$ is dropped by rule (nd) in the first reduction step. Otherwise, either $s \equiv S'[\circledcirc\, t]$, $s \equiv S[\circledcirc\, t']$ or $s \equiv S[\circledcirc]$ holds, for the latter of which the claim becomes trivial. On the other hand, to both $S'[\circledcirc\, t]$ and $S[\circledcirc\, t']$ the induction hypothesis is applicable, thus the claim holds. □

Now we can use an argument similar to, but in this case simpler than the one in lemma 3.2.12 to prove the following lemma.

Lemma 3.2.15. *Let s, t be terms such that $s \xrightarrow{\mathcal{S},\ lbeta} t$. Then whenever s has an approximation reduction to an abstraction there is also an approximation reduction sequence starting from t leading to the same abstraction, i.e.*

$$s \xrightarrow{\mathcal{S}}{}^{*}_{\boldsymbol{\lambda}_\approx} \lambda z.q \implies t \xrightarrow{\mathcal{S}}{}^{*}_{\boldsymbol{\lambda}_\approx} \lambda z.q$$

Proof. Assume $s \xrightarrow{\mathcal{S}}{}^{k}_{\boldsymbol{\lambda}_\approx} \lambda z.q$ for an induction on the length k of the sequence.

- If $k = 1$ then, according to remark 3.2.7, only (eseq), (nd) or (cpa) at top-level are possible. Hence $t \xrightarrow{\mathcal{S}}{}^{0 \vee 1}_{\boldsymbol{\lambda}_\approx} \lambda z.q$ is obvious from the first two diagrams of lemma 3.2.13.

- For the induction step assume the claim to be valid for sequences of length smaller than k. Then the sequence $s \xrightarrow{\mathcal{S}}{}^{k}_{\boldsymbol{\lambda}_\approx} \lambda z.q$ may be divided as follows:

$$s \xrightarrow{\mathcal{S}}_{\boldsymbol{\lambda}_\approx} s_1 \xrightarrow{\mathcal{S}}{}^{k-1}_{\boldsymbol{\lambda}_\approx} \lambda z.q$$

We may apply the induction hypothesis to $s_1 \xrightarrow[\boldsymbol{\lambda}_{\approx}]{\mathcal{S}} {}^{k-1} \lambda z.q$, i.e. if we have $s_1 \xrightarrow{\mathcal{S}, \text{ lbeta}} t_1$ then there is also a reduction $t_1 \xrightarrow[\boldsymbol{\lambda}_{\approx}]{\mathcal{S}} {}^{*} \lambda z.q$. This is only the case for the first forking diagram of lemma 3.2.13, so there is also a reduction $t \xrightarrow[\boldsymbol{\lambda}_{\approx}]{\mathcal{S}} {}^{*} t_1$, thus $t \xrightarrow[\boldsymbol{\lambda}_{\approx}]{\mathcal{S}} {}^{*} \lambda z.q$.

In the case of the second and third diagram, using the induction hypothesis is not necessary, since $t \xrightarrow{\mathcal{S}, \text{ nd}} t_1 \equiv s_1 \xrightarrow[\boldsymbol{\lambda}_{\approx}]{\mathcal{S}} {}^{k-1} \lambda z.q$ directly.

If the last diagram is applicable, the special treatment necessary is done in lemma 3.2.14, thus the claim holds. $\qquad\square$

Corollary 3.2.16. *Let s, t be terms such that $s \xrightarrow{\mathcal{S}, \text{ lbeta}} t$. Then for all abstractions $\lambda z.q$ the following holds:*

$$s \xrightarrow[\boldsymbol{\lambda}_{\approx}]{\mathcal{S}} {}^{*} \lambda z.q \iff t \xrightarrow[\boldsymbol{\lambda}_{\approx}]{\mathcal{S}} {}^{*} \lambda z.q$$

Proof. The "only-if"-part is just the statement of the previous lemma where the "if"-part results from the fact that $s \xrightarrow{\mathcal{S}, \text{ lbeta}} t$. $\qquad\square$

In a similar way, complete sets of forking diagrams for the remaining deterministic rules (lapp), (eseq) and (lseq) could be developed.

3.2.5 Rearrangement of Reduction Sequences

The last sections have illustrated that deterministic reductions, rules (cpa) and (lbeta) in particular, do not do any harm to the converging behaviour of approximation reduction sequences. Therefore, complete sets of forking diagrams have been used in section 3.2.2 and 3.2.4, while section 3.2.3 introduced complete sets of commuting diagrams for $\xrightarrow{i, \text{ stop}}$-reductions.

In this section we will now devise various opportunities for altering the order of reductions in converging sequences. We opt for *not* speaking of a *reduction strategy*, although some kind of guidance is offered which reduction rules may be applied when. The reason is that in general a reduction strategy should determine a way to obtain a certain result, e.g., convergence. But none of the rearrangements of approximation reduction sequences in this section can provide this, since *convergence is required*. The difficulty arises mainly from the uncertainty, when the rule (stop) should be applied. Therefore, in section 4.4 this topic will briefly be discussed again.

However, the results of this section are of high relevance for the rest of the work. E.g., if a closed term at a surface position is needed then its reduction to an abstraction may take place first, by which syntactic continuity for the contextual preorder is achieved. Furthermore, there is the ability to restrict approximation reductions to surface contexts, which do not bind free variables. This is of great help, when showing the precongruence candidate stable under reduction. Eventually, a normalisation theorem will be established, i.e., converging approximation reduction sequences are normalised insofar that reductions within the binding of the outermost `let` are performed first, without changing the overall number of approximation reductions. Ubiquitous applications of the normalisation theorem underpin its significance.

Simple Commuting Cases

We start with some simple cases in which surface reductions may be commuted.

Corollary 3.2.17. *Let $s, t \in \Lambda_\approx$ be terms and a, b two arbitrary reduction types. Then for all surface contexts $S_a, S_b \in \mathcal{S}$ the following diagrams commute.*

$$
\begin{array}{ccc}
s\,t & \xrightarrow{\;S_b\,t,\;b\;} & s'\,t \\[2pt]
{\scriptstyle s\,S_a,\;a}\Big\downarrow & & \Big\downarrow{\scriptstyle s'\,S_a,\;a} \\[2pt]
s\,t' & \xrightarrow[\;S_b\,t',\;b\;]{} & s'\,t'
\end{array}
$$

$$
\begin{array}{ccc}
\texttt{let } x = s \texttt{ in } t & \xrightarrow{\;\texttt{let } x=S_b \texttt{ in } t,\;b\;} & \texttt{let } x = s' \texttt{ in } t \\[2pt]
{\scriptstyle \texttt{let } x=s \texttt{ in } S_a,\;a}\Big\downarrow & & \Big\downarrow{\scriptstyle \texttt{let } x=s' \texttt{ in } S_a,\;a} \\[2pt]
\texttt{let } x = s \texttt{ in } t' & \xrightarrow[\;\texttt{let } x=S_b \texttt{ in } t',\;b\;]{} & \texttt{let } x = s' \texttt{ in } t'
\end{array}
$$

$$
\begin{array}{ccc}
s \texttt{ seq } t & \xrightarrow{\;S_b \texttt{ seq } t,\;b\;} & s' \texttt{ seq } t \\[2pt]
{\scriptstyle s \texttt{ seq } S_a,\;a}\Big\downarrow & & \Big\downarrow{\scriptstyle s' \texttt{ seq } S_a,\;a} \\[2pt]
s \texttt{ seq } t' & \xrightarrow[\;S_b \texttt{ seq } t',\;b\;]{} & s' \texttt{ seq } t'
\end{array}
$$

Proof. By a simple application of lemma 2.3.14.   ☐

Corollary 3.2.18. *Let* $S \in \mathcal{S}$ *be a surface context and* $r, s, t \in \Lambda_{\approx}$ *be terms. Then for all types* (*a*) *of reductions the diagrams below commute.*

$$
\begin{array}{ccc}
S[(\lambda x.s)\, t] & \xrightarrow{\ S,\ lbeta\ } & S[\mathtt{let}\ x = t\ \mathtt{in}\ s] \\
{\scriptstyle S[(\lambda x.s)\,[\ ]],\ a}\Big\downarrow & & \Big\downarrow{\scriptstyle S[\mathtt{let}\ x=[\ ]\ \mathtt{in}\ s],\ a} \\
S[(\lambda x.s)\, t'] & \xrightarrow{\ S,\ lbeta\ } & S[\mathtt{let}\ x = t'\ \mathtt{in}\ s]
\end{array}
$$

$$
\begin{array}{ccc}
S[(\lambda x.s)\ \mathtt{seq}\ t] & \xrightarrow{\ S,\ eseq\ } & S[t] \\
{\scriptstyle S[(\lambda x.s)\ \mathtt{seq}\ [\ ]],\ a}\Big\downarrow & & \Big\downarrow{\scriptstyle S,\ a} \\
S[(\lambda x.s)\ \mathtt{seq}\ t'] & \xrightarrow{\ S,\ eseq\ } & S[t']
\end{array}
$$

Proof. By lemma 2.3.14, since the respective surface contexts are disjoint. □

Corollary 3.2.19. *Let* $(\tau, b) \in \{(@, lapp), (\mathtt{seq}, lseq)\}$, $S \in \mathcal{S}$ *be a surface context and* $r, s, t \in \Lambda_{\approx}$ *be terms. Then for all types* (*a*) *of reductions the following diagrams commute.*

$$
\begin{array}{ccc}
S[\tau((\mathtt{let}\ x = r\ \mathtt{in}\ s), t)] & \xrightarrow{\ S,\ b\ } & S[\mathtt{let}\ x = r\ \mathtt{in}\ (\tau(s,t))] \\
{\scriptstyle S[\tau((\mathtt{let}\ x=[\ ]\ \mathtt{in}\ s), t)],\ a}\Big\downarrow & & \Big\downarrow{\scriptstyle S[\mathtt{let}\ x=[\ ]\ \mathtt{in}\ \tau(s,t)],\ a} \\
S[\tau((\mathtt{let}\ x = r'\ \mathtt{in}\ s), t)] & \xrightarrow{\ S,\ b\ } & S[\mathtt{let}\ x = r'\ \mathtt{in}\ (\tau(s,t))]
\end{array}
$$

$$
\begin{array}{ccc}
S[\tau((\mathtt{let}\ x = r\ \mathtt{in}\ s), t)] & \xrightarrow{\ S,\ b\ } & S[\mathtt{let}\ x = r\ \mathtt{in}\ (\tau(s,t))] \\
{\scriptstyle S[\tau((\mathtt{let}\ x=r\ \mathtt{in}\ [\ ]), t)],\ a}\Big\downarrow & & \Big\downarrow{\scriptstyle S[\mathtt{let}\ x=r\ \mathtt{in}\ \tau([\ ],t)],\ a} \\
S[\tau((\mathtt{let}\ x = r\ \mathtt{in}\ s'), t')] & \xrightarrow{\ S,\ b\ } & S[\mathtt{let}\ x = r\ \mathtt{in}\ (\tau(s',t'))]
\end{array}
$$

$$
\begin{array}{ccc}
S[\tau((\mathtt{let}\ x = r\ \mathtt{in}\ s), t)] & \xrightarrow{\ S,\ b\ } & S[\mathtt{let}\ x = r\ \mathtt{in}\ (\tau(s,t))] \\
{\scriptstyle S[\tau((\mathtt{let}\ x=r\ \mathtt{in}\ s), [\ ])],\ a}\Big\downarrow & & \Big\downarrow{\scriptstyle S[\mathtt{let}\ x=r\ \mathtt{in}\ \tau(s,[\ ])],\ a} \\
S[\tau((\mathtt{let}\ x = r\ \mathtt{in}\ s), t')] & \xrightarrow{\ S,\ b\ } & S[\mathtt{let}\ x = r\ \mathtt{in}\ (\tau(s,t'))]
\end{array}
$$

Proof. As every argument of τ is in a surface position, lemma 2.3.14 applies. □

Evaluation at Strict Positions

It is now possible to show that every closed subterm may be evaluated in advance, whenever it is necessary for an approximation reduction of its enclosing superterm to reach an abstraction. Note that the evaluation at such a *strict position* does not alter the overall outcome of the approximation reduction sequence, i.e., the abstraction it ends with.

Lemma 3.2.20. *Let $S \in \mathcal{S}$ be a surface context, $\lambda z.q \in \Lambda_\approx$ an abstraction and $t \in \Lambda_\approx^0$ a closed term such that $S[t] \Downarrow \lambda z.q$ holds. Then either there is a closed abstraction $\lambda x.s \in \Lambda_\approx^0$ with $t \Downarrow \lambda x.s$ and $S[\lambda x.s] \Downarrow \lambda z.q$, or $S[\odot]$ converges.*

Proof. Since for $S \equiv [\,]$ or $t \equiv \lambda z.q$ there is nothing to show, we rule out these cases. The proof is then by induction on the length k of an approximation reduction sequence $S[t] \xrightarrow{\mathcal{S}}{}_{\boldsymbol{\lambda}_\approx}^{k} \lambda z.q$ to an abstraction. Since S cannot be the empty context $k = 0$ is not possible and $k = 1$ forms the induction base. So if $S[t] \xrightarrow{\mathcal{S}}{}_{\boldsymbol{\lambda}_\approx} \lambda z.q$ then also $S[\odot] \Downarrow$ as t is closed and not an abstraction.

Hence for the induction step we may assume that the claim holds for all sequences of length $k \geq 1$. Then for the first reduction $S[t] \xrightarrow{\mathcal{S}}{}_{\boldsymbol{\lambda}_\approx} r$ of an approximation sequence $S[t] \xrightarrow{\mathcal{S}}{}_{\boldsymbol{\lambda}_\approx} r \xrightarrow{\mathcal{S}}{}_{\boldsymbol{\lambda}_\approx}^{k} \lambda z.q$ there are the following cases:

- For $r \equiv S[t']$, i.e. the reduction took place within t, the proposition is shown by the induction hypothesis for the remaining sequence.

- If the reduction is performed on the context S, so that the hole is deleted by rule (nd) or (stop), then obviously $S[\odot] \xrightarrow{\mathcal{S}}{}_{\boldsymbol{\lambda}_\approx} r$, thus $S[\odot]$ will converge.

- For any other reduction by rule (lapp), (lbeta), (lseq), (eseq) and (cpa), or if the hole is not deleted by rule (nd), $r \equiv S'[t]$ results. For (cpa) this is only valid because of the precondition, i.e., that t is closed.

  Therefore, the proposition is shown by one of the commuting diagrams from corollary 3.2.17, 3.2.18 or 3.2.19 in combination with the induction hypothesis. E.g., for $S[t] \equiv S'[(\lambda y.p)\,t] \xrightarrow{S',\ \text{lbeta}} S'[\texttt{let } y = t \texttt{ in } p]$ the term t remains in a surface context. Thus, to $S''[\,] \equiv S'[\texttt{let } y = [\,] \texttt{ in } p]$ the induction hypothesis may be applied. There are following possibilities:

  - In the case of $S''[\odot] \Downarrow$ we obviously have $S[\odot] \Downarrow$ too, because of the reduction $S[\odot] \equiv S'[(\lambda y.p)\,\odot] \xrightarrow{S',\ \text{lbeta}} S'[\texttt{let } y = \odot \texttt{ in } p] \equiv S''[\odot]$.

– If there exists some closed abstraction $\lambda x.s$ such that both $t \Downarrow \lambda x.s$ and $S''[\lambda x.s] \Downarrow \lambda z.q$ are satisfied, the claim will hold by the following approximation reduction sequence:

$$S[\lambda x.s] \equiv S'[(\lambda y.p)\,(\lambda x.s)] \xrightarrow{S',\ \text{lbeta}}$$
$$S'[\texttt{let } y = (\lambda x.s) \texttt{ in } p] \equiv S''[\lambda x.s] \Downarrow \lambda z.q \qquad \square$$

We now define the notion of a *strict position* for a surface context. Compared to [SSSS04] this may seem somewhat simplistic, but is sufficient for our purposes.

Definition 3.2.21. *Let $S \in \mathcal{S}$ be a surface context such that $S[\circledcirc] \Downarrow\!\!\!\!/$ is true. Then S is called* strict *and its hole is said to be at a* strict position.

Corollary 3.2.22. *Let $t \in \Lambda_\approx^0$ be a closed term and $S \in \mathcal{S}$ be a strict surface context. Then $S[t] \Downarrow \lambda z.q$ if and only if there is a closed abstraction $\lambda x.s$ such that $t \Downarrow \lambda x.s$ and $S[\lambda x.s] \Downarrow \lambda z.q$ hold.*

Proof. Since the reduction $t \xrightarrow{\ \mathcal{S}\ }{}^*_{\lambda_\approx} \lambda x.s$ may take place inside S, the "if"-part is clear. For the "only-if"-part, assume $S[t] \Downarrow \lambda z.q$. Then lemma 3.2.20 becomes applicable. Since S is strict, $S[\circledcirc] \Downarrow$ is not possible, thus the claim follows. $\square$

Non-closing Surface Contexts

Moreover, we can show that for every approximation reduction to an abstraction there is also an approximation reduction sequence to the same abstraction performing reductions on closed terms only.

Definition 3.2.23. *The class $\mathcal{N}$ of* non-closing surface contexts *is inductively defined by the following rule for the symbol N:*

$$\begin{aligned}
N ::= \ & [\,] \ | \ \texttt{let } x = N \texttt{ in } e \ | \\
& N\,e \ | \ e\,N \ | \\
& N \texttt{ seq } e \ | \ e \texttt{ seq } N \ | \\
& \texttt{pick } N\,e \ | \ \texttt{pick } e\,N
\end{aligned}$$

This means that non-closing surface contexts are just the subset of surface contexts whose construction does not involve any L_R-context. Thus a surface context $S \in \mathcal{S} \setminus \mathcal{N}$ is called *closing*, i.e., if it is not a non-closing surface context. We now show that reductions inside closing surface contexts can be moved to the end of an approximation reduction sequence.

Lemma 3.2.24. *For* $\xrightarrow{S \setminus \mathcal{N},\ a}_{\lambda_\approx}$ *- w.r.t.* $\xrightarrow{\mathcal{N},\ b}_{\lambda_\approx}$ *-reductions, the following constitutes a complete set of commuting diagrams.*

$$\xrightarrow{S \setminus \mathcal{N},\ a}_{\lambda_\approx} \cdot \xrightarrow{\mathcal{N},\ b}_{\lambda_\approx} \ \rightsquigarrow\ \xrightarrow{\mathcal{N},\ b}_{\lambda_\approx} \cdot \xrightarrow{S \setminus \mathcal{N},\ a}_{\lambda_\approx}$$

$$\xrightarrow{S \setminus \mathcal{N},\ a}_{\lambda_\approx} \cdot \xrightarrow{\mathcal{N},\ b}_{\lambda_\approx} \ \rightsquigarrow\ \xrightarrow{\mathcal{N},\ b}_{\lambda_\approx}$$

Proof. First note that any overlapping closing surface context must be of the form $N[L_R[S]]$ for some surface context S and a non-closing N. Since the property (non-) closing is retained when inserting into a non-closing surface context, it suffices to examine sequences of the form $\xrightarrow{L_R[S],\ a}_{\lambda_\approx} \cdot \xrightarrow{N,\ b}_{\lambda_\approx}$ and $\xrightarrow{N[L_R[S]],\ a}_{\lambda_\approx} \cdot \xrightarrow{[\ ],\ b}_{\lambda_\approx}$. For the former we immediately obtain the diagram

$$\xrightarrow{L_R[S],\ a}_{\lambda_\approx} \cdot \xrightarrow{N,\ b}_{\lambda_\approx} \ \rightsquigarrow\ \xrightarrow{N,\ b}_{\lambda_\approx} \cdot \xrightarrow{L_R[S],\ a}_{\lambda_\approx}$$

since a L_R- is disjoint from every N-context. If $\xrightarrow{N[L_R[S]],\ a}_{\lambda_\approx} \cdot \xrightarrow{[\ ],\ b}_{\lambda_\approx}$ then from $b = \mathrm{stop}$ the diagram $\xrightarrow{N[L_R[S]],\ a}_{\lambda_\approx} \cdot \xrightarrow{[\ ],\ \mathrm{stop}} \ \rightsquigarrow\ \xrightarrow{[\ ],\ \mathrm{stop}}$ follows. For the remaining rules we must distinguish along N.

- For the empty context $N \equiv [\ ]$ we have $b = \mathrm{cpa}$ and in contrast to the forking diagrams in 3.2.2, a (stop)-reduction can only be performed inside the target of the (cpa)-reduction, because of the L_R-context:

$$\xrightarrow{L_R[S],\ a}_{\lambda_\approx} \cdot \xrightarrow{[\ ],\ \mathrm{cpa}} \ \rightsquigarrow\ \xrightarrow{[\ ],\ \mathrm{cpa}} \cdot \xrightarrow{S,\ a}_{\lambda_\approx}$$

- Since the only top-level reduction for a `let`-term could be by rule (cpa), the case $N \equiv \texttt{let } x = N' \texttt{ in } e$ is impossible.

- If $N \equiv N'\,e$ only $b = \mathrm{lapp}$ may be a possible top-level reduction, hence

$$\xrightarrow{(\texttt{let } x=N''[L_R[S]] \texttt{ in } f)\,e,\ a}_{\lambda_\approx} \cdot \xrightarrow{[\ ],\ \mathrm{lapp}} \ \rightsquigarrow$$

$$\xrightarrow{[\ ],\ \mathrm{lapp}} \cdot \xrightarrow{\texttt{let } x=N''[L_R[S]] \texttt{ in } (f\,e),\ a}_{\lambda_\approx}$$

- For $N \equiv e \, N'$ there is an overlap with $b \in \{\text{lapp}, \text{lbeta}\}$ which results in

$$\xrightarrow[\boldsymbol{\lambda}_{\approx}]{(\lambda x.s)\,(N'[L_R[S]]),\ a} \cdot \xrightarrow{[\ ],\ \text{lbeta}} \ \rightsquigarrow$$

$$\xrightarrow{[\ ],\ \text{lbeta}} \cdot \xrightarrow[\boldsymbol{\lambda}_{\approx}]{\texttt{let } x=(N'[L_R[S]]) \texttt{ in } s,\ a}$$

$$\xrightarrow[\boldsymbol{\lambda}_{\approx}]{(\texttt{let } x=s \texttt{ in } t)\,(N'[L_R[S]]),\ a} \cdot \xrightarrow{[\ ],\ \text{lapp}} \ \rightsquigarrow$$

$$\xrightarrow{[\ ],\ \text{lapp}} \cdot \xrightarrow[\boldsymbol{\lambda}_{\approx}]{\texttt{let } x=s \texttt{ in } (t\,(N'[L_R[S]])),\ a}$$

- If $N \equiv N' \, \texttt{seq} \, e$ only $b = \text{lseq}$ is possible at top-level, therefore

$$\xrightarrow[\boldsymbol{\lambda}_{\approx}]{(\texttt{let } x=N''[L_R[S]] \texttt{ in } d) \texttt{ seq } e,\ a} \cdot \xrightarrow{[\ ],\ \text{lseq}} \ \rightsquigarrow$$

$$\xrightarrow{[\ ],\ \text{lseq}} \cdot \xrightarrow[\boldsymbol{\lambda}_{\approx}]{\texttt{let } x=N''[L_R[S]] \texttt{ in } d \texttt{ seq } e,\ a}$$

- For $N \equiv e \, \texttt{seq} \, N'$ rule (eseq) or (lseq) may be applicable at top-level.

$$\xrightarrow[\boldsymbol{\lambda}_{\approx}]{(\texttt{let } x=t_x \texttt{ in } t) \texttt{ seq } N'[L_R[S]]} \cdot \xrightarrow{[\ ],\ \text{lseq}} \ \rightsquigarrow$$

$$\xrightarrow{[\ ],\ \text{lseq}} \cdot \xrightarrow[\boldsymbol{\lambda}_{\approx}]{\texttt{let } x=t_x \texttt{ in } (t \texttt{ seq } N'[L_R[S]])}$$

$$\xrightarrow[\boldsymbol{\lambda}_{\approx}]{(\lambda x.t) \texttt{ seq } N'[L_R[S]]} \cdot \xrightarrow{[\ ],\ \text{eseq}} \ \rightsquigarrow$$

$$\xrightarrow{[\ ],\ \text{lseq}} \cdot \xrightarrow[\boldsymbol{\lambda}_{\approx}]{N'[L_R[S]]}$$

- The cases $\texttt{pick } N' \, e$, $\texttt{pick } e \, N'$ imply that the top-level (b)-reduction is (nd), and the (a)-reduction may be dropped:

$$\xrightarrow[\boldsymbol{\lambda}_{\approx}]{N[L_R[S]]],\ a} \cdot \xrightarrow{[\ ],\ \text{nd}} \ \rightsquigarrow \ \xrightarrow{[\ ],\ \text{nd}} \cdot \xrightarrow[\boldsymbol{\lambda}_{\approx}]{N[L_R[S]],\ a}$$

$$\xrightarrow[\boldsymbol{\lambda}_{\approx}]{N[L_R[S]],\ a} \cdot \xrightarrow{[\ ],\ \text{nd}} \ \rightsquigarrow \ \xrightarrow{[\ ],\ \text{nd}} \qquad \qquad \square$$

The application of the above commuting diagrams obviously terminates, since the number of $\xrightarrow[\boldsymbol{\lambda}_{\approx}]{\mathcal{N},\ b}$-reductions following each $\xrightarrow[\boldsymbol{\lambda}_{\approx}]{\mathcal{S}\setminus\mathcal{N},\ a}$-reduction is strictly decreased in every approximation reduction sequence. So the diagrams enable to successively move every $\xrightarrow[\boldsymbol{\lambda}_{\approx}]{\mathcal{S}\setminus\mathcal{N},\ a}$-reduction to the end of a sequence.

Lemma 3.2.25. *Let $s, \lambda x.t \in \Lambda_\approx^0$ be closed terms such that $s \Downarrow \lambda x.t$ holds. Then there is an approximation reduction sequence of the form $s \xrightarrow{\mathcal{N}}{}^*_{\boldsymbol{\lambda}_\approx} \lambda x.t$ too, i.e. using only reductions in non-closing surface contexts.*

Proof. By induction on the length k of an approximation reduction $s \xrightarrow{\mathcal{S}}{}^k_{\boldsymbol{\lambda}_\approx} \lambda x.t$.

- If $k = 1$, nothing has to be shown, since by remark 3.2.7 only reductions within the empty context come into question.

- For the induction step, we split a reduction $s \xrightarrow{\mathcal{S}}{}^{k+1}_{\boldsymbol{\lambda}_\approx} \lambda x.t$ into the sequence $s \xrightarrow{\mathcal{S}}_{\boldsymbol{\lambda}_\approx} s' \xrightarrow{\mathcal{S}}{}^k_{\boldsymbol{\lambda}_\approx} \lambda x.t$ and assume the reduction $s \xrightarrow{\mathcal{S}}_{\boldsymbol{\lambda}_\approx} s'$ to take place in a closing surface context, since otherwise the claim immediately follows from an easy application of the induction hypothesis.

  So applying the induction hypothesis to the suffix sequence $s' \xrightarrow{\mathcal{S}}{}^k_{\boldsymbol{\lambda}_\approx} \lambda x.t$ delivers an approximation reduction $s' \xrightarrow{\mathcal{N}}{}^k_{\boldsymbol{\lambda}_\approx} \lambda x.t$. By the diagrams of lemma 3.2.24, the reduction $s \xrightarrow{\mathcal{S}}_{\boldsymbol{\lambda}_\approx} s'$ either commutes with this sequence or is superfluous. $\qquad\blacksquare$

Normalisation

From the above we have the capability to restrict approximation reductions to closed terms. Thereby, reductions within L_L-contexts, i.e. surface contexts of the form `let` $x = [\,]$ `in` e, occur as a special case. Now, every approximation reduction leading to an abstraction can be *normalised* in that reductions within the outermost L_L-context may take place first. I.e., the goal for a term of the form `let` $x = s$ `in` t is to first reduce inside s until $\odot$ or an abstraction is reached and then to collapse the `let`-expression using (cpa) before proceeding with reductions inside t.

This means, we have to bring all the reductions that take place inside s, i.e., reductions of type $\xrightarrow{\texttt{let } x=\mathcal{S} \texttt{ in } t, \ a}_{\boldsymbol{\lambda}_\approx}$, to the front. The following lemma uses the previous commutativity results to successively move these reductions forward.

Lemma 3.2.26. *For every reduction sequence* `let` $x = s$ `in` $t \xrightarrow{\mathcal{S}}{}^n_{\boldsymbol{\lambda}_\approx} \lambda z.q$ *with $n > 0$ there is also an approximation reduction sequence*

$$\texttt{let } x = s \texttt{ in } t \xrightarrow{\texttt{let } x=\mathcal{S} \texttt{ in } t}{}^k_{\boldsymbol{\lambda}_\approx} \texttt{let } x = s' \texttt{ in } t \xrightarrow{[\,], \ cpa} t[s'/x] \xrightarrow{\mathcal{S}}{}^m_{\boldsymbol{\lambda}_\approx} \lambda z.q$$

where s' represents $\odot$ or an abstraction and $k + 1 + m \leq n$ holds.

3.2. THE APPROXIMATION CALCULUS $\pmb{\lambda}_\approx$ 79

Proof. Assume `let` $x = s$ `in` $t \xrightarrow[\pmb{\lambda}_\approx]{S}{}^n \lambda z.q$. Then the proof is by induction on the length n of the reduction sequence. According to remark 3.2.7, for a term of the form `let` $x = s$ `in` t the only rule which may produce an abstraction in a single step is (cpa), hence the induction base is clear.

For the induction step, we may split up the above reduction sequence into `let` $x = s$ `in` $t \xrightarrow[\pmb{\lambda}_\approx]{S} r \xrightarrow[\pmb{\lambda}_\approx]{S}{}^{n-1} \lambda z.q$. Now either `let` $x = s$ `in` $t \xrightarrow[\pmb{\lambda}_\approx]{S} r$ already is of the desired form, i.e., takes place within s. Thus we apply the induction hypothesis to $r \xrightarrow[\pmb{\lambda}_\approx]{S}{}^n \lambda z.q$ which proves the claim.

Or `let` $x = s$ `in` $t \xrightarrow[\pmb{\lambda}_\approx]{\text{let } x=s \text{ in } S,\, a} $ `let` $x = s$ `in` $t' \equiv r$ is the first reduction and we have to apply the induction hypothesis twice. First, from `let` $x = s$ `in` $t' \xrightarrow[\pmb{\lambda}_\approx]{S}{}^n \lambda z.q$ we yield an approximation reduction sequence

$$\text{let } x = s \text{ in } t' \xrightarrow[\pmb{\lambda}_\approx]{\text{let } x=S \text{ in } t'}{}^k \text{let } x = s' \text{ in } t' \xrightarrow[]{[\,],\ \text{cpa}} t'[s'/x] \xrightarrow[\pmb{\lambda}_\approx]{S}{}^m \lambda z.q$$

with $k + 1 + m \leq n$ for which we distinguish the following alternatives:

- For $s' \equiv s$, if $k = 0$ is the case, s must be $\odot$ or an abstraction and hence the rule (cpa) may already be applied to `let` $x = s$ `in` t, which simply commutes with the $\xrightarrow[\pmb{\lambda}_\approx]{\text{let } x=s \text{ in } S,\, a}$-reduction.

- If `let` $x = s$ `in` $t' \xrightarrow[\pmb{\lambda}_\approx]{\text{let } x=S \text{ in } t'}{}^{k\geq 1} $ `let` $x = s'$ `in` t', i.e.

$$\text{let } x = s \text{ in } t' \xrightarrow[\pmb{\lambda}_\approx]{\text{let } x=S \text{ in } t'} \text{let } x = s'' \text{ in } t'$$
$$\xrightarrow[\pmb{\lambda}_\approx]{\text{let } x=S \text{ in } t'}{}^{k-1} \text{let } x = s' \text{ in } t'$$

then by corollary 3.2.17 we may commute the first of these reductions with the preceding `let` $x = s$ `in` $t \xrightarrow[\pmb{\lambda}_\approx]{\text{let } x=s \text{ in } S,\, a} $ `let` $x = s$ `in` t' hence

$$\text{let } x = s \text{ in } t \xrightarrow[\pmb{\lambda}_\approx]{\text{let } x=S \text{ in } t} \text{let } x = s'' \text{ in } t \xrightarrow[\pmb{\lambda}_\approx]{\text{let } x=s'' \text{ in } S}$$
$$\text{let } x = s'' \text{ in } t' \xrightarrow[\pmb{\lambda}_\approx]{\text{let } x=S \text{ in } t'}{}^{k-1}$$
$$\text{let } x = s' \text{ in } t' \xrightarrow[]{[\,],\ \text{cpa}} t'[s'/x] \xrightarrow[\pmb{\lambda}_\approx]{S}{}^m \lambda z.q$$

Now apply the induction hypothesis to the suffix of the above sequence, thus the claim holds.   □

Since there is no need to shift (cpa)-reductions over (stop)-reductions in this case, the argument is far less complex than in lemma 3.2.12, where (cpa) is shown to keep an approximation reduction to an abstraction.

Combining the reduction sequence $\xrightarrow{\texttt{let } x=\mathcal{S} \texttt{ in } t}{}^{*}_{\boldsymbol{\lambda}_{\approx}} \cdot \xrightarrow{[\ ],\ cpa}$ in one distinct reduction, the result of the previous lemma is easily generalised to arbitrarily deep environments.

Definition 3.2.27. *Let r stand for $\odot$ or an abstraction. Then the reduction rule* (olf) *is defined by*

$$\texttt{let } x = r \texttt{ in } s \xrightarrow{olf} t \hspace{3cm} \text{(olf)}$$

$$\textit{if } \texttt{let } x = r \texttt{ in } s \xrightarrow{\mathcal{S}}{}^{*}_{\boldsymbol{\lambda}_{\approx}} \texttt{let } x = r' \texttt{ in } s \xrightarrow{[\ ],\ cpa} s[r'/x] \equiv t$$

We then call an approximation reduction sequence of the form $L^{*}_{R}[t] \xrightarrow{olf}{}^{*} t'$ for some environment L^{*}_{R} an <u>o</u>uter <u>l</u>et <u>f</u>irst *sequence.*

The following fruitful theorem shows that such an outer let first sequence exists for every converging reduction.

Theorem 3.2.28 (Normalisation). *Let $t \in \Lambda_{\approx}$ be a term. Then for every approximation reduction $t \xrightarrow{\mathcal{S}}{}^{*}_{\boldsymbol{\lambda}_{\approx}} \lambda z.q$ to an abstraction there exists also an outer let first sequence $t \xrightarrow{olf}{}^{*} t' \xrightarrow{\mathcal{S}}{}^{*}_{\boldsymbol{\lambda}_{\approx}} \lambda z.q$ such that t' is not a* `let`*-term.*

Proof. Assume $t \equiv L^{*}_{R}[s]$ and use lemma 3.2.26 for an induction on L^{*}_{R}. $\qquad \blacksquare$

Obviously, this process can be continued recursively, i.e., a reduction sequence of the form $\texttt{let } x = s \texttt{ in } t \xrightarrow{\texttt{let } x=\mathcal{S} \texttt{ in } t}{}^{*}_{\boldsymbol{\lambda}_{\approx}} \texttt{let } x = s' \texttt{ in } t$ may again be normalised within the surface context $\texttt{let } x = \mathcal{S} \texttt{ in } t$, if s' is an abstraction.

Although this might look like some kind of reduction strategy, theorem 3.2.28 does *not* present a standardisation in the sense that for any converging term *every* (olf)-reduction sequence leads to an abstraction. The reason is that it is unknown in advance when and to which term the rule (stop) should be applied.

3.2.6   The evaluation of `seq`

This section shows that the second argument of a `seq`-term determines its possible outcomes — provided the first has a reduction to an abstraction.

Lemma 3.2.29. *Let* $s, t \in \Lambda_{\approx}$ *be terms. Then* s `seq` t *has an approximation reduction to some abstraction if and only if there is also a reduction sequence starting from* t *leading to the same abstraction and* s *converges as well, i.e.:*

$$s \ \texttt{seq} \ t \Downarrow \lambda z.q \iff t \Downarrow \lambda z.q \ \wedge \ s \Downarrow$$

Proof. The "if"-part is trivial. For the "only-if"-part, assume s `seq` $t \xrightarrow[\boldsymbol{\lambda}_{\approx}]{\mathcal{S}}{}^{n} \lambda z.q$ and use induction on the length n of this converging reduction sequence. If the first reduction of this sequence takes place within s or t, the induction hypothesis proves the claim. That's why only the following cases remain:

- If s is an abstraction and rule (eseq) is applied to the term s `seq` t we obtain the reduction sequence $t \xrightarrow[\boldsymbol{\lambda}_{\approx}]{\mathcal{S}}{}^{n-1} \lambda z.q$ and nothing has to be shown.

- If s is a `let`-expression, i.e. $s \equiv \texttt{let } x = p_x \texttt{ in } r$ and the rule (lseq) is applied to the term s `seq` t we have

$$(\texttt{let } x = p_x \texttt{ in } r) \ \texttt{seq} \ t \xrightarrow{\text{lseq}} \texttt{let } x = p_x \texttt{ in } (r \ \texttt{seq} \ t) \xrightarrow[\boldsymbol{\lambda}_{\approx}]{\mathcal{S}}{}^{n-1} \lambda z.q$$

By the standardisation of lemma 3.2.26 there is also an approximation reduction sequence of the form

$$\texttt{let } x = p_x \texttt{ in } (r \ \texttt{seq} \ t) \xrightarrow[\boldsymbol{\lambda}_{\approx}]{\texttt{let } x = [\,] \texttt{ in } (r \ \texttt{seq} \ t)}{}^{k} \texttt{let } x = p_x' \texttt{ in } (r \ \texttt{seq} \ t)$$
$$\xrightarrow{\text{cpa}} r[p_x'/x] \ \texttt{seq} \ t[p_x'/x] \xrightarrow[\boldsymbol{\lambda}_{\approx}]{\mathcal{S}}{}^{m} \lambda z.q$$

satisfying $k + 1 + m = n$. Hence the induction hypothesis may be applied to $r[p_x'/x]$ `seq` $t[p_x'/x]$, i.e. we have $r[p_x'/x] \Downarrow$ and $t[p_x'/x] \Downarrow \lambda z.q$ where $t[p_x'/x] \equiv t$ holds, since x does not occur free in t. So it remains to prove that s may converge. We therefore perform the reductions on p_x within the surface context `let` $x = [\,]$ `in` r yielding

$$\texttt{let } x = p_x \texttt{ in } r \xrightarrow[\boldsymbol{\lambda}_{\approx}]{\texttt{let } x = [\,] \texttt{ in } r}{}^{k} \texttt{let } x = p_x' \texttt{ in } r \xrightarrow{\text{cpa}} r[p_x'/x]$$

Thus, from $r[p_x'/x] \Downarrow$ the proposition follows. $\qquad\qquad\square$

3.3   Approximation of $\boldsymbol{\lambda}_{\mathrm{ND}}$-terms in $\boldsymbol{\lambda}_{\approx}$

In the preceding sections, two non-deterministic call-by-need lambda-calculi have been introduced. The original $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus possesses a normal-order reduction based on reduction contexts, whereas the approximation reductions of the calculus $\boldsymbol{\lambda}_{\approx}$ are permitted in every surface context.

Since a sensible definition of a simulation is only possible for the $\boldsymbol{\lambda}_{\approx}$-calculus, we have an obligation to show that this indeed forms a technique for proving contextual equivalences in the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus. Therefore, this section will establish the correspondence for the respective notions of convergence in both of these calculi. So the reader mainly interested in similarity and its precongruence proof may safely skip this section.

First, we define an approximation of $\boldsymbol{\lambda}_{\mathrm{ND}}$-terms by sets of $\boldsymbol{\lambda}_{\approx}$-terms so that its convergent behaviour shall be retained.

Definition 3.3.1 (Approximation Set). *Let $s \in \Lambda_{\mathrm{ND}}$ be a $\boldsymbol{\lambda}_{\mathrm{ND}}$-term. Then its* approximation set $\wr s \wr \subseteq \Lambda_{\approx}$ *is defined as the following set of $\boldsymbol{\lambda}_{\approx}$-terms:*

$$\wr s \wr = \{\lambda x.t \in \Lambda_{\approx} \mid s \xrightarrow{\mathcal{S}}{}^{*}_{\boldsymbol{\lambda}_{\approx}} \lambda x.t\}$$

The key is the Approximation Theorem below. Section 3.3.3 will put together all the parts of its proof.

Theorem 3.3.2 (Approximation Theorem). *For all terms $s \in \Lambda_{\mathrm{ND}}$ the following holds: $s \Downarrow_{\mathrm{ND}}$ if and only if its approximation set $\wr s \wr$ is non-empty.*

To simplify matters we will consider $\boldsymbol{\lambda}_{\approx}$-terms throughout the remainder of this chapter. We therefore understand the notion of normal-order reduction in Λ_{ND} as extended to terms from $\Lambda_{\approx}$ in the obvious way, i.e. regarding $\odot$ as a constant which has no normal-order reduction. So for the above theorem, the following two implications have to be shown.

1. If there is a normal order reduction for s ending in a WHNF, then there is also an approximation reduction from s to an abstraction.

2. If there is an approximation sequence starting with s and ending in an abstraction, there is also a normal order reduction from s to a WHNF.

The latter requires complete sets of commuting diagrams for $\xrightarrow{\mathcal{S}}_{\boldsymbol{\lambda}_{\approx}}$-reductions w.r.t. $\xrightarrow{n}_{\boldsymbol{\lambda}_{\mathrm{ND}}}$-reductions, while in order to prove the former implication, we will resort to complete sets of forking diagrams. The figures 3.5 and 3.6 illustrate
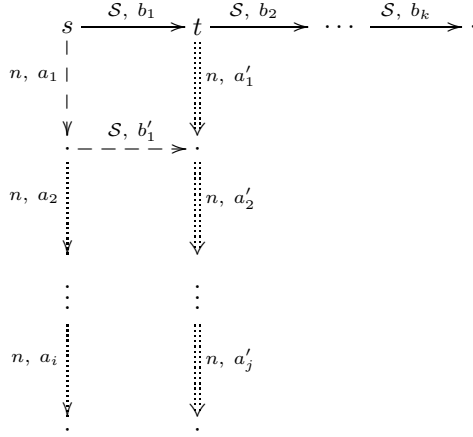
Figure 3.5: Transformation of a $\xrightarrow{\mathcal{S}}_{\boldsymbol{\lambda}_{\approx}}$- into a $\xrightarrow{n}_{\boldsymbol{\lambda}_{\mathrm{ND}}}$-reduction sequence
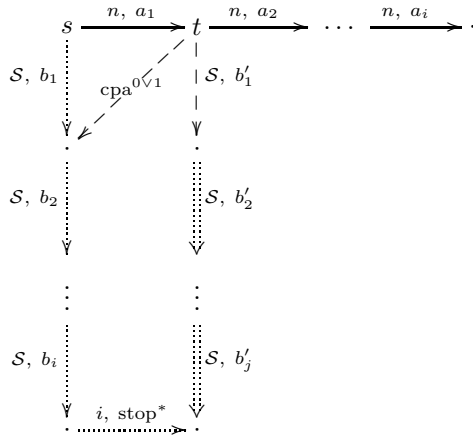
Figure 3.6: Transformation of a $\xrightarrow{n}_{\boldsymbol{\lambda}_{\mathrm{ND}}}$- into a $\xrightarrow{\mathcal{S}}_{\boldsymbol{\lambda}_{\approx}}$-reduction sequence

this fact as follows. First note, that the proof is by induction on the length of the respective existing reduction sequence. So when a converging approximation reduction for the term s is provided, the continuous line in figure 3.5, a converging normal order reduction for t, marked with the $\dashrightarrow$-styled arrows, may be assumed by the induction hypothesis. Thus, for the original term s, a reduction sequence of the form $\xrightarrow{\mathcal{S},\,b_1}_{\boldsymbol{\lambda}_\approx} \cdot \xrightarrow{n,\,a_1'} \cdot \xrightarrow{n}^{*}$ arises, that ends in a weak head normal form. Therefore, the leading approximation reduction has to be moved to the end of the whole sequence. This process can be accomplished inductively by a complete set of commuting diagrams for $\xrightarrow{\mathcal{S}}_{\boldsymbol{\lambda}_\approx}$- w.r.t. $\xrightarrow{n}$-reductions, as the dashed lines in figure 3.5 point out.

On the other hand consider the case where a converging normal order reduction sequence, the continuous line in figure 3.6, is given for a term s. Then the induction hypothesis yields a converging approximation sequence for t, whose tail is again marked with the $\dashrightarrow$-styled arrows. The only non-trivial cases for a_1 are (llet) and (cp), because every other normal-order reduction also forms an approximation reduction. Hence, a converging approximation reduction sequence for s, the dotted lines in figure 3.6, can be constructed in either of the following ways. Regarding (llet), proving the existence of a converging approximation reduction for s falls back on the normalisation of section 3.2.5. In the case of rule (cp), the complete set of forking diagrams for (cpa) from section 3.2.2 will be used to produce the desired approximation sequence. The dashed lines in figure 3.6 indicate the forking situation from which, by lemma 3.2.12, internal stop-reductions may emerge.

So why do not utilise the technique of commuting diagrams again, as for the transformation of converging approximation into normal-order reduction sequences before? The reason is simply that there is no complete set of commuting diagrams for $\xrightarrow{n}_{\boldsymbol{\lambda}_{ND}}$- w.r.t. $\xrightarrow{\mathcal{S}}_{\boldsymbol{\lambda}_\approx}$-reductions. While diagrams may be constructed for $\xrightarrow{n,\,cp} \cdot \xrightarrow{\mathcal{S},\,cpa}$ they are not possible for $\xrightarrow{n,\,llet} \cdot \xrightarrow{\mathcal{S},\,cpa}$ as the following example shows. Therein, let t_x stand for $\odot$ or an abstraction.

$$\texttt{let } x = (\texttt{let } y = t_y \texttt{ in } t_x) \texttt{ in } R[x] \xrightarrow{n,\;llet} \texttt{let } y = t_y \texttt{ in } (\texttt{let } x = t_x \texttt{ in } R[x])$$

$$\xrightarrow{\texttt{let } y=t_y \texttt{ in } [\;],\; cpa} \texttt{let } y = t_y \texttt{ in } \big(R[x]\big[t_x/x\big]\big)$$

Obviously, the same effect cannot be achieved from approximation reductions followed by (llet) or any other normal-order reduction.

3.3.1 Transforming $\xrightarrow{\;\mathcal{S}\;}_{\boldsymbol{\lambda}_{\approx}}$- into $\xrightarrow{\;n\;}_{\boldsymbol{\lambda}_{\mathrm{ND}}}$-reduction sequences

In this section we will prove the second implication of the Approximation The-orem, i.e. the transformation of a $\xrightarrow{\;\mathcal{S}\;}_{\boldsymbol{\lambda}_{\approx}}$- into a $\xrightarrow{\;n\;}$-reduction sequence, while retaining convergence. As noted above, complete sets of commuting diagrams for $\xrightarrow{\;\mathcal{S}\;}_{\boldsymbol{\lambda}_{\approx}}$- w.r.t. $\xrightarrow{\;n\;}$-reductions will therefore be established. We will then show that for every reduction sequence

$$s_0 \xrightarrow{\;\mathcal{S}\,,\,a\;}_{\boldsymbol{\lambda}_{\approx}} s_1 \xrightarrow{\;n,\,b_1\;}_{\boldsymbol{\lambda}_{\mathrm{ND}}} \cdots \xrightarrow{\;n,\,b_k\;}_{\boldsymbol{\lambda}_{\mathrm{ND}}} s_{k+1} \qquad (3.3.1)$$

ending in a weak head normal form s_{k+1} there is also a reduction sequence

$$s_0 \xrightarrow{\;n,\,b_0'\;}_{\boldsymbol{\lambda}_{\mathrm{ND}}} s_1' \xrightarrow{\;n,\,b_1'\;}_{\boldsymbol{\lambda}_{\mathrm{ND}}} \cdots \xrightarrow{\;n,\,b_{m-1}'\;}_{\boldsymbol{\lambda}_{\mathrm{ND}}} s_m' \xrightarrow{\;\mathcal{S}\,,\,a'\;}_{\boldsymbol{\lambda}_{\approx}}{}^{0\vee 1} s_{m+1}' \qquad (3.3.2)$$

where $s_{m+1}' \equiv s_{k+1}$ and s_m' is already a WHNF. Basically, this has to be done for *every* approximation reduction and *every* normal-order reduction. Fortunately, several of these cases do not need to be considered, e.g. surface reductions inside non-reduction contexts are independent from normal-order reductions.

Lemma 3.3.3. *Let S be a surface context, which is* not *a reduction context. Then for all terms $s_0, s_1, s_2 \in \Lambda_{\approx}$, every surface reduction $\xrightarrow{\;S,\,a\;}_{\boldsymbol{\lambda}_{\approx}}$ and every normal order reduction $\xrightarrow{\;n,\,b\;}_{\boldsymbol{\lambda}_{\mathrm{ND}}}$ the following holds:*

$$s_0 \xrightarrow{\;S,\,a\;}_{\boldsymbol{\lambda}_{\approx}} s_1 \xrightarrow{\;n,\,b\;}_{\boldsymbol{\lambda}_{\mathrm{ND}}} s_2 \implies \exists s_2' : s_0 \xrightarrow{\;n,\,b\;}_{\boldsymbol{\lambda}_{\mathrm{ND}}} s_2' \xrightarrow{\;S,\,a\;}_{\boldsymbol{\lambda}_{\approx}}{}^{0\vee 1} s_2$$

Proof. Let R be the reduction context in which $\xrightarrow{\;n,\,b\;}_{\boldsymbol{\lambda}_{\mathrm{ND}}}$ takes place, i.e. the normal order reduction is $\xrightarrow{\;R,\,n,\,b\;}_{\boldsymbol{\lambda}_{\mathrm{ND}}}$ in fact. Since S is *not* a reduction context, either S and R are disjoint or S has to be of the form $S \equiv R[S']$ for $S' \in \mathcal{S}$ being some further surface context.

In the former case, lemma 2.3.14 already proves the claim. For $S \equiv R[S']$ an examination of the normal-order reduction reveals that the surface context S' will not be modified: Either it will remain intact or disappear completely. I.e., the rule (cp) leaves all surfaces position untouched. Moreover, the rules (llet), (lapp) and (lseq) only shuffle surface positions, whereas (lbeta) even creates a new one. The term which is dropped by rule (eseq) does not contain any surface position. So for all of these rules the commutation is possible. Hence the only rules which can make a subsequent approximation reduction superfluous are (nd, left) and (nd, right) respectively. $\qquad\square$

Hence, by this lemma, we only have to show that $\xrightarrow{\mathcal{R},\ a}_{\boldsymbol{\lambda}_\approx}$-reductions, i.e., reductions which, in the sense of definition 3.1.9, are *internal* inside a reduction context, "commute" with normal-order reductions. By lemma 3.1.10 there are no internal reductions of type (lapp), (lbeta), (lseq), (eseq) or (nd) within a reduction context. So (llet) and (cp) would be possible only. As both of these have been removed in the $\boldsymbol{\lambda}_\approx$-calculus, solely diagrams for (cpa) and (stop) remain to be constructed.

Beforehand, we will analyse the base case of such a diagram, i.e. that whenever there is a one-step approximation reduction for a term s to a WHNF, then s has also a normal-order reduction to a WHNF.

Lemma 3.3.4. *Let* $s, t \in \Lambda_\approx$ *be terms such that* $s \xrightarrow{\mathcal{R},\ a}_{\boldsymbol{\lambda}_\approx} t$ *and* t *is a WHNF. Then either* s *is a WHNF, or there is a normal-order reduction* $s \xrightarrow{n}_{\boldsymbol{\lambda}_{\mathrm{ND}}} t'$ *in one step to a WHNF* t' *such that either* $t' \equiv t$ *or* $t' \xrightarrow{\mathcal{S},\ cpa}_{\boldsymbol{\lambda}_\approx} t$ *holds.*

Proof. Let $s \xrightarrow{\mathcal{R},\ a}_{\boldsymbol{\lambda}_\approx} t$. If this is a $\xrightarrow{i\mathcal{R},\ a}_{\boldsymbol{\lambda}_{\mathrm{ND}}}$-reduction, i.e. internal within a reduction context, then, by lemma 3.1.14, s is already a WHNF. Therefore we assume that t is a WHNF but s is not. If the reduction $s \xrightarrow{\mathcal{R},\ a}_{\boldsymbol{\lambda}_\approx} t$ forms a normal-order reduction in the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus, nothing has to be shown. Hence, only the rules new in $\boldsymbol{\lambda}_\approx$, but not present in $\boldsymbol{\lambda}_{\mathrm{ND}}$, have to be regarded.

- The case $a = \mathrm{stop}$ is not possible: Assume $s \equiv R[s']$, then $t \equiv R[\odot]$ cannot be a weak head normal form.

- If $a = \mathrm{cpa}$ is to produce a weak head normal form in one step, it must be

$$s \equiv L_R^*[\texttt{let } x = \lambda z.q \texttt{ in } {L_R^*}'[x]] \xrightarrow{\mathcal{R},\ \mathrm{cpa}} L_R^*[({L_R^*}'[x])[\lambda z.q/x]] \equiv t$$

  for which there is also a $\xrightarrow{n,\ \mathrm{cp}}_{\boldsymbol{\lambda}_{\mathrm{ND}}}$-reduction, namely

$$s \equiv L_R^*[\texttt{let } x = \lambda z.q \texttt{ in } {L_R^*}'[x]] \xrightarrow{n,\ \mathrm{cp}} L_R^*[\texttt{let } x = \lambda z.q \texttt{ in } {L_R^*}'[\lambda z.q]] \equiv t'$$

  Since $L_R^*[({L_R^*}'[x])[\lambda z.q/x]] \equiv L_R^*[({L_R^*}'[\lambda z.q])[\lambda z.q/x]]$ we clearly have

$$t' \equiv L_R^*[\texttt{let } x = \lambda z.q \texttt{ in } {L_R^*}'[\lambda z.q]] \xrightarrow{\mathcal{R},\ \mathrm{cpa}} L_R^*[({L_R^*}'[\lambda z.q])[\lambda z.q/x]] \equiv t$$

  Thus the lemma is shown. $\square$

(cpa) commutes with normal-order reductions

For a reduction by rule (cpa) inside a reduction context $R \in \mathcal{R}$ assume

$$R[\texttt{let } x = s \texttt{ in } t] \xrightarrow{R,\ \text{cpa}} R[t[s/x]]$$

where s is $\odot$ or an abstraction. On one hand the subsequent normal-order redex may be completely independent of where the substitution takes place. On the other hand, (lbeta) or (eseq) may use the abstraction substituted for x, so a preceding $\xrightarrow{n,\ \text{cp}}$ is necessary. Note that s cannot be $\odot$ in this case. If $t \equiv L_R^*[\texttt{let } y = x \texttt{ in } t']$ and $\texttt{let } y = x \texttt{ in } t'$ becomes the $\xrightarrow{n,\ \text{cp}}$-redex after (cpa), then $s \not\equiv \odot$ and two successive $\xrightarrow{n,\ \text{cp}}$-reductions have to be performed.

$$\xrightarrow{R,\ \text{cpa}} \cdot \xrightarrow{n,\ a} \quad \leadsto \quad \xrightarrow{n,\ a} \cdot \xrightarrow{R,\ \text{cpa}}$$

$$\xrightarrow{R,\ \text{cpa}} \cdot \xrightarrow{n,\ \text{lbeta}} \quad \leadsto \quad \xrightarrow{n,\ \text{cp}} \cdot \xrightarrow{n,\ \{\text{lbeta, eseq}\}} \cdot \xrightarrow{R,\ \text{cpa}}$$

$$\xrightarrow{R,\ \text{cpa}} \cdot \xrightarrow{n,\ \text{cp}} \quad \leadsto \quad \xrightarrow{n,\ \text{cp}} \cdot \xrightarrow{n,\ \text{cp}} \cdot \xrightarrow{R,\ \text{cpa}}$$

It is worth mentioning that the reduction context for the (cpa)-reduction remains the same. This is also true, if $t[s/x]$ is an abstraction which occurs in the head position of normal-order (lbeta)-redex. In this case we have to insert a normal-order (lapp)-reduction in front.

$$\xrightarrow{R,\ \text{cpa}} \cdot \xrightarrow{n,\ \text{lbeta}} \quad \leadsto \quad \xrightarrow{n,\ \text{lapp}} \cdot \xrightarrow{n,\ \text{lbeta}} \cdot \xrightarrow{R,\ \text{cpa}}$$

If the abstraction for the (lbeta)-normal-order reduction is created by the preceding (cpa), a normal-order (cp) has to follow the (lapp)-reduction:

$$\xrightarrow{R,\ \text{cpa}} \cdot \xrightarrow{n,\ \text{lbeta}} \quad \leadsto \quad \xrightarrow{n,\ \text{lapp}} \cdot \xrightarrow{n,\ \text{cp}} \cdot \xrightarrow{n,\ \text{lbeta}} \cdot \xrightarrow{R,\ \text{cpa}}$$

Similar is the situation for (eseq) and its combination with the (lseq)-rule.

$$\xrightarrow{R,\ \text{cpa}} \cdot \xrightarrow{n,\ \text{eseq}} \quad \leadsto \quad \xrightarrow{n,\ \text{lseq}} \cdot \xrightarrow{n,\ \text{eseq}} \cdot \xrightarrow{R,\ \text{cpa}}$$

$$\xrightarrow{R,\ \text{cpa}} \cdot \xrightarrow{n,\ \text{eseq}} \quad \leadsto \quad \xrightarrow{n,\ \text{lseq}} \cdot \xrightarrow{n,\ \text{cp}} \cdot \xrightarrow{n,\ \text{eseq}} \cdot \xrightarrow{R,\ \text{cpa}}$$

Now consider $R \equiv L_R^*[\texttt{let } y = [\ ] \texttt{ in } R'[x]]$ which may only be the case when $\texttt{let } x = s \texttt{ in } t$ lies inside the normal-order redex. Then possibly a (llet)-reduction has to be placed at the beginning of the reduction sequence, thereby modifying the reduction context where (cpa) is performed.

$$\xrightarrow{L_R^*[\texttt{let } y=[\ ] \texttt{ in } R'[x]],\ \text{cpa}} \cdot \xrightarrow{n,\ a} \quad \leadsto \quad \xrightarrow{n,\ \text{llet}} \cdot \xrightarrow{n,\ a} \cdot \xrightarrow{L_R^*,\ \text{cpa}}$$

Since the effect of the reduction rule (cpa) also consists of a garbage collection, it cannot be simulated by any normal-order reduction. Hence a (cpa)-reduction might not disappear.

Lemma 3.3.5. *A complete set of commuting diagrams for* $\xrightarrow{\mathcal{R},\ cpa}$ *w.r.t.* $\xrightarrow{n}$ *is:*

$$\xrightarrow{\mathcal{R},\ cpa} \cdot \xrightarrow{n,\ a} \quad \rightsquigarrow \quad \xrightarrow{n,\ a} \cdot \xrightarrow{\mathcal{R},\ cpa}$$

$$\xrightarrow{\mathcal{R},\ cpa} \cdot \xrightarrow{n,\ a} \quad \rightsquigarrow \quad \xrightarrow{n,\ cp} \cdot \xrightarrow{n,\ a} \cdot \xrightarrow{\mathcal{R},\ cpa} \quad \textit{if } a \in \{\textit{lbeta, eseq, cp}\}$$

$$\xrightarrow{\mathcal{R},\ cpa} \cdot \xrightarrow{n,\ a} \quad \rightsquigarrow \quad \xrightarrow{n,\ b} \cdot \xrightarrow{n,\ cp}{}^{0\vee1} \cdot \xrightarrow{n,\ a} \cdot \xrightarrow{\mathcal{R},\ cpa}$$
$$\textit{if } (a,b) \in \{(\textit{lbeta, lapp}), (\textit{eseq, lseq})\}$$

$$\xrightarrow{\mathcal{R},\ cpa} \cdot \xrightarrow{n,\ a} \quad \rightsquigarrow \quad \xrightarrow{n,\ llet} \cdot \xrightarrow{n,\ a} \cdot \xrightarrow{\mathcal{R},\ cpa}$$

All these transformation diagrams in the above lemma share one basic pattern, namely that they do not duplicate the (cpa)-reduction. Because of this property the composition of such diagrams terminates. As a measure for a reduction sequence consider the weighted sum $\sum_{i=1}^{\#(\mathrm{cpa})} (f-p)\,i$ counting for the ith (cpa)-reduction within in the sequence the number f of normal-order reductions following this (cpa)-reduction minus the number p of normal-order reductions preceding it. Then an application of a commuting diagram for (cpa) from the complete set above strictly decreases this weight.

Now that we have the prerequisites to move a (cpa)-reduction from the front of a normal-order reduction sequence to its tail, we will proceed with the remaining (stop)-rule.

(stop) commutes with normal-order reductions

If the reduction rule (stop) is applied within a reduction context, there is no subsequent normal-order reduction.

Lemma 3.3.6. *Let* $s, t \in \Lambda_\approx$ *be terms and* $R \in \mathcal{R}$ *a reduction context such that* $s \xrightarrow{R,\ stop} t$ *holds. Then* t *has no normal-order reduction.*

Proof. Assuming a reduction sequence of the form $\xrightarrow{R_i,\ stop} \cdot \xrightarrow{n,\ R_n,\ a}$ we have to distinguish only a few cases. Clearly, (stop) must *not* take place "above" the normal-order redex, i.e. $R_n \not\equiv R_i[C]$ for any other context $C \in \mathcal{C}$. Therefore, only $R_i \equiv R_n[[\,]\,e]$ and $R_i \equiv R_n[\texttt{let } x = [\,] \texttt{ in } e]$ are possible reduction contexts

for the preceding (stop)-reduction. But $R_n[\circledcirc\, e]$ and $R_n[\mathtt{let}\ x = \circledcirc\ \mathtt{in}\ e]$ both have no normal-order reduction either. $\qquad\square$

Commutation of $\xrightarrow{\;\mathcal{S}\;}_{\boldsymbol{\lambda}_{\approx}}$-reductions w.r.t. $\xrightarrow{\;n\;}_{\boldsymbol{\lambda}_{\mathrm{ND}}}$-reductions

So far we have seen that only (cpa)- and (stop)-reductions inside reduction contexts are worth considering. Before we proceed further towards the proof of the Approximation Theorem, we first put the preceding parts together by showing that for every reduction sequence $s \xrightarrow{\;\mathcal{S}\;}_{\boldsymbol{\lambda}_{\approx}} \cdot \xrightarrow{\;n\;}^{+} t$ such that t is a WHNF, there is either a pure normal-order reduction sequence $s \xrightarrow{\;n\;}^{+} t$ or a reduction sequence of the form $s \xrightarrow{\;n\;}^{+} \cdot \xrightarrow{\;\mathcal{S}\;}_{\boldsymbol{\lambda}_{\approx}} \cdot \xrightarrow{\;n\;}^{*} t$ to the same WHNF. This is accomplished in detail by the following lemma.

Lemma 3.3.7. *Let $s_0, s_1, s_2, s_3 \in \Lambda_{\approx}$ be terms with $s_0 \xrightarrow{\;\mathcal{S}\;}_{\boldsymbol{\lambda}_{\approx}} s_1 \xrightarrow{\;n\;} s_2 \xrightarrow{\;n\;}^{k} s_3$ and s_3 a weak head normal form. Then there exists a term $s_1' \in \Lambda_{\approx}$ such that $s_0 \xrightarrow{\;n\;}^{\leq 3} s_1' \xrightarrow{\;\mathcal{S}\;}_{\boldsymbol{\lambda}_{\approx}}^{0 \vee 1} s_2 \xrightarrow{\;n\;}^{k} s_3$ holds.*

Proof. We assume terms $s_0, s_1, s_2, s_3 \in \Lambda_{\approx}$ such that $s_0 \xrightarrow{\;\mathcal{S}\;}_{\boldsymbol{\lambda}_{\approx}} s_1 \xrightarrow{\;n\;} s_2 \xrightarrow{\;n\;}^{k} s_3$ and s_3 is a WHNF and analyse the following cases for the $\xrightarrow{\;\mathcal{S}\;}_{\boldsymbol{\lambda}_{\approx}}$-reduction:

- If $s_0 \xrightarrow{\;\mathcal{S}\;}_{\boldsymbol{\lambda}_{\approx}} s_1 \xrightarrow{\;n\;} s_2$ with a surface context $S \in \mathcal{S} \setminus \mathcal{R}$ which is not a reduction context, we have a term $s_1' \in \Lambda_{\approx}$ from lemma 3.3.3 such that $s_0 \xrightarrow{\;n\;} s_1' \xrightarrow{\;\mathcal{S}\;}_{\boldsymbol{\lambda}_{\approx}}^{0 \vee 1} s_2$ holds. Since we can prolong this into the reduction sequence $s_0 \xrightarrow{\;n\;} s_1' \xrightarrow{\;\mathcal{S}\;}_{\boldsymbol{\lambda}_{\approx}}^{0 \vee 1} s_2 \xrightarrow{\;n\;}^{k} s_3$ ending in the WHNF s_3, the proposition holds. Hence for the following cases, we may w.l.o.g. assume the reduction to take place inside a reduction context.

- The case $s_0 \xrightarrow{\;\mathcal{R},\ \mathrm{stop}\;} s_1$ is impossible since it would contradict lemma 3.3.6.

- If $s_0 \xrightarrow{\;\mathcal{R},\ \mathrm{cpa}\;} s_1 \xrightarrow{\;n\;} s_2$ then one of the diagrams from the complete set in lemma 3.3.5 is applicable: Because of $s_2 \xrightarrow{\;n\;}^{k} s_3$ the term s_2 reduces to a WHNF. So we obtain the sequence $s_0 \xrightarrow{\;n\;}^{\leq 3} s_1' \xrightarrow{\;\mathcal{R},\ \mathrm{cpa}\;} s_2$ which can be prolonged into $s_0 \xrightarrow{\;n\;}^{\leq 3} s_1' \xrightarrow{\;\mathcal{R},\ \mathrm{cpa}\;} s_2 \xrightarrow{\;n\;}^{k} s_3$ yielding the WHNF s_3. $\qquad\square$

Now it is an easy induction to show that every $\xrightarrow{\;\mathcal{S}\;}_{\boldsymbol{\lambda}_{\approx}}$-reduction may be moved from the front of a normal-order reduction sequence to its tail.

Lemma 3.3.8. *Let $s_0, s_1, s_2 \in \Lambda_\approx$ be terms with $s_0 \xrightarrow{\mathcal{S}}_{\boldsymbol{\lambda}_\approx} s_1 \xrightarrow{n}{}^* s_2$ and s_2 a WHNF. Then there is also a reduction sequence $s_0 \xrightarrow{n}{}^* s_2' \xrightarrow{\mathcal{S}}{}^{0 \vee 1}_{\boldsymbol{\lambda}_\approx} s_2$ such that already s_2' is a WHNF.*

Proof. For a proof by induction on the length k of the normal order reduction sequence assume terms s_i with $0 \leq i \leq k+1$ such that $s_0 \xrightarrow{\mathcal{S}}_{\boldsymbol{\lambda}_\approx} s_1$, $s_i \xrightarrow{n} s_{i+1}$ for $0 < i \leq k$ and s_{k+1} is a WHNF.

- If $k = 0$ then the approximation reduction $s_0 \xrightarrow{\mathcal{S}}_{\boldsymbol{\lambda}_\approx} s_1$ must be take place within a reduction context, thus lemma 3.3.4 shows the claim.

- For $k > 0$ assume the statement to be true for normal order reduction sequences of length at most $k - 1$ and split up the given sequence as follows:
$$s_0 \xrightarrow{\mathcal{S}}_{\boldsymbol{\lambda}_\approx} s_1 \xrightarrow{n} s_2 \xrightarrow{n}{}^{k-1} s_{k+1}$$

  Then by lemma 3.3.7, we either already have $s_0 \xrightarrow{n}{}^{\leq 3} s_2$, for which nothing has to be shown, or obtain a term s_1' such that

$$s_0 \xrightarrow{n}{}^{\leq 3} s_1' \xrightarrow{\mathcal{S}}_{\boldsymbol{\lambda}_\approx} s_2 \xrightarrow{n}{}^{k-1} s_{k+1}$$

  Now the induction hypothesis may be applied to the remaining shorter sequence $s_1' \xrightarrow{\mathcal{S}}_{\boldsymbol{\lambda}_\approx} s_2 \xrightarrow{n}{}^{k-1} s_{k+1}$, thus the claim holds. □

Note that the induction argument in the proof is only valid because none of the commuting diagrams for $\xrightarrow{\mathcal{S},\,\text{cpa}}$ multiplies the number of (cpa)-reductions.

So with the ability to transform converging $\xrightarrow{\mathcal{S}}_{\boldsymbol{\lambda}_\approx}$- into converging $\xrightarrow{n}_{\boldsymbol{\lambda}_{\text{ND}}}$- reduction sequences one part of the Approximation Theorem can be established. For the other part we still need the transformation of converging $\xrightarrow{n}_{\boldsymbol{\lambda}_{\text{ND}}}$- into converging $\xrightarrow{\mathcal{S}}_{\boldsymbol{\lambda}_\approx}$-reduction sequences which is addressed in the next section. Both parts will eventually add up to the whole proof in section 3.3.3.

3.3.2   Transforming $\xrightarrow{n}_{\boldsymbol{\lambda}_{\text{ND}}}$- into $\xrightarrow{\mathcal{S}}_{\boldsymbol{\lambda}_\approx}$-reduction sequences

We will now address the first part of the Approximation Theorem, i.e. that for every term $s \in \Lambda_\approx$ there is a reduction $s \xrightarrow{\mathcal{S}}{}^*_{\boldsymbol{\lambda}_\approx} \lambda x.r$ whenever s has a normal order reduction $s \xrightarrow{n}{}^*_{\boldsymbol{\lambda}_{\text{ND}}} t$ such that t is a weak head normal form. Therefore remember, that every normal-order reduction is also an approximation reduction

except for the ones by rule (llet) and (cp). For the induction step, it is then to show that for every reduction sequence of the form

$$s_0 \xrightarrow{n,\ \{\text{llet cp}\}} s_1 \xrightarrow{\mathcal{S},\ a_1}_{\boldsymbol{\lambda}_{\approx}} \ldots s_k \xrightarrow{\mathcal{S}\ a_k}_{\boldsymbol{\lambda}_{\approx}} s_{k+1} \qquad (3.3.3)$$

where s_{k+1} is an abstraction, there is also a pure approximation reduction sequence $s_0 \xrightarrow{\mathcal{S}\ *}_{\boldsymbol{\lambda}_{\approx}} s'$ ending with an abstraction. As explained before, a normal-order reduction by rule (cp) will be replaced by (cpa) for which the result of lemma 3.2.12 comes into play.

Lemma 3.3.9. *Let* $s, t \in \Lambda_{\approx}$ *be terms such that* $s \xrightarrow{n,\ cp} t$ *holds. If* t *has an approximation reduction to an abstraction* $\lambda x.t'$, *there is also an approximation reduction for* s *leading to an abstraction* $\lambda x.s'$ *such that* $\lambda x.t'$ *differs from* $\lambda x.s'$ *only by internal* (stop)*-reductions, i.e.* $\lambda x.s' \xrightarrow{i,\ stop\ *} \lambda x.t'$ *is true.*

Proof. We may assume w.l.o.g.

$$s \equiv L_R^*[\text{let } z = \lambda y.p \text{ in } R[z]] \xrightarrow{n,\ \text{cp}} L_R^*[\text{let } z = \lambda y.p \text{ in } R[\lambda y.p]] \equiv t$$

Furthermore, suppose t has an approximation reduction to an abstraction, i.e.

$$t \equiv L_R^*[\text{let } z = \lambda y.p \text{ in } R[\lambda y.p]] \xrightarrow{\mathcal{S}\ *}_{\boldsymbol{\lambda}_{\approx}} \lambda x.t'$$

Then, because of its structure, also (cpa) may be applied to t:

$$t \equiv L_R^*[\text{let } z = \lambda y.p \text{ in } R[\lambda y.p]] \xrightarrow{\text{cpa}} L_R^*[(R[\lambda y.p])[\lambda y.p/z]]$$

Hence from lemma 3.2.12 we obtain the following approximation reduction:

$$L_R^*[(R[\lambda y.p])[\lambda y.p/z]] \xrightarrow{\mathcal{S}\ *}_{\boldsymbol{\lambda}_{\approx}} \lambda x.t'' \xrightarrow{i,\ \text{stop}\ *} \lambda x.t'$$

Note that we could also reduce the initial term s with (cpa) directly

$$s \equiv L_R^*[\text{let } z = \lambda y.p \text{ in } R[z]] \xrightarrow{\text{cpa}} L_R^*[(R[z])[\lambda y.p/z]]$$

Clearly, $L_R^*[(R[z])[\lambda y.p/z]]$ and $L_R^*[(R[\lambda y.p])[\lambda y.p/z]]$, which results from t by the (cpa)-reduction, are syntactically the same terms. Thus the claim holds. $\square$

The other major requirement is the elimination of normal-order (llet)-reductions.

Lemma 3.3.10. *Let $s, s' \in \Lambda_\approx$ be terms such that $s \xrightarrow{n,\ llet} s'$ is a top-level reduction. Then $s' \xrightarrow[\boldsymbol{\lambda}_\approx]{\mathcal{S}\ \ *} \lambda x.r$ implies $s \xrightarrow[\boldsymbol{\lambda}_\approx]{\mathcal{S}\ \ *} \lambda x.r$, i.e. s also has a surface approximation reduction sequence to the same abstraction.*

Proof. Assume $s, s' \in \Lambda_\approx$ which match the given preconditions, that is to say

$$s \equiv \texttt{let}\ x = (\texttt{let}\ y = t_y\ \texttt{in}\ t_x)\ \texttt{in}\ R[x] \xrightarrow{n,\ llet}$$

$$\texttt{let}\ y = t_y\ \texttt{in}\ (\texttt{let}\ x = t_x\ \texttt{in}\ R[x]) \equiv s' \xrightarrow[\boldsymbol{\lambda}_\approx]{\mathcal{S}\ \ *} \lambda x.r$$

By lemma 3.2.26 we may perform reductions in the outermost `let`-environment first, i.e. within surface contexts of the form `let` $y = \mathcal{S}$ `in let` $x = t_x$ `in` $R[x]$. Hence from $s' \xrightarrow[\boldsymbol{\lambda}_\approx]{\mathcal{S}\ \ *} \lambda x.r$ we obtain a reduction sequence

$$\texttt{let}\ y = t_y\ \texttt{in}\ (\texttt{let}\ x = t_x\ \texttt{in}\ R[x]) \xrightarrow[\boldsymbol{\lambda}_\approx]{\texttt{let}\ y=\mathcal{S}\ \texttt{in}\ \texttt{let}\ x=t_x\ \texttt{in}\ R[x]\ \ *}$$

$$\texttt{let}\ y = t'_y\ \texttt{in}\ (\texttt{let}\ x = t_x\ \texttt{in}\ R[x]) \xrightarrow{\text{cpa}}$$

$$(\texttt{let}\ x = t_x\ \texttt{in}\ R'[x])[t'_y/y] \xrightarrow[\boldsymbol{\lambda}_\approx]{\mathcal{S}\ \ *} \lambda x.r \quad (3.3.4)$$

with t'_y being $\odot$ or an abstraction now. Note, that for every surface context S also `let` $x = (\texttt{let}\ y = S\ \texttt{in}\ t'_x)$ `in` $R'[x]$ is a surface context, so we may transfer the above reduction sequences to s, thereby eliminating the $\xrightarrow{n,\ llet}$-reduction.

We therefore apply to `let` $x = (\texttt{let}\ y = t_y\ \texttt{in}\ t_x)$ `in` $R[x]$ exactly the same reductions as in equation (3.3.4), but within surface contexts of the respective form `let` $x = (\texttt{let}\ y = \mathcal{S}\ \texttt{in}\ t'_x)$ `in` $R'[x]$, resulting in

$$\texttt{let}\ x = (\texttt{let}\ y = t_y\ \texttt{in}\ t_x)\ \texttt{in}\ R[x] \xrightarrow[\boldsymbol{\lambda}_\approx]{\texttt{let}\ x=(\texttt{let}\ y=\mathcal{S}\ \texttt{in}\ t_x)\ \texttt{in}\ R[x]\ \ *}$$

$$\texttt{let}\ x = (\texttt{let}\ y = t'_y\ \texttt{in}\ t_x)\ \texttt{in}\ R[x] \xrightarrow{\texttt{let}\ x=[\ ]\ \texttt{in}\ R[x],\ \text{cpa}}$$

$$\texttt{let}\ x = t_x[t'_y/y]\ \texttt{in}\ R[x]$$

which is syntactically identical to $(\texttt{let}\ x = t_x\ \texttt{in}\ R[x])[t'_y/y]$, thus the claim. $\qquad\square$

3.3.3 Proof of the Approximation Theorem

The previous sections have led to a bunch of results on replacing and commuting $\xrightarrow[\boldsymbol{\lambda}_\approx]{\mathcal{S}}$-reductions with $\xrightarrow[\boldsymbol{\lambda}_{\text{ND}}]{n}$ and vice versa. These results are now put together, establishing the whole proof of the Approximation Theorem.

Proof of Theorem 3.3.2. For the "if"-part assume an arbitrary but fixed reduction sequence $s \xrightarrow[\boldsymbol{\lambda}_{\approx}]{\mathcal{S} \ m} \lambda x.r$ of length m. We will show, that there exists also a normal-order reduction $s \xrightarrow[\boldsymbol{\lambda}_{\mathrm{ND}}]{n \ *} t$ to a WHNF t by induction on m:

- If $m = 1$ then by lemma 3.3.4 the proposition holds.

- For the induction step we assume that the claim is valid for a reduction sequence of length $< m$. Now suppose a reduction sequence $s \xrightarrow[\boldsymbol{\lambda}_{\approx}]{\mathcal{S} \ n} \lambda x.r$ of length m, which can be split as follows:

$$s \xrightarrow[\boldsymbol{\lambda}_{\approx}]{\mathcal{S}} s_1 \xrightarrow[\boldsymbol{\lambda}_{\approx}]{\mathcal{S} \ m-1} \lambda x.r$$

  By the induction hypothesis there is a normal order reduction sequence $s_1 \xrightarrow{n \ *} t$ where t is a WHNF, i.e. we have the following situation

$$s \xrightarrow[\boldsymbol{\lambda}_{\approx}]{\mathcal{S}} s_1 \xrightarrow{n \ *} t$$

  Now by lemma 3.3.8 we have $s \xrightarrow{n \ *} t' \xrightarrow[\boldsymbol{\lambda}_{\approx}]{\mathcal{S}} t$ with t' already a WHNF.

For the "only if"-part we will construct a reduction $s \xrightarrow[\boldsymbol{\lambda}_{\approx}]{\mathcal{S} \ *} \lambda x.r$, so assume a normal-order reduction sequence $s \xrightarrow[\boldsymbol{\lambda}_{\mathrm{ND}}]{n \ m} t$ of length m such that t is a WHNF.

- From $m = 1$ we have $s \xrightarrow{n} t$ where t is a WHNF and s is not — otherwise s would not have a normal-order reduction. The normal-order reduction cannot be (llet), since then s would already be a WHNF, cf. remark 3.1.8 for the possible cases.

  The normal-order reductions by rule (lbeta), (eseq) and (nd) all are valid approximation reductions, too. So for $t \equiv \lambda x.r$ nothing has to be shown. Hence assume $t \equiv L_R^+[\lambda x.r]$ with L_R^+ be a non-empty environment. Then we may apply (stop) to every term bound by L_R^+ followed by a sequence of (cpa)-reductions. For instance, if x_i denotes the variables bound by L_R^+ this yields $\lambda x.r[\odot/x_i]$, i.e. an abstraction.

  Therefore, (cp) is the only remaining possibility for a normal-order reduction which is *not* an approximation reduction. However, according to remark 3.1.8 this may only happen in the specific case

$$L_R^*[\texttt{let } y = \lambda x.r \texttt{ in } L_R'^*[y]] \xrightarrow{n, \ \mathrm{cp}} L_R^*[\texttt{let } y = \lambda x.r \texttt{ in } L_R'^*[\lambda x.r]]$$

  Clearly, this effect can be simulated by an application of the (cpa)-rule. An abstraction can then be obtained as described above, by applying (stop) to the left hand sides in $L_R^*[L_R'^*]$ followed by appropriate (cpa)-reductions.

- Assume the proposition holds for normal-order reduction sequences of length $k < m$. Then a sequence $s \xrightarrow{n}{}^m t$ may be split up as follows

$$s \xrightarrow{n} s_1 \xrightarrow{n}{}^{m-1} t$$

with t being a WHNF. By the induction hypothesis, $s_1 \xrightarrow{\mathcal{S}}{}^*_{\boldsymbol{\lambda}_\approx} \lambda x.r$, hence

$$s \xrightarrow{n} s_1 \xrightarrow{\mathcal{S}}{}^*_{\boldsymbol{\lambda}_\approx} \lambda x.r$$

So we examine the possibilities for $s \xrightarrow{n} s_1$:

- If $s \xrightarrow{n} s_1$ is also a $\rightarrow_{\boldsymbol{\lambda}_\approx}$-reduction then $s \xrightarrow{\mathcal{S}}{}^*_{\boldsymbol{\lambda}_\approx} \lambda x.r$.

- For $L_R^*[\texttt{let } x = \lambda y.s \texttt{ in } R[x]] \xrightarrow{n, \text{ cp}} L_R^*[\texttt{let } x = \lambda y.s \texttt{ in } R[\lambda y.s]]$ we may apply lemma 3.3.9 which proves the proposition.

- In the case of

$$s \equiv L_R^*[\texttt{let } x = (\texttt{let } y = t_y \texttt{ in } t_x) \texttt{ in } R[x]] \xrightarrow{n, \text{ llet}}$$
$$L_R^*[\texttt{let } y = t_y \texttt{ in } (\texttt{let } x = t_x \texttt{ in } R[x])] \equiv s_1$$

we obtain, by theorem 3.2.28, the following reduction sequence for s_1

$$L_R^*[\texttt{let } y = t_y \texttt{ in } (\texttt{let } x = t_x \texttt{ in } R[x])] \xrightarrow{\text{olf}}{}^*$$
$$\texttt{let } y = t'_y \texttt{ in } (\texttt{let } x = t'_x \texttt{ in } R'[x]) \xrightarrow{\mathcal{S}}{}^*_{\boldsymbol{\lambda}_\approx} \lambda x.r \quad (3.3.5)$$

Here $\xrightarrow{\text{olf}}{}^*$ proceeds completely inside the L_R^*-surface context. Therefore, these reductions could be transferred to s, giving

$$s \equiv L_R^*[\texttt{let } x = (\texttt{let } y = t_y \texttt{ in } t_x) \texttt{ in } R[x]] \xrightarrow{\text{olf}}{}^*$$
$$\texttt{let } x = (\texttt{let } y = t'_y \texttt{ in } t'_x) \texttt{ in } R'[x] \quad (3.3.6)$$

The terms, i.e. $\texttt{let } x = (\texttt{let } y = t'_y \texttt{ in } t'_x) \texttt{ in } R'[x]$ of (3.3.5) and $\texttt{let } y = t'_y \texttt{ in } (\texttt{let } x = t'_x \texttt{ in } R'[x])$ of (3.3.6) respectively, we got from the (olf)-reductions, are identical up to an (llet)-reduction:

$$\texttt{let } x = (\texttt{let } y = t'_y \texttt{ in } t'_x) \texttt{ in } R'[x] \xrightarrow{n, \text{ llet}}$$
$$\texttt{let } y = t'_y \texttt{ in } (\texttt{let } x = t'_x \texttt{ in } R'[x])$$

Since this is top-level normal-order, lemma 3.3.10 applies, hence

$$\texttt{let } x = (\texttt{let } y = t'_y \texttt{ in } t'_x) \texttt{ in } R'[x] \xrightarrow{\;\;\mathcal{S}\;\;}{}^{*}_{\lambda_\approx} \lambda x.r$$

and thus the claim holds by

$$s \equiv L^*_R[\texttt{let } x = (\texttt{let } y = t_y \texttt{ in } t_x) \texttt{ in } R[x]] \xrightarrow{\;\;\text{olf}\;\;}{}^{*}$$

$$\texttt{let } x = (\texttt{let } y = t'_y \texttt{ in } t'_x) \texttt{ in } R'[x] \xrightarrow{\;\;\mathcal{S}\;\;}{}^{*}_{\lambda_\approx} \lambda x.r \qquad \square$$

Thus, the preceding sections have dealt with two non-deterministic call-by-need lambda calculi, namely the $\boldsymbol{\lambda}_{\mathrm{ND}}$- and the $\boldsymbol{\lambda}_\approx$-calculus. The above proof of the Approximation Theorem finally establishes the equivalence of convergence in both of these calculi. It will be shown later how this result implies that the respective contextual equivalences of the calculus $\boldsymbol{\lambda}_{\mathrm{ND}}$ and $\boldsymbol{\lambda}_\approx$ coincide.

3.4 Related Work

At the end of this chapter, some related work is discussed. Worth mentioning is that some calculi, as e.g. [MS99, MSC99a], have a restricted syntax regarding the use of applications: Only variables are permitted in the argument position. The purpose is to simplify matters w.r.t. sharing. However, although the transformation $s\,t \rightarrow \texttt{let } x = t \texttt{ in } s\,x$ is legal, cf. [SS03a], this syntax restriction prohibits rules like (cpa) in a possible approximation calculus. There seems not to be an easy workaround for this.

In the following, a representation of the constant `choice` from [Kut99] will be given for the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus. Furthermore, the relationship of call-by-need and call-by-value will be explained in more detail We will see that evaluation in the calculus $\boldsymbol{\lambda}_\approx$ could be seen, to some extent, as a kind of call-by-value evaluation. Though originally devised as an optimisation of call-by-name in a deterministic setting, the presence of non-determinism causes call-by-need to be much closer to call-by-value than to call-by-name. Kutzner, cf. [Kut99, p. 20ff.], has already remarked on that.

3.4.1 Non-deterministic `choice` as a Constant

The non-deterministic constant `choice` of [Kut99] cannot be encoded as the term $\lambda x.\lambda y.(\texttt{pick } x\ y)$ in the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus. The reason is that `choice` s has an

immediate normal-order reduction in the Λ_{let}-calculus of [Kut99] as follows:

$$\texttt{choice } s \xrightarrow{\text{nd, left}} (\lambda x.(\lambda y.x))\, s$$
$$\xrightarrow{n,\text{ lbeta}} \texttt{let } x = s \texttt{ in } (\lambda y.x)$$

Clearly, the non-deterministic choice has already been carried out. On the other hand, although $(\lambda x.\lambda y.(\texttt{pick } x\ y))\, s$ has the normal-order (lbeta)-reduction

$$(\lambda x.\lambda y.(\texttt{pick } x\ y))\, s \xrightarrow{n,\text{ lbeta}} \texttt{let } x = s \texttt{ in } (\lambda y.(\texttt{pick } x\ y))$$

the non-deterministic choice might still be made — and copied. The solution would be to observe the behaviour of the constant `choice` more carefully.

In fact, as soon as applied to an argument, the `choice`-reduction chooses between the functions **K** and **K2** and applies the result to its argument. Thereafter, a reduction by rule (lbeta) will always be necessary. This (lbeta)-reduction can also be performed before. Thus we can reproduce the original behaviour of `choice` in the calculus $\boldsymbol{\lambda}_{\mathrm{ND}}$ by setting

$$\texttt{choice} \stackrel{\text{def}}{=} \lambda x.\texttt{pick } (\lambda y.x)\ (\lambda y.y)$$

As the example `choice` s demonstrates, this is sensible:

$$\texttt{choice}\, s \equiv (\lambda x.\texttt{pick } (\lambda y.x)\ (\lambda y.y))\, s \xrightarrow{n,\text{ lbeta}} \texttt{let } x = s \texttt{ in } (\texttt{pick } (\lambda y.x)\ (\lambda y.y))$$
$$\xrightarrow{\text{nd, left}} \texttt{let } x = s \texttt{ in } (\lambda y.x)$$

It is easily seen that $\lambda x.(\texttt{choice } s\ x)$ can be distinguished from `choice` s by contexts. Establishing the contextual equivalence of the terms $\lambda x.(\texttt{choice } s\ x)$ and $\lambda x.(\texttt{pick } s\ x)$ is left to the reader.

3.4.2   Approximation and Call-by-Value Evaluation

An observation of the approximation reduction in the $\boldsymbol{\lambda}_{\approx}$-calculus reveals that it corresponds to some kind of resource-bounded, call-by-value evaluation. To see this, suppose some term $(\lambda x.t)\, s$ which converges. Such an application may directly be reduced with the (lbeta)-rule, which does not change its converging behaviour, cf. corollary 3.2.16.

To the `let`-term which arises therefrom, the normalisation of lemma 3.2.26 applies. Thus the beginning of the converging approximation reduction sequence

for $(\lambda x.t)\,s$ may be assumed to obey the following form.

$$(\lambda x.t)\,s$$

$$\xrightarrow[\boldsymbol{\lambda}_{\approx}]{\text{lbeta}}\ \texttt{let}\ x = s\ \texttt{in}\ t$$

$$\xrightarrow[\boldsymbol{\lambda}_{\approx}]{\texttt{let}\ x=[\,]\ \texttt{in}\ t}{}^{*}\ \texttt{let}\ x = s'\ \texttt{in}\ t \qquad\qquad (\text{where } s' \in \mathcal{PV})$$

$$\xrightarrow[\boldsymbol{\lambda}_{\approx}]{\text{cpa}}\ t[s'/x]$$

According to corollary 3.2.18, a reduction by rule (lbeta) may be commuted with a reduction within the second argument of an application. Therefore, also the approximation reduction sequence below can be obtained.

$$(\lambda x.t)\,s$$

$$\xrightarrow[\boldsymbol{\lambda}_{\approx}]{(\lambda x.t)\,[\,]}{}^{*}\ (\lambda x.t)\,s' \qquad\qquad (\text{where } s' \in \mathcal{PV})$$

$$\xrightarrow[\boldsymbol{\lambda}_{\approx}]{\text{lbeta}}\ \texttt{let}\ x = s'\ \texttt{in}\ t$$

$$\xrightarrow[\boldsymbol{\lambda}_{\approx}]{\text{cpa}}\ t[s'/x]$$

Looking at the role of (lbeta) and (cpa) in the above sequence, it becomes apparent that they may be combined into the following two rules.

$$(\lambda x.s)\,\odot \xrightarrow{\beta_{\odot}} s[\odot/x] \qquad\qquad\qquad (\beta_{\odot})$$

$$(\lambda x.s)\,(\lambda y.t) \xrightarrow{\beta_{\lambda}} s[\lambda y.t/x] \qquad\qquad (\beta_{\lambda})$$

The second (β)-rule is common for a call-by-value calculus, cf. [Plo75], where it would be the only one. In this case here, it is complemented by one variant for the $\odot$-term since in the $\boldsymbol{\lambda}_{\approx}$-calculus also $\odot$ may be copied. Thus, when classifying these above copying (β)-rules we might speak of "call-by-pseudo-value" in contrast to call-by-value where only (β_{λ}) is applicable.

However, this view seems a bit too simplistic. It stems from the fact that abstractions, as variable-binding operators, may be copied regardless of the subterms they contain. I.e., the situation is different when, e.g., data types come into play. Therefore, consider the $(\lambda x.(\texttt{fst}\ x)\,(\texttt{fst}\ x))\,(\texttt{pick}\ r\ s,t)$, where pairs are denoted by (a,b) from which $\texttt{fst}$ extracts the first component.

Regarding the pair $(\texttt{pick}\ r\ s,t)$ as a value which may be copied has undesired effects, cf. [SSSS04]. So reduction must be continued within the components of a pair as will be discussed in section 6.1 for the possible extension of the base calculus. Furthermore, even if "call-by-pseudo-value" was implemented this way,

this would not make the `let`-construct obsolete: Otherwise, recursive bindings via `letrec` could not be modelled.

Chapter 4

Similarity in $\boldsymbol{\lambda}_{\approx}$ is a Precongruence

This chapter now brings in the return of the previous two, i.e., similarity can be defined and shown to be a precongruence.

The first section of the preceding chapter presented the non-deterministic call-by-need lambda calculus $\boldsymbol{\lambda}_{\mathrm{ND}}$, the subject of this work. From example 3.1.18 it has been learned that the naive definition of similarity would not be correct w.r.t. contextual preorder. Therefore, in section 3.2 the $\boldsymbol{\lambda}_{\approx}$-calculus has been introduced. It keeps up the distinction between canonical and non-canonical terms insofar that only λ-terms do form answers. This enables us to use the definition of similarity from chapter 2 and the corresponding technique to prove it a precongruence. This means, the notion of the precongruence candidate to be stable under reduction becomes applicable. Proving the extension $(\cdot)^o$ to open terms admissible represents an essential step for this.

This open extension has to be defined first in section 4.1 right before similarity itself. The structure of the remaining chapter is then relatively simple. Section 4.2 deals with the specific definition of the precongruence candidate for the calculus $\boldsymbol{\lambda}_{\approx}$ and actually provides the result that the open extension $(\cdot)^o$ is admissible. Subsequently, in section 4.3 the precongruence candidate will be shown stable under reduction. This leads to the Precongruence Theorem showing that similarity in the $\boldsymbol{\lambda}_{\approx}$-calculus, or more precisely, its extension to open terms, is a precongruence. Finally, section 4.4 discusses possible future work.

4.1   Similarity in the $\boldsymbol{\lambda}_{\approx}$-Calculus

This section is dealing with similarity for the $\boldsymbol{\lambda}_{\approx}$-calculus. The definition of similarity accords to the ideas that were presented in section 2.4 before. Proving the extension of similarity to open terms a precongruence forms one of the major themes in this work. This result is stated in the Precongruence Theorem which will be established by applying the proof method for lazy computations systems described in sections 2.2 and 2.4.1.

The use of this powerful framework still requires quite some effort. First note that definition 2.4.1 makes use of the extension $(\cdot)^o$ of a preorder to open terms. Hence its concrete definition in section 4.1.1 is a prerequisite for similarity. Some of the properties necessary for showing the open extension $(\cdot)^o$ admissible will already be established there, too. Recall that admissibility of the open extension was a precondition for the application of theorem 2.2.13. That theorem formulated the crucial connection between a precongruence and its precongruence candidate relation.

The precongruence candidate for similarity in the calculus $\boldsymbol{\lambda}_{\approx}$ will be defined in section 4.2 where it is necessary to show that the open extension is admissible. Another requisite to apply theorem 2.2.13 is the stability of the precongruence candidate under reduction which is the subject of section 4.3.1. Establishing all those results requires a better understanding and sometimes different representations of similarity which therefore are treated in section 4.1.2. Moreover, section 4.1.4 demonstrates how similarity can be used to prove the reduction rules of the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus correct while section 4.1.3 is devoted to a deeper analysis of reduction as well as similarity on open terms, called open similarity.

4.1.1   The Open Extension $(\cdot)^o$ in the $\boldsymbol{\lambda}_{\approx}$-Calculus

Not only in lambda calculi it is common to extend a preorder to open terms by demanding the relation to hold for all closing substitutions. This principle may be found, e.g., in [Abr90] and also Howe relies on it in his work.

However, in a call-by-need calculus like $\boldsymbol{\lambda}_{\mathrm{ND}}$ and $\boldsymbol{\lambda}_{\approx}$ respectively, such an approach *will* break sharing. To see this, consider the following example.

Example 4.1.1. *Let $s \equiv y\,y$ and $t \equiv (\lambda x.x\,x)\,y$ be terms. In the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus, these are contextually equivalent as both uses of the variable y in s will be shared.*

Now regard the substition $\sigma = \{y \mapsto \mathtt{pick}\ a\ b\}$ where a and b denote contextually different terms. While $\sigma(t)$ yields $\mathtt{let}\ x = \mathtt{pick}\ a\ b\ \mathtt{in}\ (x\,x)$ after a

normal-order (lbeta)*-reduction, the term $\sigma(s) \equiv (\text{pick } a\ b)\,(\text{pick } a\ b)$ contains two non-deterministic choices between a and b.*

It is noteworthy that the effect only becomes visible because of the non-determinism, or more precisely, because in the presence of non-determinism not every term may be copied. The rule (cpa) in the calculus $\boldsymbol{\lambda}_\approx$ suggests that making copies of $\odot$ and arbitrary abstractions is correct. This obviously extends to terms in their respective equivalence classes.

From the complete set of forking diagrams in section 3.2.2 a proof could be constructed that the rule (cpa) preserves contextual equivalence. However, this is beyond the scope of this work and thus not carried out here.

Definition 4.1.2. *A closed term is said to be a* pseudo-value *if it is $\odot$ or an abstraction. We write $\mathcal{PV} = \{\, p \in \Lambda^0_\approx \mid p \equiv \odot \vee p \equiv \lambda z.q \,\}$ for the set of all pseudo-values. Moreover, a substitution σ whose range satisfies $\boldsymbol{rng}(\sigma) \subseteq \mathcal{PV}$ is called a* pseudo-valued *substitution, or $\mathcal{PV}$-substitution for short.*

The open extension $(\cdot)^o$ is then defined in terms of pseudo-valued substitutions. Later, when similarity will have been defined, we will be able to show that in the calculus $\boldsymbol{\lambda}_\approx$ pseudo-valued substitutions have the same expressiveness as arbitrary `let`-environments. The advantage of (pseudo-valued) substitutions is that they do not depend on the `let`-construct.

Definition 4.1.3. *Let $\eta \subseteq \Lambda^0_\approx \times \Lambda^0_\approx$ be a preorder on closed terms. Then its extension $\eta^o \subseteq \Lambda_\approx \times \Lambda_\approx$ to open terms is defined by relating two terms $s, t \in \Lambda_\approx$, written $s\ \eta^o\ t$, if and only if $\sigma(s)\ \eta\ \sigma(t)$ for all pseudo-valued closing substitutions σ holds.*

In other words, open terms are related when the underlying preorder does so for all kinds of instantiations with $\odot$ or a closed abstraction. Of course, it is suitable to consider closing substitutions with a "minimal" domain in the sense, that for open terms $s, t \in \Lambda_\approx$ the domain $\boldsymbol{dom}(\sigma)$ of the substitution σ is just the set of free variables, i.e., $\boldsymbol{dom}(\sigma) = \mathcal{FV}(s) \cup \mathcal{FV}(t)$ holds.

Corollary 4.1.4. *Let $\eta \subseteq \Lambda^0_\approx \times \Lambda^0_\approx$ be a preorder on closed terms and $s, t \in \Lambda_\approx$ be two (possibly open) terms. Then $s\ \eta^o\ t$ if and only if $\sigma(s)\ \eta\ \sigma(t)$ holds for all pseudo-valued substitutions σ such that $\boldsymbol{dom}(\sigma) = \mathcal{FV}(s) \cup \mathcal{FV}(t)$ is true.*

It is not necessary to substitute the free variables all at once. Instead, this can be done successively which is used in the proof of the substitution lemmas.

Corollary 4.1.5. *Let $s, t \in \Lambda_\approx$ be (possibly open) terms and $\eta \subseteq \Lambda_\approx^0 \times \Lambda_\approx^0$ be a preorder on closed terms. Then $s \ \eta^o \ t$ implies $\rho(s) \ \eta^o \ \rho(t)$ for every pseudo-valued substitution ρ with domain $\boldsymbol{dom}(\rho) \subseteq \mathcal{FV}(s) \cup \mathcal{FV}(t)$.*

Proof. The composition $\sigma \circ \rho$ of the substitutions ρ, σ is again pseudo-valued. $\qquad \square$

We proceed with a property which has been fundamental for the precongruence proof in theorem 2.2.13.

Lemma 4.1.6. *The open extension $(\cdot)^o$ is preorder-preserving.*

Proof. Assume a preorder $\eta \subseteq \Lambda_\approx^0 \times \Lambda_\approx^0$ on closed terms. It has to be shown that also η^o is a preorder. However, reflexivity and transitivity transfer from η to η^o easily: The former is trivial. So assume terms $r, s, t \in \Lambda_\approx$ such that $r \ \eta^o \ s$ as well as $s \ \eta^o \ t$ holds. Thus $\sigma(r) \ \eta \ \sigma(s)$ and $\sigma(s) \ \eta \ \sigma(t)$ for every pseudo-valued substitution. Since η is transitive, $\sigma(r) \ \eta \ \sigma(t)$ holds. $\qquad \square$

This is exactly condition 1 of definition 2.2.11 for $(\cdot)^o$ to be admissible. It is notable that in the proof of the previous lemma the restriction to pseudo-valued closing substitutions is immaterial. I.e., the argument is the same as if all closing substitutions had to be considered. This indicates the capability of the approach using pseudo-valued substitutions. Even property 3 and 2 of definition 2.2.11 can be established this way in the following.

However, the notion of a pseudo-value has to be justified by the calculus. This means it may comprise those terms only, for which copying is correct. Otherwise, the condition $\widehat{\eta} \subseteq (\widehat{\eta}_0)^o$, i.e., property 4 from definition 2.2.11, cannot be met. Section 4.2, which is devoted to its proof, will illustrate this when dealing with the precongruence candidate in the $\boldsymbol{\lambda}_\approx$-calculus.

The corollary below plays a major role for the stability of the precongruence candidate $\widehat{\eta}$ under reduction.

Corollary 4.1.7. *Let $\eta \subseteq \Lambda_\approx^0 \times \Lambda_\approx^0$ be a preorder. Then $\eta \circ \eta^o \subseteq \eta^o$ holds.*

Proof. Assume closed terms $r, s \in \Lambda_\approx^0$ and a (possibly open) term $t \in \Lambda_\approx$ such that $r \ \eta \ s$ and $s \ \eta^o \ t$ holds. The latter means that for every pseudo-valued substitution σ the condition $\sigma(s) \ \eta \ \sigma(t)$ is valid. This boils down to $s \ \eta \ \sigma(t)$ since s is closed and therefore $\sigma(s) \equiv s$ holds. Thus, for every pseudo-valued substitution σ we have $r \ \eta \ s \ \eta \ \sigma(t)$ and the claim is true since η is transitive. $\qquad \square$

The following corollary establishes property 2 of definition 2.2.11 for the open extension $(\cdot)^o$ being admissible.

Corollary 4.1.8. *Let $\eta \subseteq \Lambda_\approx^0 \times \Lambda_\approx^0$ be a relation. Then $(\eta^o)_0 = \eta$ holds.*

Proof. Since $s\,(\eta^o)_0\,t$ means $s\,\eta^o\,t$ under the premise that $s, t \in \Lambda_\approx^0$ are closed terms, for all pseudo-valued substitutions σ we obtain $s \equiv \sigma(s)\,\eta\,\sigma(t) \equiv t$. $\qquad\square$

Eventually, property 3 of definition 2.2.11 will be discussed, i.e., that $(\cdot)^o$ is monotonic w.r.t. set inclusion.

Corollary 4.1.9. *Let $\nu, \eta \subseteq \Lambda_\approx^0 \times \Lambda_\approx^0$ be relations. Then $\nu \subseteq \eta \implies \nu^o \subseteq \eta^o$.*

Proof. Assume $\nu \subseteq \eta$ and (possibly open) terms $s, t \in \Lambda_\approx$ such that $s\,\nu^o\,t$ holds. Then $s\,\eta^o\,t$ is to show. From $s\,\nu^o\,t$ for every pseudo-valued substitution σ we have $\sigma(s)\,\nu\,\sigma(t)$. Since $\nu \subseteq \eta$ by the premise, $\sigma(s)\,\eta\,\sigma(t)$ may be inferred. $\qquad\square$

4.1.2 Representations for Similarity

Recalling that $\mathtt{ans}(\cdot)$ denotes the set of all possible answers for some term, in the calculus $\boldsymbol{\lambda}_\approx$ the experiment $[\cdot]$ may be defined as $[\cdot]_\approx$ below.

$$s\ [\eta]_\approx\ t\ \overset{\text{def}}{\iff}\ \forall \lambda x.s' \in \mathtt{ans}(s): \ \exists \lambda x.t' \in \mathtt{ans}(t): \ s'\,\eta\,t' \qquad (4.1.1)$$

It should be noted that the additional degree of non-determinism introduced by rule (stop) is essential to compute the answer set $\mathtt{ans}(\cdot)$ in combination with the (cpa)-rule. The example below exhibits a situation in which experiments seem by far easier to deal with than contextual preorder.

Example 4.1.10. *Let $r, s, t \in \Lambda_\approx^0$ be closed and $\eta \subseteq \Lambda_\approx \times \Lambda_\approx$ be a preorder:*

$$r\ [\eta]_\approx\ t\ \wedge\ s\ [\eta]_\approx\ t \implies \mathtt{pick}\ r\ s\ [\eta]_\approx\ t$$

i.e. if t behaves "better" than both r and s, then it is immaterial which one is chosen thereof. So assume $\mathtt{pick}\ r\ s \Downarrow \lambda y.p$ then, by lemma 2.3.14, the reduction by rule (nd) may be moved forward, hence $r \Downarrow \lambda y.p$ or $s \Downarrow \lambda y.p$. Since by the premise we have $r\ [\eta]_\approx\ t$ and $s\ [\eta]_\approx\ t$ the claim is shown.

We proceed with some basic properties of $[\cdot]_\approx$ which are pretty self-explanatory. Therefore, subsequently only a few comments are to be found but rather some examples. Note that $s\ [\emptyset]_\approx\ t$ implies $\forall \eta: \ s\ [\eta]_\approx\ t$ since $[\cdot]_\approx$ is monotone. Hence for simplicity, whenever $s\ [\eta]_\approx\ t$ has to be shown for an arbitrary η the statement $s\ [\emptyset]_\approx\ t$ will be used instead.

Corollary 4.1.11. *Let $s \in \Lambda_\approx^0$ be a closed term such that $s \not\Downarrow$ holds. Then for every closed term $t \in \Lambda_\approx^0$ the relation $s\,[\emptyset]_\approx\,t$ is true.*

Proof. Since $s \not\Downarrow$ means that $\mathtt{ans}(s) = \emptyset$ holds, the claim is valid. $\square$

The corollary below reflects the fact that an approximation reduction could diminish the set of possible answers but not enlarge it.

Corollary 4.1.12. *Let* $s, t \in \Lambda^0_\approx$ *be closed. Then* $s \xrightarrow{\mathcal{S}}{}^*_{\boldsymbol{\lambda}_\approx} t$ *implies* $t \, [\emptyset]_\approx s$.

Proof. Obviously $\mathtt{ans}(t) \subseteq \mathtt{ans}(s)$ since with $t \Downarrow \lambda x.r$ also $s \Downarrow \lambda x.r$ holds. $\square$

The next lemma will frequently be used without further reference. It establishes an important proof principle, namely that $s \, [\eta]_\approx t$ holds, if in every approximation reduction sequence from s to an abstraction there is a term s' such that also t has an approximation reduction to some term t' for which the relation $s' \, [\eta]_\approx t'$ holds. This represents some kind of short-cut, i.e., in order to show $s \, [\eta]_\approx t$ not the whole reduction sequence to an abstraction has to be taken into account but only certain points within.

Lemma 4.1.13. *For all closed terms* $s, t \in \Lambda^0_\approx$ *and all preorders* $\eta \subseteq \Lambda_\approx \times \Lambda_\approx$ *on terms the relation* $s \, [\eta]_\approx t$ *is valid whenever the following condition is satisfied:*

$$\forall \lambda x.s'' : \; s \xrightarrow{\mathcal{S}}{}^*_{\boldsymbol{\lambda}_\approx} \lambda x.s'' \implies$$
$$(\exists s', t' : \; s \xrightarrow{\mathcal{S}}{}^*_{\boldsymbol{\lambda}_\approx} s' \xrightarrow{\mathcal{S}}{}^*_{\boldsymbol{\lambda}_\approx} \lambda x.s'' \; \wedge \; t \xrightarrow{\mathcal{S}}{}^*_{\boldsymbol{\lambda}_\approx} t' \; \wedge \; s' \, [\eta]_\approx t')$$

Proof. Since the claim constitutes a central tool, we will give a detailed proof here. So we assume closed terms $s, t \in \Lambda^0_\approx$ as well as a preorder $\eta \subseteq \Lambda_\approx \times \Lambda_\approx$.

To prove $s \, [\eta]_\approx t$, we further assume $s \Downarrow \lambda x.s''$, i.e. there is an approximation reduction sequence $s \xrightarrow{\mathcal{S}}{}^*_{\boldsymbol{\lambda}_\approx} \lambda x.s''$ to a closed abstraction $\lambda x.s'' \in \Lambda^0_\approx$ arbitrary but fixed. So under the precondition

$$\exists s', t' : \; s \xrightarrow{\mathcal{S}}{}^*_{\boldsymbol{\lambda}_\approx} s' \xrightarrow{\mathcal{S}}{}^*_{\boldsymbol{\lambda}_\approx} \lambda x.s'' \; \wedge \; t \xrightarrow{\mathcal{S}}{}^*_{\boldsymbol{\lambda}_\approx} t' \; \wedge \; s' \, [\eta]_\approx t' \qquad (4.1.2)$$

we have to show that there is a $\lambda y.t''$ such that $t \Downarrow \lambda y.t''$ and $s'' \, \eta^o \, t''$ holds. So assume $s', t' \in \Lambda^0_\approx$ to be the closed terms given by (4.1.2), then from $s' \, [\eta]_\approx t'$ we obtain for every term which s' converges to, in particular for $\lambda x.s''$ fixed above, a $\lambda y.t''$ such that $s'' \, \eta^o \, t''$ holds. Since by $t \xrightarrow{\mathcal{S}}{}^*_{\boldsymbol{\lambda}_\approx} t'$, there is also an approximation reduction sequence from t to $\lambda y.t''$, the claim is shown. $\square$

The following corollary presents a special case of the precedent lemma, where the intermediate terms s', t' coincide with the abstractions at the end of an approximation reduction sequence.

Corollary 4.1.14. *For all closed terms $s, t \in \Lambda_\approx^0$ and all preorders $\eta \subseteq \Lambda_\approx \times \Lambda_\approx$ on terms the following proof principle is valid:*

$$\left(\forall s' \in \Lambda_\approx^0 : \ s \Downarrow s' \implies (\exists t' \in \Lambda_\approx^0 : \ t \Downarrow t' \ \wedge \ s' \ [\eta]_\approx \ t')\right) \implies s \ [\eta]_\approx \ t$$

From the complete set of forking diagrams for the (lbeta)-reduction it may be inferred that the $[\cdot]_\approx$-relation holds:

Corollary 4.1.15. *Let $r, \lambda x.s \in \Lambda_\approx^0$. Then $(\lambda x.s)\, r \ [\emptyset]_\approx \ \mathtt{let}\ x = r \ \mathtt{in}\ s$ holds.*

Proof. By the approximation reduction $(\lambda x.s)\, r \xrightarrow{\ [\ ],\ \mathrm{lbeta}\ } \mathtt{let}\ x = r \ \mathtt{in}\ s$ in conjunction with corollary 3.2.16, both terms have the same answer set, i.e., $\mathtt{ans}((\lambda x.s)\, r) = \mathtt{ans}(\mathtt{let}\ x = r \ \mathtt{in}\ s)$ holds. $\qquad\square$

Note that the opposite relation $\mathtt{let}\ x = r \ \mathtt{in}\ s \ [\emptyset]_\approx \ (\lambda x.s)\, r$ is already true by corollary 4.1.12.

Corollary 4.1.16. *Let $r, \lambda x.s, \lambda x.t \in \Lambda_\approx^0$ be closed. Then for all $\eta \subseteq \Lambda_\approx \times \Lambda_\approx$:*

$$\mathtt{let}\ x = r \ \mathtt{in}\ s \ [\eta]_\approx \ \mathtt{let}\ x = r \ \mathtt{in}\ t \quad \Longleftrightarrow \quad (\lambda x.s)\, r \ [\eta]_\approx \ (\lambda x.t)\, r$$

Proof. Suppose $\mathtt{let}\ x = r \ \mathtt{in}\ s \ [\eta]_\approx \ \mathtt{let}\ x = r \ \mathtt{in}\ t$ for the "only-if"-part. Then

$$(\lambda x.s)\, r \ [\eta]_\approx \ \mathtt{let}\ x = r \ \mathtt{in}\ s \ [\eta]_\approx \ \mathtt{let}\ x = r \ \mathtt{in}\ t \ [\eta]_\approx \ (\lambda x.t)\, r$$

by corollary 4.1.12 and 4.1.15 as $[\eta]_\approx$ is transitive. Same for the "if"-part. $\quad\square$

The conclusion of the preceding corollaries seems natural since in call-by-need calculi like [AF97] without an explicit $\mathtt{let}$, terms of the form $(\lambda x.s)\, r$ represent a $\mathtt{let}$-binding.

By the complete set of forking diagrams for (cpa) a similar result can be obtained for terms related by a (cpa)-reduction. However, section 3.2.2 shows that internal (stop)-reduction might be introduced. Hence this requires a treatment of internal (stop)-reductions first, which is easier in the context of simulations and similarity. These notions are given for the $\boldsymbol{\lambda}_\approx$-calculus here.

Definition 4.1.17. *Let $\eta \subseteq \Lambda_\approx^0 \times \Lambda_\approx^0$ be a preorder on closed terms. Then η is called a* simulation, *if and only if it is $[\cdot^o]_\approx$-dense, i.e. $\eta \subseteq [\eta^o]_\approx$ holds.*

As already known from section 2.4, the operator $[\cdot^o]_\approx$ is monotonic w.r.t. set inclusion, hence its greatest fixed point exists by the fixed point theorem.

Definition 4.1.18. *Define* similarity $\precsim_b = \mathrm{gfp}([\cdot^o]_{\approx})$ *to be the greatest fixed point of* $[\cdot^o]_{\approx}$ *and* mutual similarity $\simeq_b$ *by* $s \simeq_b t \iff s \precsim_b t \,\wedge\, t \precsim_b s$.

The following are either obvious consequences from the definitions or their proof can be found in the literature, e.g. [DP92].

Corollary 4.1.19. *The relation* $\precsim_b$ *is a preorder and* $\simeq_b$ *an equivalence.*

Corollary 4.1.20. *The similarity* $\precsim_b$ *is the greatest* $[\cdot^o]_{\approx}$*-dense set, i.e., it is itself* $[\cdot^o]_{\approx}$*-dense and can be characterised as the union of all* $[\cdot^o]_{\approx}$*-dense sets:*

$$\precsim_b = \bigcup \{\eta \mid \eta \subseteq [\eta^o]_{\approx}\}$$

Because of this, also $[\precsim_b{}^o]_{\approx}$ is contained in similarity: From $\precsim_b \subseteq [\precsim_b{}^o]_{\approx}$ follows that $[\precsim_b{}^o]_{\approx} \subseteq \big[[\precsim_b{}^o]_{\approx}{}^o\big]_{\approx}$ since $[\cdot^o]_{\approx}$ is monotonic. Hence $\precsim_b = [\precsim_b{}^o]_{\approx}$ and similarity may be represented in the recursive manner below.

$$s \;\precsim_b\; t \iff \forall \lambda x.s' \in \mathtt{ans}(s) : \exists \lambda x.t' \in \mathtt{ans}(t) : \; s' \precsim_b{}^o t' \qquad (4.1.3)$$

An example shows that $\boldsymbol{\lambda}_{\approx}$ has, w.r.t. $\precsim_b$, incomparable, closed abstractions.

Example 4.1.21. *Consider the combinators* $\mathbf{K}$ *and* $\mathbf{K2}$*, which both clearly are abstractions. If applied, e.g., to the argument* $\mathbf{I}$ *we yield*

$$\mathbf{K}\,\mathbf{I} \xrightarrow{lbeta} \mathtt{let}\ x = \mathbf{I}\ \mathtt{in}\ \lambda y.x \xrightarrow{cpa} \lambda y.\mathbf{I}$$

$$\mathbf{K2}\,\mathbf{I} \xrightarrow{lbeta} \mathtt{let}\ x = \mathbf{I}\ \mathtt{in}\ \lambda y.y \xrightarrow{cpa} \lambda y.y$$

Hence both of these reducts are again abstractions. But if applied to $\boldsymbol{\Omega}$ *as a further argument, the difference becomes apparent:*

$$(\lambda y.\mathbf{I})\,\boldsymbol{\Omega} \xrightarrow{lbeta} \mathtt{let}\ y = \boldsymbol{\Omega}\ \mathtt{in}\ \mathbf{I} \xrightarrow{cpa} \mathbf{I}$$

$$(\lambda y.y)\,\boldsymbol{\Omega} \xrightarrow{lbeta} \mathtt{let}\ y = \boldsymbol{\Omega}\ \mathtt{in}\ y \to \ldots$$

Obviously, the term $\mathtt{let}\ y = \boldsymbol{\Omega}\ \mathtt{in}\ y$ *has no approximation to an abstraction, thus* $\mathbf{K} \not\precsim_b \mathbf{K2}$*. With a similar argument — just applying first to* $\boldsymbol{\Omega}$ *and then to* $\mathbf{I}$ *— one can show that* $\mathbf{K2} \not\precsim_b \mathbf{K}$ *holds.*

As a further motivating example, we demonstrate that similarity as defined in the $\boldsymbol{\lambda}_{\approx}$-calculus avoids the pitfalls of the naive approach.

Example 4.1.22. *As is known, the two terms $s \equiv$ let $v =$ pick $\mathbf{K}$ $\mathbf{K2}$ in $\lambda w.v$ and $t \equiv \lambda w.$let $v =$ pick $\mathbf{K}$ $\mathbf{K2}$ in v of example 3.1.18 could be distinguished by contexts. Now we can show that $t \not\lesssim_b s$ does not hold either. Since t already is an abstraction, we therefore consider all possible approximation reduction sequences for s that lead to an abstraction:*

$$s \xrightarrow{\text{let } v=[\,] \text{ in } \lambda w.v, \ nd, \ left} \text{let } v = \mathbf{K} \text{ in } \lambda w.v \xrightarrow{[\,], \ cpa} \lambda w.\mathbf{K}$$

$$s \xrightarrow{\text{let } v=[\,] \text{ in } \lambda w.v, \ nd, \ right} \text{let } v = \mathbf{K2} \text{ in } \lambda w.v \xrightarrow{[\,], \ cpa} \lambda w.\mathbf{K2}$$

Since the non-deterministic choice has been fixed, neither of these abstractions exposes the necessary behaviour. Particularly, t may converge when applied to the argument sequences $\boldsymbol{\Omega}, \boldsymbol{\Omega}, \mathbf{K}$ and $\boldsymbol{\Omega}, \mathbf{K}, \boldsymbol{\Omega}$, while $\lambda w.\mathbf{K}$ does not converge for the former, nor does $\lambda w.\mathbf{K2}$ for the latter.

Especially useful is the result, that (stop)-reductions in general lead to terms that are smaller w.r.t. similarity. At the same time the following lemma gives a nice example of a coinductive proof, exploiting the fact that $\lesssim_b$ as a greatest fixed point of $[\cdot^o]_{\approx}$ contains every $[\cdot^o]_{\approx}$-dense set.

Lemma 4.1.23. *Let $s, t \in \Lambda^0_{\approx}$ be closed. Then $s \xrightarrow{\mathcal{C}, \ stop}{}^* t$ implies $t \lesssim_b s$.*

Proof. Since $\lesssim_b$ is the union of all $[\cdot^o]_{\approx}$-dense sets, we will simply show that the set $\nu = \{\,(t,s) \in \Lambda^0_{\approx} \times \Lambda^0_{\approx} \mid s \xrightarrow{\mathcal{C}, \ stop}{}^* t\,\}$ is $[\cdot^o]_{\approx}$-dense, i.e., $\nu \subseteq [\nu^o]_{\approx}$ holds. So assume $s \xrightarrow{\mathcal{C}, \ stop}{}^* t$ such that $t \Downarrow \lambda x.t'$. Then by lemma 3.2.11, there is also a reduction $s \xrightarrow{\mathcal{S}}{}^*_{\boldsymbol{\lambda}_{\approx}} \lambda x.s'$ such that $\lambda x.s' \xrightarrow{\mathcal{C}, \ stop}{}^* \lambda x.t'$ holds. Hence for every pseudo-valued substitution σ we obviously have $\sigma(s') \xrightarrow{\mathcal{C}, \ stop}{}^* \sigma(t')$ too. Therefore, by $(\sigma(t'), \sigma(s')) \in \nu$ the claim is shown. $\qquad\square$

The bottom line of the next lemma, that reduction by rule (cpa) complies with mutual similarity, is integral for arguing within the $\boldsymbol{\lambda}_{\approx}$-calculus. In order to prove this, we will fall back upon the previous lemma.

Lemma 4.1.24. *If* let $x = s$ in $t \xrightarrow{\mathcal{S}, \ cpa} t[s/x]$ *then* let $x = s$ in $t \simeq_b t[s/x]$.

Proof. Since $t[s/x] \lesssim_b$ let $x = s$ in t is clear by corollary 4.1.12, we only have to show let $x = s$ in $t \lesssim_b t[s/x]$, i.e. that $\xrightarrow{\mathcal{S}, \ cpa}$ does not change the result of a $\xrightarrow{\mathcal{S}}{}^k_{\boldsymbol{\lambda}_{\approx}}$-reduction sequence.

By corollary 4.1.14, it is sufficient to show that for every reduction sequence $\mathtt{let}\ x = s\ \mathtt{in}\ t \xrightarrow{\;\mathcal{S}\;}{}^*_{\boldsymbol{\lambda}_{\approx}} \lambda y.p$ there is a corresponding reduction sequence for $t[s/x]$, i.e., $t[s/x] \xrightarrow{\;\mathcal{S}\;}{}^*_{\boldsymbol{\lambda}_{\approx}} \lambda z.q$ which satisfies $\lambda y.p \lesssim_b \lambda z.q$. By lemma 3.2.12 there is an approximation reduction $t[s/x] \xrightarrow{\;\mathcal{S}\;}{}^*_{\boldsymbol{\lambda}_{\approx}} \lambda z.q$ such that $\lambda z.q \xrightarrow{\;i,\ \mathrm{stop}\;}{}^* \lambda y.p$ holds. Thus by lemma 4.1.23 the claim is shown. $\qquad\square$

Since pseudo-values are just those which are copied by(cpa) anyway, there is no difference between a $\mathcal{PV}$-substitution and the corresponding $\mathtt{let}$-environment.

Corollary 4.1.25. *Let $s \in \Lambda_{\approx}$ be an open term with a single free variable x, i.e., $\mathcal{FV}(s) = \{x\}$ holds, and let $p \in \mathcal{PV}$ be a pseudo-value, i.e., $p \equiv \circledcirc$ or $p \equiv \lambda z.q$ for some closed abstraction. Then $\mathtt{let}\ x = p\ \mathtt{in}\ s \simeq_b s[p/x]$ is true.*

Proof. By lemma 4.1.24, since reduction by rule (cpa) immediately applies. $\quad\square$

So far, we have a correlation between pseudo-valued substitutions and $\mathtt{let}$-bindings of pseudo-values as well as, by corollary 4.1.15 and 4.1.16 respectively, one between $\mathtt{let}$-terms and λ-applications. To fill in the gap, what we need is simply a connection between $\mathtt{let}$-bindings of arbitrary terms and pseudo-values.

Lemma 4.1.26. *Let $s, t \in \Lambda_{\approx}$ be terms. Then the following implication is true.*

$$(\forall r \in \mathcal{PV}:\ \mathtt{let}\ x = r\ \mathtt{in}\ s \lesssim_b \mathtt{let}\ x = r\ \mathtt{in}\ t) \implies$$
$$\left(\forall p \in \Lambda_{\approx}^0:\ \mathtt{let}\ x = p\ \mathtt{in}\ s \lesssim_b \mathtt{let}\ x = p\ \mathtt{in}\ t\right)$$

Proof. Let $p \in \Lambda_{\approx}^0$ be an arbitrary but fixed closed term. Since otherwise nothing has to be shown, we assume that $\mathtt{let}\ x = p\ \mathtt{in}\ s$ converges. Legitimated by lemma 4.1.13, we will now prove that for every approximation reduction sequence $\mathtt{let}\ x = p\ \mathtt{in}\ s \xrightarrow{\;\mathcal{S}\;}{}^*_{\boldsymbol{\lambda}_{\approx}} \lambda z.q$ yielding an abstraction, there will be an intermediate term $s' \in \Lambda_{\approx}$ with $\mathtt{let}\ x = p\ \mathtt{in}\ s \xrightarrow{\;\mathcal{S}\;}{}^*_{\boldsymbol{\lambda}_{\approx}} s' \xrightarrow{\;\mathcal{S}\;}{}^*_{\boldsymbol{\lambda}_{\approx}} \lambda z.q$ and a corresponding reduction sequence for $\mathtt{let}\ x = p\ \mathtt{in}\ t \xrightarrow{\;\mathcal{S}\;}{}^*_{\boldsymbol{\lambda}_{\approx}} t'$ such that $s' \lesssim_b t'$ holds. So assuming $\mathtt{let}\ x = p\ \mathtt{in}\ s \xrightarrow{\;\mathcal{S}\;}{}^*_{\boldsymbol{\lambda}_{\approx}} \lambda z.q$ arbitrary but fixed, by the standardisation lemma 3.2.26 this converging reduction sequence may be reordered to

$$\mathtt{let}\ x = p\ \mathtt{in}\ s \xrightarrow{\;\mathtt{let}\ x = \mathcal{S}\ \mathtt{in}\ s\;}{}^*_{\boldsymbol{\lambda}_{\approx}} \mathtt{let}\ x = p'\ \mathtt{in}\ s \xrightarrow{\;\mathcal{S}\;}{}^*_{\boldsymbol{\lambda}_{\approx}} \lambda z.q$$

for p' being $\odot$ or an abstraction, i.e., a pseudo-value. Obviously, this reduction sequence is also possible for $\mathtt{let}\ x = p\ \mathtt{in}\ t$, thus we have

$$\mathtt{let}\ x = p\ \mathtt{in}\ t \xrightarrow[\boldsymbol{\lambda}_{\approx}]{\ \mathtt{let}\ x=\mathcal{S}\ \mathtt{in}\ s\ }{}^{*} \mathtt{let}\ x = p'\ \mathtt{in}\ t$$

and in conjunction with lemma 4.1.13 the claim holds by the premise. $\qquad\square$

We are now in a position to show that the concepts discussed above, i.e., instantiation with pseudo-values, $\mathtt{let}$-bindings to pseudo-values or arbitrary terms and application to arguments, are all interchangeable.

Proposition 4.1.27. *Let $s, t \in \Lambda_{\approx}$ be terms with $\mathcal{FV}(s) \cup \mathcal{FV}(t) = \{x\}$, i.e., containing only a single free variable. Then the following are equivalent:*

1. *$\forall \sigma :\ \boldsymbol{rng}(\sigma) \subseteq \mathcal{PV} \implies \sigma(s)\ \lesssim_b\ \sigma(t)$*

2. *$\forall p \in \mathcal{PV} :\ (\lambda x.s)\, p\ \lesssim_b\ (\lambda x.t)\, p$*

3. *$\forall p \in \mathcal{PV} :\ \mathtt{let}\ x = p\ \mathtt{in}\ s\ \lesssim_b\ \mathtt{let}\ x = p\ \mathtt{in}\ t$*

4. *$\forall p \in \Lambda_{\approx}^0 :\ (\lambda x.s)\, p\ \lesssim_b\ (\lambda x.t)\, p$*

5. *$\forall p \in \Lambda_{\approx}^0 :\ \mathtt{let}\ x = p\ \mathtt{in}\ s\ \lesssim_b\ \mathtt{let}\ x = p\ \mathtt{in}\ t$*

Proof. The equivalence (1) $\iff$ (3) is by corollary 4.1.25 while (5) $\implies$ (3) and (4) $\implies$ (2) are trivial. From lemma 4.1.26 the implication (3) $\implies$ (5) may be concluded. Corollary 4.1.16 establishes the equivalence (2) $\iff$ (3) as well as (4) $\iff$ (5) and thus the proposition. $\qquad\square$

Since for a closed abstraction $\lambda x.s'$ the subterm s' contains only the single free variable x, this proposition discloses a whole series of representations of similarity. Each of these arises from (4.1.3) by replacing the condition $s' \lesssim_b^{\,o} t'$ with one of those from the above proposition. Particularly worth mentioning is the style of what Abramsky calls *applicative bisimulation* in [Abr90].

$$s\ \lesssim_b\ t \iff \forall \lambda x.s' \in \mathtt{ans}(s) :\ \exists \lambda x.t' \in \mathtt{ans}(t) :$$
$$\forall p \in \Lambda_{\approx}^0 :\ (\lambda x.s')\, p\ \lesssim_b\ (\lambda x.t')\, p \quad (4.1.4)$$

Of specific interest is also the variant which explicitly expresses sharing:

$$s\ \lesssim_b\ t \iff \forall \lambda x.s' \in \mathtt{ans}(s) :\ \exists \lambda x.t' \in \mathtt{ans}(t) :$$
$$\forall p \in \Lambda_{\approx}^0 :\ \mathtt{let}\ x = p\ \mathtt{in}\ s'\ \lesssim_b\ \mathtt{let}\ x = p\ \mathtt{in}\ t' \quad (4.1.5)$$

As can be seen from proposition 4.1.27, both of the previous representations exist also in a version only quantifying over pseudo-values instead of all closed terms. In the remainder we will freely make use of these results without further reference, i.e., in each case the most suitable representation will be chosen. It is clear how these different styles could also have been used for defining the experiment $[\cdot]_{\approx}$ and thus similarity as its greatest fixed point.

Furthermore, the claim can be generalised to arbitrary open terms using nested applications and `let`-environments. The proof is a straightforward induction on the number of free variables in the respective terms but will not be carried out here.

Referring to the view of applicative bisimulation it turns out that abstractions are characterised by their applications to arguments.

Corollary 4.1.28. *Let $\lambda x.s, \lambda x.t \in \Lambda_{\approx}^0$ be closed abstractions. Then*

$$\lambda x.s \;\precsim_b\; \lambda x.t \iff \forall p \in \Lambda_{\approx}^0 : \; (\lambda x.s)\,p \;\precsim_b\; (\lambda x.t)\,p \qquad (4.1.6)$$

Proof. By (4.1.4) as $\mathtt{ans}(\lambda x.s) = \{\,\lambda x.s\,\}$ and $\mathtt{ans}(\lambda x.t) = \{\,\lambda x.t\,\}$ holds. □

For arbitrary terms instead of abstractions such a characterisation is not possible. The reason is that in the $\boldsymbol{\lambda}_{\approx}$-calculus *extensionality*, i.e., the proof rule $(\forall r : s\,r \simeq_b t\,r) \implies s \simeq_b t$ is in general not correct. Corresponding to [Abr90, p. 71], already the counter-example $\lambda x.\boldsymbol{\Omega}\,x \precnsim_b \boldsymbol{\Omega}$ demonstrates this. Though, corollary 4.1.28 above provides some limited kind of extensionality.

Remark 4.1.29. *We speak of "limited" extensionality since in our non-deterministic setting we don't even have the* conditional *version of the (η)-rule*

$$s \Downarrow \;\wedge\; x \notin \mathcal{FV}(s) \implies \lambda x.s\,x \simeq_b s$$

like in [Abr90, p. 71]. We give $s \equiv \mathtt{pick}\ \mathbf{K}\ \mathbf{K2}$ as a counter-example here. Obviously, $\lambda x.(s\,x) \Downarrow \lambda x.(s\,x)$ but for s to converge, a choice has to be made in advance, i.e., either $s \Downarrow \mathbf{K}$ or $s \Downarrow \mathbf{K2}$. Since $\mathbf{K}$ and $\mathbf{K2}$ are incomparable as shown in example 4.1.21, neither $\mathbf{K}$ or $\mathbf{K2}$ alone is capable to exhibit the same convergent behaviour as $\lambda x.(s\,x)$ if applied to an argument.

From the characterisation of abstractions in corollary 4.1.28 a further representation of similarity may be deduced. It is remarkable that a similar property does *not* hold for contextual preorder. Though this will not be shown here, it underpins that similarity is strictly contained in contextual preorder.

Corollary 4.1.30. *For all closed terms $s, t \in \Lambda_{\approx}^{0}$ the following is true:*

$$s \lesssim_b t \iff \forall \lambda x.s' \in \mathtt{ans}(s) : \exists \lambda x.t' \in \mathtt{ans}(t) : \lambda x.s' \lesssim_b \lambda x.t' \quad (4.1.7)$$

Proof. Apply equivalence (4.1.6) in equation (4.1.4) to the right hand side. $\square$

Contrary to extensionality the converse property can be established for arbitrary terms. However, this is not trivial as it involves the proof principle from lemma 4.1.13.

Lemma 4.1.31. *Let $s, t \in \Lambda_{\approx}^{0}$ be closed terms such that $s \lesssim_b t$ holds. Then for every closed term $r \in \Lambda_{\approx}^{0}$ also $s\,r \lesssim_b t\,r$ is true.*

Proof. Given $s \lesssim_b t$ and a closed term $r \in \Lambda_{\approx}^{0}$ suppose that $s\,r$ converges. Obviously $s \Downarrow \lambda x.s'$ because[1] otherwise $s\,r \Downarrow$ would not be possible. Since the surface contexts $[\,]\,r$ and $s\,[\,]$ are disjoint, lemma 2.3.14 is applicable. I.e., in the converging approximation reduction for $s\,r$ the reductions on s may successively be shifted forward. Thus for every $s\,r \Downarrow \lambda z.q$ an approximation sequence of the form $s\,r \xrightarrow{[\,]\,r}{}^{*}_{\boldsymbol{\lambda}_{\approx}} (\lambda x.s')\,r \Downarrow \lambda z.q$ can be obtained. Furthermore, from $s \lesssim_b t$ and $s \Downarrow \lambda x.s'$ it follows that $t \Downarrow \lambda x.t'$ for a closed abstraction $\lambda x.t'$ such that the condition $\forall p \in \Lambda_{\approx}^{0} : (\lambda x.s')\,p \lesssim_b (\lambda x.t')\,p$ holds. Since $t\,r \xrightarrow{[\,]\,r}{}^{*}_{\boldsymbol{\lambda}_{\approx}} (\lambda x.t')\,r$ an application of lemma 4.1.13 proves the claim. $\square$

We conclude the section with a more sophisticated simulation proof using the fact that $\lesssim_b$ contains every $[\cdot]_{\approx}$-dense set.

Example 4.1.32 (Simulation Proof). *Let $r, s, t \in \Lambda_{\approx}^{0}$ be closed terms such that $r \lesssim_b \mathtt{pick}\ s\ t$ and the set $\mathtt{ans}(r)$ has, w.r.t. $\lesssim_b$, one greatest element. Then $r \lesssim_b s$ or $r \lesssim_b t$ holds.*

First note that $r \lesssim_b \mathtt{pick}\ s\ t$ implies that there must exist a $[\cdot^o]_{\approx}$-dense set η which contains $(r, \mathtt{pick}\ s\ t)$, i.e., $\eta \subseteq [\eta^o]_{\approx}$ and $(r, \mathtt{pick}\ s\ t) \in \eta$, hence $(r, \mathtt{pick}\ s\ t) \in [\eta^o]_{\approx}$ as well. Since the union of $[\cdot^o]_{\approx}$-dense sets again is $[\cdot^o]_{\approx}$-dense, it suffices to show that ν or υ, with $\nu = \eta \cup \{(r, s)\}$ and $\upsilon = \eta \cup \{(r, t)\}$ respectively, is $[\cdot^o]_{\approx}$-dense.

Obviously, for $r \nDownarrow$ there is nothing to show, so we assume $\lambda y.p \in \Lambda_{\approx}^{0}$ to be the $\lesssim_b$-greatest closed abstraction such that $r \Downarrow \lambda y.p$ holds. From the prerequisite $r\ [\eta^o]_{\approx}\ \mathtt{pick}\ s\ t$ we have an abstraction $\lambda z.q$ such that $\mathtt{pick}\ s\ t \Downarrow \lambda z.q$ and $y.p\ \eta^o\ z.q$ are satisfied. Note that this $\lambda z.q$ usually depends on $\lambda y.p$ but under the premise that $\lambda y.p$ is greater than every closed abstraction r converges to, we may

---

[1]This statement corresponds to the contraposition of lemma 3.2.14

fix $\lambda z.q$ *here. So in the approximation reduction sequence* `pick` $s\ t \xrightarrow{\mathcal{S}}{}^*_{\boldsymbol{\lambda}_\approx} \lambda z.q$ *the reduction by rule* (nd) *could be moved forward, hence we can argue that* $s \Downarrow \lambda z.q$ *or* $t \Downarrow \lambda z.q$ *must hold.*

   Since these cases behave symmetrically, we assume $s \Downarrow \lambda z.q$ *w.l.o.g., for which we will show* $\nu \subseteq [\nu^o]_\approx$, *i.e.,* $(a,b) \in \nu \implies (a,b) \in [\nu^o]_\approx$, *for which we distinguish the two cases:*

- *For* $(a,b) \in \eta$ *nothing has to be shown since* η *is* $[\cdot^o]_\approx$-*dense.*

- *If* $(a,b) \equiv (r,s)$, *we know from what has been said before that only* $r \Downarrow \lambda y.p$ *has to be considered. Hence* $s \Downarrow \lambda z.q$ *with* $y.p\ \eta^o\ z.q$ *shows the claim, since* $[\eta^o]_\approx \subseteq [\nu^o]_\approx$ *by monotonicity of the* $[\cdot^o]_\approx$-*operator.*

4.1.3   Open Simulations and Open Similarity

When the relation $s \precsim_b{}^o t$ should be established for two open terms s and t it seems rather tedious to consider all pseudo-valued closing substitutions. This is particularly true for the use of reductions on open terms.

   So, is there an alternative for relations which are defined on open terms? The question leads to the notion below. However, this section will illustrate that its answer is not quite as satisfactory as expected.

Definition 4.1.33. *A preorder* $\eta \subseteq \Lambda_\approx \times \Lambda_\approx$ *on (possibly open) terms is called an* open simulation *if and only if* $\eta \subseteq [\eta]_\approx{}^o$ *is satisfied.*

   The relation $\precsim_b{}^o$ is an open simulation: $\precsim_b \subseteq [\precsim_b{}^o]_\approx$ implies $\precsim_b{}^o \subseteq [\precsim_b{}^o]_\approx{}^o$ since $(\cdot)^o$ is monotone. It even is the largest open simulation and thus called *open similarity*. In order to see that *every* open simulation is contained in $\precsim_b{}^o$ consider $\eta \subseteq [\eta]_\approx{}^o$ from which $[\eta]_\approx \subseteq [[\eta]_\approx{}^o]_\approx$ follows because $[\cdot]_\approx$ is monotone. Thus $[\eta]_\approx$ is a simulation and contained in similarity. Furthermore $[\eta]_\approx{}^o \subseteq \precsim_b{}^o$ since the open extension $(\cdot)^o$ is monotone. Therefore $\eta \subseteq [\eta]_\approx{}^o \subseteq \precsim_b{}^o$ as desired.

   Accordingly, the relation $\simeq_b{}^o = \precsim_b{}^o \cap \succsim_b{}^o$ is called *mutual open similarity*. Some of the results for closed terms transfer directly to open similarity.

Corollary 4.1.34. *For every closed term* $s \in \Lambda_\approx^0$ *the relation* $\circledcirc \precsim_b s$ *holds and for every (possibly open) term* $t \in \Lambda_\approx$ *the statement* $\circledcirc \precsim_b{}^o t$ *is true.*

Proof. Since $\mathrm{ans}(\circledcirc) = \emptyset$ and $\circledcirc$ is closed, i.e., $\sigma(\circledcirc) \equiv \circledcirc$ for all substitutions.   □

Corollary 4.1.35. *Let* $s,t \in \Lambda_\approx$ *be terms such that* $s \precsim_b{}^o t$ *holds. Then for every term* $r \in \Lambda_\approx$ *also* $s\,r \precsim_b{}^o t\,r$ *is valid.*

Proof. Assume terms $s, t \in \Lambda_\approx$ such that $s \lesssim_b^o t$ is true. Let σ be an arbitrary substitution with $\boldsymbol{rng}(\sigma) \subseteq \mathcal{PV}$ such that $\sigma(s\,r)$ is closed. For $s\,r \lesssim_b^o t\,r$ it is to show that $\sigma(s\,r) \lesssim_b \sigma(t\,r)$ holds. This is equivalent to $\sigma(s)\,\sigma(r) \lesssim_b \sigma(t)\,\sigma(r)$ while $\sigma(s) \lesssim_b \sigma(t)$ from the premise. Thus by lemma 4.1.31 the claim holds. $\square$

The notion of an open simulation does not automatically provide a more straightforward proof technique. The reason is that in order to establish some relation η to be an open simulation, the condition $\eta \subseteq [\eta]_\approx^o$ has to be checked. This requires $s\,\eta\,t \implies \big(\forall \sigma : \boldsymbol{rng}(\sigma) \subseteq \mathcal{PV} \implies \sigma(s)\ [\eta]_\approx\ \sigma(t)\big)$ to be shown. Note that this does not necessarily involve all closing substitutions that map only to $\odot$ or closed abstraction but rather only those for which $\sigma(s)$ converges.

In the following it will be discussed briefly when and under what preconditions the convergent behaviours of s and $\sigma(s)$ are connected.

Lemma 4.1.36. *Let $s \in \Lambda_\approx$ be a term. Then $s \Downarrow \lambda z.q$ implies $\sigma(s) \Downarrow \sigma(\lambda z.q)$ for all closing substitutions σ that map only to $\odot$ or closed abstractions.*

Proof. Note that the proposition of the lemma is identical to

> Let $s \in \Lambda_\approx$ be a (possibly open) term and σ a pseudo-valued closing substitution. Then $s \Downarrow \lambda z.q$ implies that also $\sigma(s) \Downarrow \sigma(\lambda z.q)$ holds.

Hence assume a substitution σ which meets the prerequisites and a term $s \in \Lambda_\approx$ such that $s \Downarrow \lambda z.q$ holds. The proof is then by induction on the length k of the converging approximation reduction $s \xrightarrow{\mathcal{S}\ k}_{\boldsymbol{\lambda}_\approx} \lambda z.q$ leading to the abstraction.

- If $k = 0$ then $s \equiv \lambda z.q$ already is an abstraction and $\sigma(s) \equiv \sigma(\lambda z.q)$ holds.

- A reduction sequence of length $k \geq 1$ can be split:

$$s \xrightarrow{\mathcal{S}}_{\boldsymbol{\lambda}_\approx} t \xrightarrow{\mathcal{S}\ k-1}_{\boldsymbol{\lambda}_\approx} \lambda z.q$$

  By the induction hypothesis we may assume $\sigma(t) \Downarrow \sigma(\lambda z.q)$ so only the reduction $\sigma(s) \xrightarrow{\mathcal{S}}_{\boldsymbol{\lambda}_\approx} \sigma(t)$ remains to be established. This reduction is obvious for the rules (lapp), (lbeta), (lseq), (eseq), (nd) as well as (stop). In order to address the rule (cpa) in more detail, consider

$$s \equiv S[\texttt{let } x = s_1 \texttt{ in } s_2] \xrightarrow{S,\ \text{cpa}} S[s_2[s_1/x]] \equiv t$$

  where s_1 is $\odot$ or an abstraction. Thus, applying the substitution σ yields

$$\sigma(s) \equiv \sigma(S)[\texttt{let } x = \sigma(s_1) \texttt{ in } \sigma(s_2)]$$
$$\xrightarrow{\sigma(S),\ \text{cpa}} \sigma(S)[\sigma(s_2)[\sigma(s_1)/x]] \equiv \sigma(t)$$

which proves the claim. $\square$

The proof of the converse property will make use of the substitution mapping all (free) variables to the ⊙-term. Denoted by $\sigma_\odot$, this substitution represents a tool for expressing that the outcome of an approximation reduction on an open term does not (essentially) depend on free variables. Otherwise, the reduction of the substituted term would not lead to an abstraction since ⊙ has no approximation reduction.

Proposition 4.1.37. *Let $s \in \Lambda_\approx$ be a (possibly open) term and σ range over closing substitutions which only map to ⊙ or an abstraction. Then the following statements are equivalent:*

1. *$s \Downarrow$*

2. *$\forall \sigma : \ \sigma(s) \Downarrow$*

3. *$\sigma_\odot(s) \Downarrow$*

Proof. Since "(1) $\implies$ (2)" is by lemma 4.1.36 and "(2) $\implies$ (3)" is trivial, only the implication "(3) $\implies$ (1)" has to be shown. So assume $\sigma_\odot(s) \Downarrow$ which means $\sigma_\odot(s) \xrightarrow{\mathcal{S}}{}^*_{\boldsymbol{\lambda}_\approx} \lambda y.p$ for some abstraction. The term $\sigma_\odot(s)$ can obviously be obtained from s by certain, possibly internal, (stop)-reductions. Hence a reduction sequence $s \xrightarrow{\mathcal{C},\, \text{stop}}{}^* \sigma_\odot(s) \xrightarrow{\mathcal{S}}{}^*_{\boldsymbol{\lambda}_\approx} \lambda y.p$ results. Thus by lemma 3.2.11 there is an approximation reduction $s \xrightarrow{\mathcal{S}}{}^*_{\boldsymbol{\lambda}_\approx} \lambda z.q$ to an abstraction $\lambda z.q$ such that $\lambda z.q \xrightarrow{i,\, \text{stop}}{}^* \lambda y.p$ holds. $\square$

The consequence of this proposition is a necessary but not sufficient criterion for open simulations, e.g. open similarity: A preorder $\eta \subseteq \Lambda_\approx \times \Lambda_\approx$ is an open simulation *only* if it meets (4.1.8) below.

$$s \, \eta \, t \implies (\forall \lambda x.s' \in \mathtt{ans}(s) : \ \exists \lambda x.t' \in \mathtt{ans}(t) : \ s' \lesssim_b{}^o t') \qquad (4.1.8)$$

So in the particular instance for $\lesssim_b{}^o$ it can be used to show that two open terms are *not* related. This is especially true if $s, t \in \Lambda_\approx$ are two open terms such that s converges but t does not. Since $s \Downarrow \implies (\forall \sigma : \ \boldsymbol{rng}(\sigma) \subseteq \mathcal{PV} \implies \sigma(s) \Downarrow)$ and $t \not\Downarrow \implies \sigma_\odot(t) \not\Downarrow$ by proposition 4.1.37, clearly $s \not\lesssim_b{}^o t$ must hold.

4.1.4 Soundness of (cp), (llet) and Two Further Reductions

Of particular interest is certainly the proof that the reduction rules of the calculus $\boldsymbol{\lambda}_{\mathrm{ND}}$ constitute correct program transformations. Though at this stage open similarity has not been shown to form a precongruence nor to imply contextual equivalence, the achievement of the remaining sections will justify its use here.

This means, we will now discuss that some of the rules from the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus as well as two further rules comply to open similarity. Therefore, a notion for the soundness of reductions, i.e., program transformations in fact, is introduced first. Note that it is sufficient to consider transformations on top-level only, because in section 4.3 open similarity will be proven a precongruence. The argument accords to the one in [Kut99, Proposition 3.1.2] but is more powerful as it refers to open similarity rather than only convergence. The reason is that establishing contextual equivalence already involves the converging behaviour in all contexts. Thus it makes essentially no difference if the implication $s \xrightarrow{[\ ],\ a} t \implies s \simeq_c t$ or $C[s] \xrightarrow{C,\ a} C[t] \implies C[s] \simeq_c C[t]$ is shown.

Definition 4.1.38. *Let $\eta \subseteq \Lambda_{\approx} \times \Lambda_{\approx}$ be a preorder and $\nu \subseteq \Lambda_{\approx} \times \Lambda_{\approx}$ be a program transformation. Then ν is said to be* sound w.r.t. η *if $s\ \nu\ t$ implies $s\ \eta\ t$ for all terms $s, t \in \Lambda_{\approx}$.*

Soundness of a program transformation can be carried over from closed terms to open terms if it is closed under pseudo-valued substitutions.

Corollary 4.1.39. *Let $\xrightarrow{a} \subseteq \Lambda_{\approx} \times \Lambda_{\approx}$ be some transformation that is is sound w.r.t. $\lesssim_b$ on closed terms and closed under substitutions σ with $\boldsymbol{rng}(\sigma) \subseteq \mathcal{PV}$. Then $\xrightarrow{a}$ is also sound w.r.t. $\lesssim_b^o$ on open terms.*

Proof. Let $s, t \in \Lambda_{\approx}$ be open terms such that $s \xrightarrow{a} t$ and assume $\xrightarrow{a}$ to be sound w.r.t. $\lesssim_b$. Since $\xrightarrow{a}$ is closed under substitutions that only map to $\circledcirc$ or abstractions, $\forall \sigma :\ \boldsymbol{rng}(\sigma) \subseteq \mathcal{PV} \implies \sigma(s) \xrightarrow{a} \sigma(t)$ holds. Therefore $\forall \sigma :\ \boldsymbol{rng}(\sigma) \subseteq \mathcal{PV} \implies \sigma(s)\ \lesssim_b\ \sigma(t)$ shows the claim. $\qquad\square$

In the previous corollary, the symbol $\xrightarrow{a}$ has been used to indicate a program transformation which is based on some kind of rewriting. The motivation for this is that reduction in the calculi $\boldsymbol{\lambda}_{\mathrm{ND}}$ and $\boldsymbol{\lambda}_{\approx}$ is closed even under arbitrary substitutions. However, this is not to be expected for program transformations in general as the example $y\,y \xrightarrow{a} \texttt{let } x = y \texttt{ in } x\,x$ illustrates.

Corollary 4.1.40. *Let $s, t \in \Lambda_{\approx}$ be terms. Then $s \xrightarrow{\ \mathcal{S}\ *}_{\boldsymbol{\lambda}_{\approx}} t$ implies $t \lesssim_b^o s$.*

Proof. By corollary 4.1.12 in connection with corollary 4.1.39.   □

Additionally, the soundness of the rules (cpa) and (lbeta) w.r.t. mutual open similarity is inferred from the respective results in lemma 4.1.24 and corollary 4.1.15 for closed terms.

Corollary 4.1.41. *Let $s, t \in \Lambda_{\approx}$ be terms such that the reduction $s \xrightarrow{\mathcal{S}, \ cpa} t$ or $s \xrightarrow{\mathcal{S}, \ lbeta} t$ holds. Then $s \simeq_b{}^o t$ is true.*

Out of the rules of the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus, the rule (lbeta) will not be treated here, since it is also a $\boldsymbol{\lambda}_{\approx}$-rule and has already been addressed above. Furthermore, (lapp) will be omitted, since a complete set of forking diagrams is analogous to the one for (lbeta) in section 3.2.4. As explained before, cf. corollary 3.2.19, the reduction rules (eseq) and (lseq) resemble (lbeta) and (lapp) respectively, and thus are left out, too.

Since non-deterministic choice does clearly *not* preserve contextual equivalence, only the reduction rules (llet) and (cp) remain. These are quite interesting as they are not contained in the $\boldsymbol{\lambda}_{\approx}$-calculus. Moreover, we introduce two further rules which may be viewed as program transformations and prove that these also comply with mutual open similarity. The two additional rules comprise the so-called *garbage collection* rule (gc) which deletes a let-binding when the variable is no longer referenced. In [Kut99] it is called (ldel). The second, named (cpx) contrary to (lcv) in [Kut99], concerns copying of variables. For completeness (llet) and (cp) are listed again.

$$\texttt{let } x = s \texttt{ in } t \xrightarrow{gc} t \quad \text{if } x \notin \mathcal{FV}(t) \tag{gc}$$

$$\texttt{let } x = y \texttt{ in } D[x] \xrightarrow{cpx} \texttt{let } x = y \texttt{ in } D[y] \tag{cpx}$$

$$\texttt{let } x = (\texttt{let } y = t_y \texttt{ in } t_x) \texttt{ in } s \xrightarrow{llet} \texttt{let } y = t_y \texttt{ in } (\texttt{let } x = t_x \texttt{ in } s) \tag{llet}$$

$$\texttt{let } x = \lambda y.r \texttt{ in } D[x] \xrightarrow{cp} \texttt{let } x = \lambda y.r \texttt{ in } D[\lambda y.r] \tag{cp}$$

Note that here the two occurrences of y in $\texttt{let } x = y \texttt{ in } D[y]$ denote the *same* variable since y occurs free in $\texttt{let } x = y \texttt{ in } D[x]$ and therefore no alpha-renaming takes place during a (cpx)-reduction.

We begin with (cp) since its soundness can be deduced from (cpa) relatively straightforward.

Corollary 4.1.42. *Let $\texttt{let } x = \lambda y.r \texttt{ in } D[x] \xrightarrow{cp} \texttt{let } x = \lambda y.r \texttt{ in } D[\lambda y.r]$ then $\texttt{let } x = \lambda y.r \texttt{ in } D[x] \simeq_b{}^o \texttt{let } x = \lambda y.r \texttt{ in } D[\lambda y.r]$ holds.*

Proof. Using the (cpa)-reductions $\texttt{let } x = \lambda y.r \texttt{ in } D[x] \xrightarrow{\text{cpa}} D[x][\lambda y.r/x]$ and $\texttt{let } x = \lambda y.r \texttt{ in } D[\lambda y.r] \xrightarrow{\text{cpa}} D[\lambda y.r][\lambda y.r/x]$ respectively. These are both sound w.r.t. $\simeq_b^o$ by corollary 4.1.41 and moreover result in syntactically the same term $D[x][\lambda y.r/x] \equiv D[\lambda y.r][\lambda y.r/x]$. $\qquad\square$

With soundness of (cp) w.r.t. mutual open similarity the proof for (cpx) becomes very concise.

Corollary 4.1.43. *Let* $\texttt{let } x = y \texttt{ in } D[x] \xrightarrow{cpx} \texttt{let } x = y \texttt{ in } D[y]$ *then* $\texttt{let } x = y \texttt{ in } D[x] \simeq_b^o \texttt{let } x = y \texttt{ in } D[y]$ *holds.*

Proof. First note that only (mutual) open similarity makes sense here, since y is free. Showing $\texttt{let } x = y \texttt{ in } D[x] \simeq_b^o \texttt{let } x = y \texttt{ in } D[y]$ amounts to

$$\forall \sigma : \ \boldsymbol{rng}(\sigma) \subseteq \mathcal{PV} \implies \sigma(\texttt{let } x = y \texttt{ in } D[x]) \ \simeq_b \ \sigma(\texttt{let } x = y \texttt{ in } D[y])$$

The application of the substitution yields the proof obligation

$$\texttt{let } x = \sigma(y) \texttt{ in } \sigma(C[x]) \ \simeq_b \ \texttt{let } x = \sigma(y) \texttt{ in } \sigma(C[\sigma(y)]) \qquad (4.1.9)$$

W.l.o.g. we may assume that x is not in the domain of σ because of corollary 4.1.4 in combination with the variable convention. Thus, in case of $\sigma(y) \equiv \lambda z.q$ equation (4.1.9) is an instance of rule (cp) while $\sigma(y) \equiv \odot$ is shown similarly. $\quad\square$

For the rule (llet), the equations (3.3.5) and (3.3.6) from the proof of the Approximation Theorem give a hint how the approximation reduction to an abstraction is retained. For simplicity we only perform this for closed terms, but the transfer to open similarity is easy.

Lemma 4.1.44. *For all* $s, t_x, t_y \in \Lambda_\approx$ *such that* $\texttt{let } x = (\texttt{let } y = t_y \texttt{ in } t_x) \texttt{ in } s$ *is closed:* $\texttt{let } x = (\texttt{let } y = t_y \texttt{ in } t_x) \texttt{ in } s \simeq_b \texttt{let } y = t_y \texttt{ in } (\texttt{let } x = t_x \texttt{ in } s)$.

Proof. It will be shown that both terms have the same answer set. Suppose that $\texttt{let } x = (\texttt{let } y = t_y \texttt{ in } t_x) \texttt{ in } s \Downarrow \lambda z.q$, i.e., the first term has a converging approximation reduction. Since $\texttt{let } y = t_y \texttt{ in } t_x \xrightarrow{\text{stop}} \odot$ is even simpler, only the case $\texttt{let } y = t_y \texttt{ in } t_x \Downarrow$ is treated. By two applications of the normalisation

from lemma 3.2.26 the reduction sequence can be reordered to proceed as below.

$$\text{let } x = (\text{let } y = t_y \text{ in } t_x) \text{ in } s$$
$$\xrightarrow[\lambda_\approx]{\text{let } x=(\text{let } y=[\,] \text{ in } t_x) \text{ in } s}{}^{*} \text{let } x = (\text{let } y = t'_y \text{ in } t_x) \text{ in } s$$
$$\xrightarrow[\lambda_\approx]{\text{let } x=[\,] \text{ in } s, \text{ cpa}} \text{let } x = t_x[t'_y/y] \text{ in } s$$
$$\xrightarrow[\lambda_\approx]{\text{let } x=[\,] \text{ in } s}{}^{*} \text{let } x = t'_x \text{ in } s$$
$$\xrightarrow[\lambda_\approx]{[\,], \text{ cpa}} s[t'_x/x] \xrightarrow[\lambda_\approx]{\mathcal{S}}{}^{*} \lambda z.q$$

All these surface reductions are possible for the second term and vice versa, thus we obtain an approximation reduction to exactly the same abstraction:

$$\text{let } y = t_y \text{ in } (\text{let } x = t_x \text{ in } s)$$
$$\xrightarrow[\lambda_\approx]{\text{let } y=[\,] \text{ in } (\text{let } x=t_x \text{ in } s)}{}^{*} \text{let } y = t'_y \text{ in } (\text{let } x = t_x \text{ in } s)$$
$$\xrightarrow[\lambda_\approx]{[\,], \text{ cpa}} (\text{let } x = t_x \text{ in } s)[t'_y/y] \qquad (x \notin \mathcal{FV}(s))$$
$$\equiv \text{let } x = t_x[t'_y/y] \text{ in } s$$
$$\xrightarrow[\lambda_\approx]{\text{let } x=[\,] \text{ in } s}{}^{*} \text{let } x = t'_x \text{ in } s$$
$$\xrightarrow[\lambda_\approx]{[\,], \text{ cpa}} s[t'_x/x] \xrightarrow[\lambda_\approx]{\mathcal{S}}{}^{*} \lambda z.q \qquad\qquad \square$$

Finally, the soundness of rule (gc) enables a result which will be used frequently: Corollary 4.1.46 states that let-bindings for open similarity may be introduced successively and only for the variables affected.

Corollary 4.1.45. *Let* let $x = s$ in $t \xrightarrow{gc} t$ *then* let $x = s$ in $t \simeq_b{}^o t$ *holds.*

Proof. Since it could be simulated by the approximation reduction $\xrightarrow{\text{stop}} \cdot \xrightarrow{\text{cpa}}$ the relation $t \lesssim_b{}^o$ let $x = s$ in t is clear. In order to show let $x = s$ in $t \lesssim_b{}^o t$ we consider a closing substitution σ that maps only to $\odot$ or closed abstractions. Whenever $\sigma(\text{let } x = s \text{ in } t) \Downarrow \lambda z.q$, i.e., has a converging approximation

reduction, then $t \Downarrow \lambda z.q$ by normalisation:

$$\sigma(\texttt{let } x = s \texttt{ in } t)$$

$$\equiv \texttt{let } x = \sigma(s) \texttt{ in } \sigma(t)$$

$$\xrightarrow{\texttt{let } x=[\ ] \texttt{ in } \sigma(s)}_{\boldsymbol{\lambda}_{\approx}} \texttt{let } x = s' \texttt{ in } \sigma(t)$$

$$\xrightarrow{[\ ],\ \text{cpa}}_{\boldsymbol{\lambda}_{\approx}} \sigma(t)[s'/x] \qquad\qquad (x \notin \mathcal{FV}(t))$$

$$\equiv \sigma(t) \xrightarrow{\ \mathcal{S}\ }{}^{*}_{\boldsymbol{\lambda}_{\approx}} \lambda z.q \qquad\qquad \square$$

Corollary 4.1.46. *Let $s, t \in \Lambda_{\approx}$ be terms and $x \in \mathcal{FV}(s) \cup \mathcal{FV}(t)$. Then the following statements are true:*

$$s \lesssim_b{}^o t \iff (\forall p \in \Lambda_{\approx}^0 : \texttt{let } x = p \texttt{ in } s \lesssim_b{}^o \texttt{let } x = p \texttt{ in } t)$$

$$x \notin \mathcal{FV}(s) \implies \left(s \lesssim_b{}^o t \iff (\forall p \in \Lambda_{\approx}^0 : s \lesssim_b{}^o \texttt{let } x = p \texttt{ in } t) \right)$$

$$x \notin \mathcal{FV}(t) \implies \left(s \lesssim_b{}^o t \iff (\forall p \in \Lambda_{\approx}^0 : \texttt{let } x = p \texttt{ in } s \lesssim_b{}^o t) \right)$$

Obviously, according to proposition 4.1.27 the condition $\forall p \in \Lambda_{\approx}^0$ could be relaxed to $\forall p \in \mathcal{PV}$ in the above corollary. After this section has demonstrated the power of similarity proofs, the remainder of this chapter is devoted to the proof that similarity is a precongruence. Hence the subsequent section starts with an examination of the precongruence candidate in the $\boldsymbol{\lambda}_{\approx}$-calculus.

4.2 Admissibility of the Open Extension $(\cdot)^o$

So far, this chapter has covered similarity and its extension to open terms, as well as, in the precedent section, the soundness of some reduction rules which are quite of significance. This section now treats one of the major prerequisites in the proof that open similarity forms a precongruence. Namely, the open extension $(\cdot)^o$ is admissible, i.e., meets the conditions of definition 2.2.11.

Therefore, section 4.2.2 provides substitution lemmas for the respective terms which may be copied in the $\boldsymbol{\lambda}_{\approx}$-calculus. These represent the counterpart to [How96, Lemma 3.2], since the approach of Howe originally permitted to copy every term.

4.2.1 The Precongruence Candidate Revisited

Before the substitution lemmas will be established, some properties of the precongruence candidate in the calculus $\boldsymbol{\lambda}_{\approx}$ are discussed. Note that it can now be portrayed as below.

Definition 4.2.1. *Let the relation* $\widehat{\lesssim}_b \subseteq \Lambda_{\approx} \times \Lambda_{\approx}$ *defined by induction:*

- $x \widehat{\lesssim}_b b$ *if* $x \in V$ *is a variable and* $x \lesssim_b{}^o b$.

- $\tau(\overline{a}_i) \widehat{\lesssim}_b b$ *if there exists* $\overline{a}'_i$ *such that* $\overline{a}_i \widehat{\lesssim}_b \overline{a}'_i$ *and* $\tau(\overline{a}'_i) \lesssim_b{}^o b$.

Since the term $\odot$ has no operands, $\odot \widehat{\lesssim}_b s$ holds if and only if $\odot \lesssim_b{}^o s$ is true. In connection with corollary 4.1.34 this immediately yields the following.

Corollary 4.2.2. *Let* $s \in \Lambda_{\approx}$ *be a term. Then* $\odot \widehat{\lesssim}_b s$ *holds.*

Operands of the form $x.s$ occur solely for the cases $\tau = \lambda$ and $\tau = \mathtt{let}$ in definition 4.2.1. It will be used frequently that the condition $\exists y.t : \ x.s \widehat{\lesssim}_b y.t$ may be simplified to $\exists t : \ s \widehat{\lesssim}_b t$ by the definition 2.2.1 of relations on operands.

By default, no restriction on the intermediate operands $\overline{a}'_i$ is imposed in the definition 4.2.1 of the precongruence candidate. However, it seems natural that no additional free variables are necessary. This results in the characterisation of lemma 4.2.3 below. When dealing with closed terms, or, more precisely, if $a \widehat{\lesssim}_b b$ for a closed term a, the statement boils down to the subsequent corollary 4.2.4 that the intermediate term $\tau(\overline{a}'_i)$ can be chosen to be closed, too. It will be heavily used throughout section 4.3.1 where the precongruence candidate is shown to be stable under reduction.

Lemma 4.2.3. *Let* $a, b \in \Lambda_{\approx}$ *be terms. Then* $a \widehat{\lesssim}_b b$ *iff one of the following:*

- $a \equiv x$ *for a variable* $x \in V$ *and* $x \lesssim_b{}^o b$.

- $a \equiv \tau(\overline{a}_i)$ *for some operator* $\tau \in O$, *operands* a_i *and there are* a'_i *such that* $\overline{a}_i \widehat{\lesssim}_b \overline{a}'_i$ *and* $\tau(\overline{a}'_i) \lesssim_b{}^o b$ *hold with* $\mathcal{FV}(a'_i) \subseteq \mathcal{FV}(b) \cup \mathcal{FV}(a_i)$ *for all* i.

Proof. Since the "if"-part is merely a special case of definition 4.2.1, we just show the "only-if"-part. Therefore we assume $a \widehat{\lesssim}_b b$ for a case analysis along the definition of the precongruence candidate.

- For $a \equiv x \widehat{\lesssim}_b b$ with a variable x we have $x \lesssim_b{}^o b$.

- If $a \equiv \tau(\overline{a}_i) \widehat{\lesssim}_b b$, from definition 4.2.1 of the precongruence candidate we obtain operands $\overline{a}''_i$ such that $\overline{a}_i \widehat{\lesssim}_b \overline{a}''_i$ and $\tau(\overline{a}''_i) \lesssim_b{}^o b$ hold. W.l.o.g. for every i let $\mathcal{FV}(a''_i) \setminus (\mathcal{FV}(b) \cup \mathcal{FV}(a_i)) = \{\, x_{i,k} \mid 1 \leq k \leq n \,\}$. Then construct the desired operands $\overline{a}'_i$ by substituting every $x_{i,k}$ with $\odot$, i.e.:

$$a'_i \stackrel{\mathrm{def}}{\equiv} a''_i[\odot/x_{i,1}, \ldots, \odot/x_{i,n}]$$

Using these substitutions $[\overline{\circledcirc/x_{i,k}}]$ we have $\tau(\overline{a}'_i) \lesssim_b{}^o b$ from $\tau(\overline{a}''_i) \lesssim_b{}^o b$ by corollary 4.1.5, because none of the $x_{i,k}$ does occur free in b either. Now an easy induction on the structure of the operands $\overline{a}'_i$ shows that $a_i \widehat{\lesssim_b} a'_i$ follows from $a_i \widehat{\lesssim_b} a''_i$ for every i. $\qquad\square$

Corollary 4.2.4. *Let $\tau(\overline{a}_i) \in \Lambda^0_\approx$ be a closed and $b \in \Lambda_\approx$ an arbitrary term such that $\tau(\overline{a}_i) \widehat{\lesssim_b} b$ holds. Then there are operands $\overline{a}'_i$ such that $\tau(\overline{a}'_i)$ is closed too, and $\overline{a}_i \widehat{\lesssim_b} \overline{a}'_i$ as well as $\tau(\overline{a}'_i) \lesssim_b{}^o b$ is true.*

Proof. Immediately from lemma 4.2.3 since $\mathcal{FV}(\tau(\overline{a}_i)) = \emptyset$ holds. $\qquad\square$

From the previous corollary an interesting conclusion may be drawn when both terms are closed and a is an abstraction.

Lemma 4.2.5. *If $\lambda x.r \; (\widehat{\lesssim_b})_0 \; s$ for closed terms $\lambda x.r, s \in \Lambda^0_\approx$ then there is a closed $\lambda x.t$ such that $s \Downarrow \lambda x.t$ and $\lambda x.r \left[(\widehat{\lesssim_b})_0{}^o\right]_\approx \lambda x.t$ as well as $\lambda x.r \; (\widehat{\lesssim_b})_0 \; \lambda x.t$.*

Proof. From $\lambda x.r \; (\widehat{\lesssim_b})_0 \; s$ we have $\lambda x.r'$ such that $r \widehat{\lesssim_b} r'$ and $\lambda x.r' \lesssim_b{}^o s$ holds. Note that we may assume $\lambda x.r'$ to be closed according to corollary 4.2.4. By the premise s is closed, hence $\lambda x.r' \lesssim_b{}^o s$ is equivalent to $\lambda x.r' \lesssim_b s$ by corollary 4.1.8. Since $\lambda x.r'$ is an abstraction, from $\lambda x.r' \lesssim_b s$ we obtain $\lambda x.t$ such that $s \Downarrow \lambda x.t$ and $\lambda x.r' \lesssim_b \lambda x.t$ holds by corollary 4.1.30. Thus $\lambda x.r \; (\widehat{\lesssim_b})_0 \; \lambda x.t$ by composition, i.e., property 4 of lemma 2.2.10, and eventually $\lambda x.r \left[(\widehat{\lesssim_b})_0{}^o\right]_\approx \lambda x.t$ because both terms are abstractions already. $\qquad\square$

This leads to a result which is analogous to [How89, Lemma 2].

Corollary 4.2.6. *For all closed terms $a, a', b \in \Lambda^0_\approx$ we have*

$$a \Downarrow a' \;\wedge\; a' \widehat{\lesssim_b}_0 \; b \implies (\exists b' : \; b \Downarrow b' \;\wedge\; a \left[(\widehat{\lesssim_b})_0{}^o\right]_\approx b)$$

Proof. From $a' \widehat{\lesssim_b} b$ we have a closed b' such that $b \Downarrow b'$ and $a' \left[(\widehat{\lesssim_b})_0{}^o\right]_\approx b'$ by lemma 4.2.5 since a' is an abstraction. Thus $a \left[(\widehat{\lesssim_b})_0{}^o\right]_\approx b$ follows from corollary 4.1.14 since all the terms are closed. $\qquad\square$

4.2.2   Substitution Lemmas

The following substitution lemmas present an essential step forward to the proof that $\lesssim_b{}^o$ is a precongruence.

The ⊙-Substitution Lemma states that it is safe for $\widehat{\lesssim_b}$ to replace free variables by the ⊙-term. A slightly more complex case is treated by the Value-Substitution Lemma. I.e., the substitution of a free variable with $\widehat{\lesssim_b}$-related abstractions in the respective terms which are related by $\widehat{\lesssim_b}$ itself.

Lemma 4.2.7 (⊙-Substitution Lemma). *For all $b, b' \in \Lambda_{\approx}$ we have*

$$b \; \widehat{\lesssim_b} \; b' \implies b[\odot/x] \; \widehat{\lesssim_b} \; b'[\odot/x]$$

Proof. We use induction on the definition of $\widehat{\lesssim_b}$.

- If $b \equiv y \; \widehat{\lesssim_b} \; b'$ for a variable $y \in V$, then by definition 4.2.1 we have $y \lesssim_b{}^o b'$. This then, by corollary 4.1.5, means that $y[\odot/x] \lesssim_b{}^o b'[\odot/x]$ holds. Thus property 3 of lemma 2.2.10 shows the claim.

- For $b \equiv \tau(\overline{b}_i) \; \widehat{\lesssim_b} \; b'$ there must exist $\overline{b}'_i$ such that $\overline{b}_i \; \widehat{\lesssim_b} \; \overline{b}'_i$ and $\tau(\overline{b}'_i) \lesssim_b{}^o b'$ hold. From $\tau(\overline{b}'_i) \lesssim_b{}^o b'$, again by corollary 4.1.5, $\tau(\overline{b}'_i)[\odot/x] \lesssim_b{}^o b'[\odot/x]$ is obtained. On the other hand $\tau(\overline{b}_i)[\odot/x] \equiv \tau(\overline{b}_i[\odot/x]) \; \widehat{\lesssim_b} \; \tau(\overline{b}'_i[\odot/x]) \equiv \tau(\overline{b}'_i)[\odot/x]$ by the induction hypothesis and since $\widehat{\lesssim_b}$ is operator-respecting. Therefore, $\tau(\overline{b}_i)[\odot/x] \; \widehat{\lesssim_b} \; b'[\odot/x]$ is established by composition, i.e., property 4 of lemma 2.2.10. $\square$

Lemma 4.2.8 (Value-Substitution Lemma). *For all $b, b' \in \Lambda_{\approx}$ and closed terms $\lambda z.r, \lambda z.r' \in \Lambda_{\approx}^0$ the following holds:*

$$b \; \widehat{\lesssim_b} \; b' \; \wedge \; \lambda z.r \; \widehat{\lesssim_b} \; \lambda z.r' \implies b[\lambda z.r/x] \; \widehat{\lesssim_b} \; b'[\lambda z.r'/x]$$

Proof. Assume $x \in \mathcal{FV}(b) \cup \mathcal{FV}(b')$ for a proof by induction on the definition of the precongruence candidate $\widehat{\lesssim_b}$ since otherwise nothing has to be shown.

- If $b \equiv y \; \widehat{\lesssim_b} \; b'$ for a variable $y \in V$ then $y \lesssim_b{}^o b'$ from which, by corollary 4.1.5, $y[\lambda z.r'/x] \lesssim_b{}^o b'[\lambda z.r'/x]$ is implied. Now the two cases $y \equiv x$ and $y \not\equiv x$ have to be distinguished. If $y \equiv x$ then $y[\lambda z.r/x] \equiv \lambda z.r \; \widehat{\lesssim_b} \; \lambda z.r' \equiv y[\lambda z.r'/x] \lesssim_b{}^o b'[\lambda z.r'/x]$ by the premise. Thus composition, i.e., property 4 of lemma 2.2.10, establishes the claim. For $y \not\equiv x$ we have $y[\lambda z.r/x] \equiv y \equiv y[\lambda z.r'/x] \lesssim_b{}^o b'[\lambda z.r'/x]$ by the premise. Hence the proposition holds by the definition of the precongruence candidate.

- For $b \equiv \tau(\overline{b}_i) \; \widehat{\lesssim}_b \; b'$ with $\overline{b}'_i$ such that $\overline{b}_i \; \widehat{\lesssim}_b \; \overline{b}'_i$ and $\tau(\overline{b}'_i) \lesssim_b{}^o \; b'$ holds, we have $\tau(\overline{b}'_i)[\lambda z.r'/x] \lesssim_b{}^o \; b'[\lambda z.r'/x]$ by corollary 4.1.5. Furthermore, $\tau(\overline{b}_i)[\lambda z.r/x] \equiv \tau(\overline{b}_i[\lambda z.r/x]) \; \widehat{\lesssim}_b \; \tau(\overline{b}'_i[\lambda z.r'/x]) \equiv \tau(\overline{b}'_i)[\lambda z.r'/x]$ by an application of the induction hypothesis and since $\widehat{\lesssim}_b$ is operator-respecting. Thus composition by property 4 of lemma 2.2.10 proves the claim. $\square$

By means of the substitution lemmas it easily follows that the precongruence candidate is stable under substitutions which map only to $\odot$ or abstractions.

Corollary 4.2.9. *Let $s, t \in \Lambda_{\approx}$ be terms such that $s \; \widehat{\lesssim}_b \; t$ is satisfied. Then for all substitution σ with $\mathbf{rng}(\sigma) \subseteq \mathcal{PV}$ also $\sigma(s) \; \widehat{\lesssim}_b \; \sigma(t)$ holds.*

Proof. Obvious from lemma 4.2.7 and 4.2.8, since $\widehat{\lesssim}_b$ is reflexive. $\square$

The above corollary represents the up to now missing property 4 for the admissibility of the open extension $(\cdot)^o$ which thus can established now.

Proposition 4.2.10. *The extension $\lesssim_b{}^o$ of $\lesssim_b$ to open terms is admissible.*

Proof. We restate the properties from definition 2.2.11 that have to be proven.

1. $(\cdot)^o$ is preorder-preserving

2. $(\lesssim_b{}^o)_0 = \lesssim_b$

3. $\forall \nu, \eta : \; \nu \subseteq \eta \implies \nu^o \subseteq \eta^o$

4. $\widehat{\lesssim}_b \subseteq \left((\widehat{\lesssim}_b)_0 \right)^o$

Lemma 4.1.6 shows that $(\cdot)^o$ is preorder-preserving, thus condition 1 is met. Property 2 represents just the statement of corollary 4.1.8 while 3 is the consequence of corollary 4.1.9. Finally, by corollary 4.2.9, condition 4 is implied as follows. Suppose (possibly open) terms $s, t \in \Lambda_{\approx}$ such that $s \; \widehat{\lesssim}_b \; t$ holds. Furthermore, let σ be a substitution with $\mathbf{rng}(\sigma) \subseteq \mathcal{PV}$ such that $\sigma(s)$ and $\sigma(t)$ are closed. By corollary 4.2.9, from $s \; \widehat{\lesssim}_b \; t$ we obtain $\sigma(s) \; \widehat{\lesssim}_b \; \sigma(t)$ which is $\sigma(s) \; (\widehat{\lesssim}_b)_0 \; \sigma(t)$ in fact. As σ was arbitrary, $s \left((\widehat{\lesssim}_b)_0 \right)^o t$ and thus the claim. $\square$

4.3 Proving Open Similarity a Precongruence

This section is devoted to the proof that open similarity is a precongruence. According to theorem 2.2.13, it suffices to show $(\widehat{\lesssim_b})_0 \subseteq \lesssim_b$ in order to establish this result. Since $\lesssim_b$ is the greatest fixed point of $[\cdot]_{\approx}$ it contains all $[\cdot]_{\approx}$-dense sets. Thus the condition $(\widehat{\lesssim_b})_0 \subseteq \lesssim_b$ follows from $(\widehat{\lesssim_b})_0 \subseteq \left[(\widehat{\lesssim_b})_0{}^o\right]_{\approx}$ by coinduction. This in turn has been addressed in section 2.4.1 by the notion that a relation is respected by evaluation. Corollary 2.4.9 shows that in such a case the relation is indeed $[\cdot^o]_{\approx}$-dense, i.e, a simulation.

   Definition 2.4.10 suggests the related notion *stable under reduction* which is more suitable for small-step reduction. Before we can show that $s' \widehat{\lesssim_b} t$ holds whenever $s \widehat{\lesssim_b} t$ and $s \xrightarrow{\mathcal{S}}_{\boldsymbol{\lambda}_{\approx}} s'$ does, we have to verify how condition (2.4.4) implies that the precongruence candidate is respected by evaluation.

Lemma 4.3.1. *If $\left(s \ (\widehat{\lesssim_b})_0 \ t \ \wedge \ s \Downarrow \lambda x.s'\right) \implies \lambda x.s' \ (\widehat{\lesssim_b})_0 \ t$ holds for all closed terms $s, \lambda x.s', t \in \Lambda_{\approx}^0$ then $(\widehat{\lesssim_b})_0$ is a simulation.*

Proof. Suppose closed terms $s, \lambda x.s', t \in \Lambda_{\approx}^0$ for which $s \ (\widehat{\lesssim_b})_0 \ t$ and $s \Downarrow \lambda x.s'$ as well as $\lambda x.s' \ (\widehat{\lesssim_b})_0 \ t$ holds. To show that $(\widehat{\lesssim_b})_0$ is respected by evaluation requires to find a $\lambda x.t'$ such that $t \Downarrow \lambda x.t'$ and $\lambda x.s' \ \left[(\widehat{\lesssim_b})_0{}^o\right]_{\approx} \lambda x.t'$ are satisfied. From $\lambda x.s' \ (\widehat{\lesssim_b})_0 \ t$ such an abstraction is obtained by lemma 4.2.5. $\qquad\square$

   For lemma 4.3.1 to apply it is to prove that $s' \widehat{\lesssim_b} t$ holds whenever $s \widehat{\lesssim_b} t$ and $s \Downarrow s'$ do so for closed terms. As the convergence relation $\Downarrow$ is defined in terms of stepwise reduction, this can be accomplished for every reduction rule separately. So it will be shown that the precongruence candidate restricted to closed terms is stable under reduction, i.e., $s \ (\widehat{\lesssim_b})_0 \ t \ \wedge \ s \xrightarrow{\mathcal{S}}_{\boldsymbol{\lambda}_{\approx}} s' \implies s' \ (\widehat{\lesssim_b})_0 \ t$ holds. For this, we fall back upon non-closing surface contexts in lemma 4.3.10, since the reduction rules shall only be treated at top-level.

4.3.1 Precongruence Candidate Stable Under Reduction

Now a series of lemmas will be established showing that $s \widehat{\lesssim_b} t \ \wedge \ s \xrightarrow{[\ ],\ a}_{\boldsymbol{\lambda}_{\approx}} s'$ implies $s' \widehat{\lesssim_b} t$ for each reduction rule (a) of the $\boldsymbol{\lambda}_{\approx}$-calculus.

Lemma 4.3.2 (Stability of $(\widehat{\lesssim_b})_0$ Under (lapp)**).** *Let $s, t, t_x \in \Lambda_\approx$ be terms such that $(\mathtt{let}\ x = t_x\ \mathtt{in}\ s)\, t$ is closed. Then we have*

$$((\mathtt{let}\ x = t_x\ \mathtt{in}\ s)\, t \xrightarrow{lapp}_{\pmb{\lambda}_\approx} \mathtt{let}\ x = t_x\ \mathtt{in}\ (s\, t) \ \wedge$$
$$(\mathtt{let}\ x = t_x\ \mathtt{in}\ s)\, t \ \widehat{\lesssim_b}\ b) \implies \mathtt{let}\ x = t_x\ \mathtt{in}\ (s\, t) \ \widehat{\lesssim_b}\ b$$

Proof. From $(\mathtt{let}\ x = t_x\ \mathtt{in}\ s)\, t \ \widehat{\lesssim_b}\ b$ we have

$$\exists l', t' : \ \mathtt{let}\ x = t_x\ \mathtt{in}\ s \ \widehat{\lesssim_b}\ l' \ \wedge \ t \ \widehat{\lesssim_b}\ t' \ \wedge \ l'\, t' \ \lesssim_b{}^o\ b \qquad (4.3.1)$$

such that, by corollary 4.2.4, we may assume $l'\, t'$ and hence the subterms l', t' itself to be closed. Furthermore, $\mathtt{let}\ x = t_x\ \mathtt{in}\ s \ \widehat{\lesssim_b}\ l'$ implies

$$\exists x.s', t'_x : \ x.s \ \widehat{\lesssim_b}\ x.s' \ \wedge \ t_x \ \widehat{\lesssim_b}\ t'_x \ \wedge \ \mathtt{let}\ x = t'_x\ \mathtt{in}\ s' \ \lesssim_b{}^o\ l' \qquad (4.3.2)$$

with $\mathtt{let}\ x = t'_x\ \mathtt{in}\ s'$ again to be closed according to corollary 4.2.4, so that $\mathtt{let}\ x = t'_x\ \mathtt{in}\ s' \ \lesssim_b\ l'$ holds on closed terms. Since $\lesssim_b$ is respected by closed A_L^*-contexts, cf. lemma 4.1.31, we have

$$(\mathtt{let}\ x = t'_x\ \mathtt{in}\ s')\, t' \ \lesssim_b\ l'\, t'$$

from $\mathtt{let}\ x = t'_x\ \mathtt{in}\ s' \ \lesssim_b\ l'$, and hence $(\mathtt{let}\ x = t'_x\ \mathtt{in}\ s')\, t' \ \lesssim_b{}^o\ b$ by corollary 4.1.7. As approximation reductions are sound w.r.t. $\gtrsim_b{}^o$ by corollary 4.1.12, we may apply the (lapp)-reduction also to $(\mathtt{let}\ x = t'_x\ \mathtt{in}\ s')\, t'$ which results in

$$(\mathtt{let}\ x = t'_x\ \mathtt{in}\ s')\, t' \xrightarrow{lapp}_{\pmb{\lambda}_\approx} \mathtt{let}\ x = t'_x\ \mathtt{in}\ s'\, t' \ \lesssim_b{}^o\ b$$

We have $t_x \ \widehat{\lesssim_b}\ t'_x$, $s \ \widehat{\lesssim_b}\ s'$ and $t \ \widehat{\lesssim_b}\ t'$ by construction and $s\, t \ \widehat{\lesssim_b}\ s'\, t'$ since $\widehat{\lesssim_b}$ is operator-respecting, thus $\mathtt{let}\ x = t_x\ \mathtt{in}\ s\, t \ \widehat{\lesssim_b}\ b$ holds. $\qquad\qquad\square$

Lemma 4.3.3 (Stability of $(\widehat{\lesssim_b})_0$ Under (lbeta)**).** *Let $s, t \in \Lambda_\approx$ be terms such that $(\lambda x.s)\, t$ is closed. Then*

$$(\lambda x.s)\, t \xrightarrow{lbeta}_{\pmb{\lambda}_\approx} \mathtt{let}\ x = t\ \mathtt{in}\ s \ \wedge \ (\lambda x.s)\, t \ \widehat{\lesssim_b}\ b \implies \mathtt{let}\ x = t\ \mathtt{in}\ s \ \widehat{\lesssim_b}\ b$$

Proof. Assume $(\lambda x.s)\, t \xrightarrow{lbeta} \mathtt{let}\ x = t\ \mathtt{in}\ s$ and $(\lambda x.s)\, t \ \widehat{\lesssim_b}\ b$. By corollary 4.2.4, from the latter we have closed terms $f', t' \in \Lambda_\approx^0$ such that $(\lambda x.s) \ \widehat{\lesssim_b}\ f'$ and $t \ \widehat{\lesssim_b}\ t'$, as well as $f'\, t' \ \lesssim_b{}^o\ b$ is valid. Expanding $(\lambda x.s) \ \widehat{\lesssim_b}\ f'$ further, we obtain a closed $\lambda x.s' \in \Lambda_\approx^0$ which fulfils $s \ \widehat{\lesssim_b}\ s'$ and $\lambda x.s' \ \lesssim_b{}^o\ f'$, hence $\lambda x.s' \ \lesssim_b\ f'$

for the closed relation, too. By corollary 4.1.30 this means, that there is a closed abstraction $\lambda x.s'' \in \Lambda_\approx^0$ such that $f' \Downarrow \lambda x.s''$ and $\lambda x.s' \lesssim_b \lambda x.s''$ hold.

We obviously may perform the reduction $f' \xrightarrow{S}{}^*_{\lambda_\approx} \lambda x.s''$ also inside the surface context $[\,]\,t'$ so $(\lambda x.s'')\,t' \lesssim_b f'\,t'$ holds. From $\lambda x.s' \lesssim_b \lambda x.s''$ we also have $(\lambda x.s')\,t' \lesssim_b (\lambda x.s'')\,t'$ by corollary 4.1.28. Furthermore we may reduce $(\lambda x.s')\,t' \xrightarrow{\text{lbeta}} \texttt{let } x = t' \texttt{ in } s'$ hence the chain

$$\texttt{let } x = t' \texttt{ in } s' \lesssim_b (\lambda x.s')\,t' \lesssim_b (\lambda x.s'')\,t' \lesssim_b f'\,t' \lesssim_b{}^o b$$

Since then $\texttt{let } x = t' \texttt{ in } s' \lesssim_b{}^o b$ holds by corollary 4.1.7, we complete the proof by recognising that $\texttt{let } x = t \texttt{ in } s \;\widehat{\lesssim_b}\; \texttt{let } x = t' \texttt{ in } s'$ holds since $\widehat{\lesssim_b}$ is operator-respecting. Thus $\texttt{let } x = t \texttt{ in } s \;\widehat{\lesssim_b}\; b$ by property (4) of lemma 2.2.10.　　$\square$

Lemma 4.3.4 (Stability of $(\widehat{\lesssim_b})_0$ Under (lseq)). *Let* $\texttt{let } x = s_x \texttt{ in } s, t \in \Lambda_\approx$ *be terms such that* $(\texttt{let } x = s_x \texttt{ in } s) \texttt{ seq } t$ *is closed. Then*

$$((\texttt{let } x = s_x \texttt{ in } s) \texttt{ seq } t \xrightarrow{lseq} \texttt{let } x = s_x \texttt{ in } (s \texttt{ seq } t) \;\wedge$$

$$(\texttt{let } x = s_x \texttt{ in } s) \texttt{ seq } t \;\widehat{\lesssim_b}\; b) \implies \texttt{let } x = s_x \texttt{ in } (s \texttt{ seq } t) \;\widehat{\lesssim_b}\; b$$

Proof. From $(\texttt{let } x = s_x \texttt{ in } s) \texttt{ seq } t \;\widehat{\lesssim_b}\; b$ by corollary 4.2.4 we have

$$\exists l', t' : \;\texttt{let } x = s_x \texttt{ in } s \;\widehat{\lesssim_b}\; l' \;\wedge\; t \;\widehat{\lesssim_b}\; t' \;\wedge\; l' \texttt{ seq } t' \lesssim_b{}^o b \qquad (4.3.3)$$

such that $l' \texttt{ seq } t'$ is closed and hence the subterms l' and t' itself, too. Again, from $\texttt{let } x = s_x \texttt{ in } s \;\widehat{\lesssim_b}\; l'$ we have

$$\exists s_x', s' : \; s_x \;\widehat{\lesssim_b}\; s_x' \;\wedge\; s \;\widehat{\lesssim_b}\; s' \;\wedge\; \texttt{let } x = s_x' \texttt{ in } s' \lesssim_b{}^o l' \qquad (4.3.4)$$

Hence it remains to show that $\texttt{let } x = s_x' \texttt{ in } (s' \texttt{ seq } t') \lesssim_b{}^o b$ is true. So assume $\texttt{let } x = s_x' \texttt{ in } (s' \texttt{ seq } t') \Downarrow \lambda z.q$. Then by the standardisation of lemma 3.2.26 there is also an approximation reduction

$$\texttt{let } x = s_x' \texttt{ in } (s' \texttt{ seq } t') \xrightarrow[\quad]{\texttt{let } x=[\,]\texttt{ in } (s'\texttt{ seq } t')}{}^*_{\lambda_\approx} \texttt{let } x = s_x'' \texttt{ in } (s' \texttt{ seq } t')$$

$$\xrightarrow{\text{cpa}} s'[s_x''/x] \texttt{ seq } t'[s_x''/x] \equiv s'[s_x''/x] \texttt{ seq } t' \xrightarrow{S}{}^*_{\lambda_\approx} \lambda z.q$$

where $t'[s_x''/x] \equiv t'$ holds, since x does not occur free in t'. Clearly, the first part of these reductions may independently also be performed within the surface context $\texttt{let } x = [\,] \texttt{ in } s'$ which yields

$$\texttt{let } x = s_x' \texttt{ in } s' \xrightarrow[\quad]{\texttt{let } x=[\,]\texttt{ in } s'}{}^*_{\lambda_\approx} \texttt{let } x = s_x'' \texttt{ in } s' \xrightarrow{\text{cpa}} s'[s_x''/x] \qquad (4.3.5)$$

From $s'[s''_x/x]$ `seq` $t' \Downarrow \lambda z.q$ we have $t' \Downarrow \lambda z.q$ and $s'[s''_x/x] \Downarrow$ by lemma 3.2.29, hence `let` $x = s'_x$ `in` $s' \lesssim_b{}^o l'$ of (4.3.4) implies $l' \Downarrow$ too. Thus l' `seq` $t' \Downarrow \lambda z.q$ again by lemma 3.2.29. So l' `seq` $t' \lesssim_b{}^o b$ in (4.3.3) proves the claim. □

Lemma 4.3.5 (Stability of $(\widehat{\lesssim_b})_0$ Under (eseq)). *For $\lambda x.s, t \in \Lambda_\approx$ we have:*

$$(\lambda x.s) \ \texttt{seq} \ t \xrightarrow{eseq} t \ \wedge \ (\lambda x.s) \ \texttt{seq} \ t \ \widehat{\lesssim_b} \ b \implies t \ \widehat{\lesssim_b} \ b$$

Proof. In an analogous manner like for the (nd)-rules below. □

Lemma 4.3.6 (Stability of $(\widehat{\lesssim_b})_0$ Under (nd, left)). *For $s, t \in \Lambda_\approx$ we have:*

$$\texttt{pick} \ s \ t \xrightarrow{nd, \ left}_{\lambda_\approx} s \ \wedge \ \texttt{pick} \ s \ t \ \widehat{\lesssim_b} \ b \implies s \ \widehat{\lesssim_b} \ b$$

Proof. From `pick` $s \ t \ \widehat{\lesssim_b} \ b$ we obtain

$$\exists s', t' : \ s \ \widehat{\lesssim_b} \ s' \ \wedge \ t \ \widehat{\lesssim_b} \ t' \ \wedge \ \texttt{pick} \ s' \ t' \lesssim_b{}^o \ b$$

Approximation reductions are sound w.r.t. $\gtrsim_b{}^o$ by corollary 4.1.39, so we have

$$s' \lesssim_b{}^o \ \texttt{pick} \ s' \ t'$$

from `pick` $s' \ t' \xrightarrow{nd, \ left}_{\lambda_\approx} s'$. Hence by transitivity

$$s \ \widehat{\lesssim_b} \ s' \lesssim_b{}^o \ \texttt{pick} \ s' \ t' \lesssim_b{}^o \ b$$

thus $s \ \widehat{\lesssim_b} \ b$ by composition, i.e. property 4 of lemma 2.2.10. □

Lemma 4.3.7 (Stability of $(\widehat{\lesssim_b})_0$ Under (nd, right)). *For $s, t \in \Lambda_\approx$ we have:*

$$\texttt{pick} \ s \ t \xrightarrow{nd, \ right}_{\lambda_\approx} s \ \wedge \ \texttt{pick} \ s \ t \ \widehat{\lesssim_b} \ b \implies s \ \widehat{\lesssim_b} \ b$$

Proof. The argument is symmetric to the one for the (nd, left)-rule. □

Lemma 4.3.8 (Stability of $(\widehat{\lesssim_b})_0$ Under (stop)). *Let $s \in \Lambda_\approx$ be a term such that $s \not\equiv \odot$ holds. Then the following holds:*

$$s \xrightarrow{stop} \odot \ \wedge \ s \ \widehat{\lesssim_b} \ b \implies \odot \ \widehat{\lesssim_b} \ b$$

Proof. Obvious by corollary 4.2.2. □

Based on the substitution lemmas from section 4.2.2, the proof for a reduction by rule (cpa) is quite involved.

Lemma 4.3.9 (Stability of $(\widehat{\lesssim}_b)_0$ Under (cpa)). *Let $s, t \in \Lambda_{\approx}$ be terms such that* let $x = s$ in t *is closed. Then the following is true:*

$$\texttt{let } x = s \texttt{ in } t \xrightarrow{cpa} t[s/x] \ \wedge \ \texttt{let } x = s \texttt{ in } t \ \widehat{\lesssim}_b \ b \implies t[s/x] \ \widehat{\lesssim}_b \ b$$

Proof. Show the cases for $s \equiv \odot$ and $s \equiv \lambda z.q$ from definition 3.2.2 separately.

- For $s \equiv \odot$ the proposition is

$$\texttt{let } x = \odot \texttt{ in } t \xrightarrow{cpa} t[\odot/x] \ \wedge \ \texttt{let } x = \odot \texttt{ in } t \ \widehat{\lesssim}_b \ b \implies t[\odot/x] \ \widehat{\lesssim}_b \ b$$

  From let $x = \odot$ in $t \ \widehat{\lesssim}_b \ b$, by definition 4.2.1, we have s', t' such that

$$\odot \ \widehat{\lesssim}_b \ s' \ \wedge \ x.t \ \widehat{\lesssim}_b \ x.t' \ \wedge \ \texttt{let } x = s' \texttt{ in } t' \lesssim_b^o b$$

  hence $t[\odot/x] \ \widehat{\lesssim}_b \ t'[\odot/x]$ from $t \ \widehat{\lesssim}_b \ t'$ by lemma 4.2.7 and furthermore

$$t'[\odot/x] \lesssim_b^o \texttt{let } x = \odot \texttt{ in } t' \lesssim_b^o \texttt{let } x = s' \texttt{ in } t' \lesssim_b^o b$$

  since let $x = s'$ in $t' \xrightarrow{stop}$ let $x = \odot$ in $t' \xrightarrow{cpa} t'[\odot/x]$. Thus $t[\odot/x] \ \widehat{\lesssim}_b \ b$ holds by composition, i.e. property (4) of lemma 2.2.10.

- If $s \equiv \lambda z.q$ the claim reads as follows

$$\texttt{let } x = \lambda z.q \texttt{ in } t \xrightarrow{cpa} t[\lambda z.q/x] \ \wedge \ \texttt{let } x = \lambda z.q \texttt{ in } t \ \widehat{\lesssim}_b \ b \implies$$
$$t[\lambda z.q/x] \ \widehat{\lesssim}_b \ b$$

  From let $x = \lambda z.q$ in $t \ \widehat{\lesssim}_b \ b$, by corollary 4.2.4, we have l', t' such that

$$\lambda z.q \ \widehat{\lesssim}_b \ l' \ \wedge \ t \ \widehat{\lesssim}_b \ t' \ \wedge \ \texttt{let } x = l' \texttt{ in } t' \lesssim_b^o b$$

  holds and let $x = l'$ in t' is closed, which implies that l' itself is closed. Expanding the definition of $\widehat{\lesssim}_b$ further, from $\lambda z.q \ \widehat{\lesssim}_b \ l'$ we have q' with

$$q \ \widehat{\lesssim}_b \ q' \ \wedge \ \lambda z.q' \lesssim_b^o l'$$

which obviously implies $\lambda z.q \mathrel{\widehat{\precsim_b}} \lambda z.q'$ because $\widehat{\precsim_b}$ is operator-respecting. Since $\lambda z.q'$ again is closed by corollary 4.2.4, from $\lambda z.q' \precsim_b^o l'$ we even have $\lambda z.q' \precsim_b l'$, hence $l' \Downarrow \lambda z.q''$ with

$$\lambda z.q' \precsim_b \lambda z.q''$$

by corollary 4.1.30, as $\lambda z.q'$ is an abstraction. So $\lambda z.q \mathrel{\widehat{\precsim_b}} \lambda z.q''$ follows from $\lambda z.q \mathrel{\widehat{\precsim_b}} \lambda z.q'$ in connection with $\lambda z.q' \precsim_b \lambda z.q''$ by composition, i.e., property 4 of lemma 2.2.10. Moreover, the reduction $l' \xrightarrow{\mathcal{S}}{}^*_{\boldsymbol{\lambda}_\approx} \lambda z.q''$ can also be performed inside the surface context $\mathtt{let}\ x = [\,]\ \mathtt{in}\ t'$, hence

$$\mathtt{let}\ x = l'\ \mathtt{in}\ t' \xrightarrow{\mathcal{S}}{}^*_{\boldsymbol{\lambda}_\approx} \mathtt{let}\ x = \lambda z.q''\ \mathtt{in}\ t' \xrightarrow{\mathrm{cpa}} t'[\lambda z.q''/x]$$

Since $a \to b$ implies $b \precsim_b a$ by corollary 4.1.12, from $\mathtt{let}\ x = l'\ \mathtt{in}\ t' \precsim_b b$, by transitivity of $\precsim_b$ and corollary 4.1.7, we have

$$t'[\lambda z.q''/x] \precsim_b^o b$$

With $t \mathrel{\widehat{\precsim_b}} t'$ and $\lambda z.q \mathrel{\widehat{\precsim_b}} \lambda z.q''$ the preconditions of lemma 4.2.8 are satisfied, thus $t[\lambda z.q/x] \mathrel{\widehat{\precsim_b}} b$ follows from $t[\lambda z.q/x] \mathrel{\widehat{\precsim_b}} t'[\lambda z.q''/x]$ and the above by property (4) of lemma 2.2.10. □

From the above, we achieve that the precongruence candidate restricted to closed terms is stable under top-level reductions. This result will be carried over to reductions within non-closing surface contexts below.

4.3.2 Establishing the Precongruence Theorem

In the previous section we have investigated the stability of the precongruence candidate on closed terms under top-level reductions, but we have not yet completely established the link for the use of theorem 2.2.13 in the proof that open similarity forms a precongruence.

To prove the Precongruence Theorem, first the precongruence candidate on closed terms is shown to be stable under reductions within non-closing surface contexts and subsequently the corresponding instance of (2.4.4) is deduced.

Lemma 4.3.10. *Let $p, q \in \Lambda_\approx^0$ be closed terms such that $p \xrightarrow{\mathcal{N},\,a}{}_{\boldsymbol{\lambda}_\approx} q$ with some rule (a) of definition 3.2.2. Then for every term r: $p \mathrel{\widehat{\precsim_b}} r$ implies $q \mathrel{\widehat{\precsim_b}} r$.*

Proof. We assume $p \equiv N[p'] \xrightarrow{N,\, a} N[q'] \equiv q$ with $p' \xrightarrow{[\,],\, a} q'$ for some arbitrary, but fixed non-closing surface context $N \in \mathcal{N}$. Then $p', q' \in \Lambda_{\approx}^0$ have to be closed terms by definition 3.2.23 and we may use induction over the structure of N:

- For $N \equiv [\,]$ the claim holds by one of the lemmas above.

- If $N \equiv N'\, t$ for some surface context $N' \in \mathcal{N}$ and a term $t \in \Lambda_{\approx}^0$ such that $p \equiv N'[p']\, t$ holds, then from $p \stackrel{\frown}{\lesssim}_b r$ we have s_1, s_2 such that

$$N'[p'] \stackrel{\frown}{\lesssim}_b s_1 \ \wedge\ t \stackrel{\frown}{\lesssim}_b s_2 \ \wedge\ s_1\, s_2 \lesssim_b^o r$$

  by definition 4.2.1. Since N has only one unique hole, the (a)-reduction may also take place within N', hence

$$N'[p'] \xrightarrow{N',\, a} N'[q']$$

  to which we may apply the induction hypothesis, i.e.

$$N'[p'] \xrightarrow{N',\, a} N'[q'] \ \wedge\ N'[p'] \stackrel{\frown}{\lesssim}_b s_1 \implies N'[q'] \stackrel{\frown}{\lesssim}_b s_1$$

  thus $q \equiv N[q'] \stackrel{\frown}{\lesssim}_b r$ immediately follows.

- The cases $s\, N'$, `pick` $N'\, t$, `pick` $s\, N'$ and `let` $x = N$ `in` t can be shown accordingly. □

Using induction on the length of $\mathcal{N}$-approximation reduction sequences, yields $s \left(\stackrel{\frown}{\lesssim}_b\right)_0 t \ \wedge\ s \Downarrow \lambda x.s' \implies \lambda x.s' \left(\stackrel{\frown}{\lesssim}_b\right)_0 t$ by the above lemma.

Lemma 4.3.11. *Let $\lambda x.r, s, t \in \Lambda_{\approx}^0$ be closed terms such that $s \left(\stackrel{\frown}{\lesssim}_b\right)_0 t$ and $s \Downarrow \lambda x.r$ hold. Then also $\lambda x.r \left(\stackrel{\frown}{\lesssim}_b\right)_0 t$ is true.*

Proof. By lemma 3.2.25 it suffices to regard a converging approximation reduction involving non-closing surface contexts only. So assume $s \left(\stackrel{\frown}{\lesssim}_b\right)_0 t$ and $s \xrightarrow[\boldsymbol{\lambda}_{\approx}]{\mathcal{N},\ k} \lambda x.r$ for an induction on the length k of the reduction sequence.

- For $k = 0$ nothing has to be shown.

- If $s \xrightarrow[\boldsymbol{\lambda}_{\approx}]{\mathcal{N}} s' \xrightarrow[\boldsymbol{\lambda}_{\approx}]{\mathcal{N}\ k-1} \lambda x.r$ with $k > 0$ we obtain $s' \left(\stackrel{\frown}{\lesssim}_b\right)_0 t$ from $s \left(\stackrel{\frown}{\lesssim}_b\right)_0 t$ by lemma 4.3.10. Thus an application of the induction hypothesis to the approximation reduction sequence $s' \xrightarrow[\boldsymbol{\lambda}_{\approx}]{\mathcal{N}\ k-1} \lambda x.r$ proves the claim. □

Proposition 4.3.12. *The restriction $(\widehat{\lesssim}_b)_0$ of $\widehat{\lesssim}_b$ to closed terms is a simulation, i.e., the inclusion $(\widehat{\lesssim}_b)_0 \subseteq \left[(\widehat{\lesssim}_b)_0^{\;o} \right]_{\approx}$ is valid.*

Proof. Let $s, t \in \Lambda_{\approx}^0$ be closed terms such that $s \; (\widehat{\lesssim}_b)_0 \; t$ holds. Furthermore, assume $s \Downarrow \lambda x.s'$, i.e., $\exists k : \; s \xrightarrow[\boldsymbol{\lambda}_{\approx}]{\mathcal{S} \;\; k} \lambda x.s'$ as well. By lemma 3.2.25, there is also an approximation reduction sequence $s \xrightarrow[\boldsymbol{\lambda}_{\approx}]{\mathcal{N} \;\; k'} \lambda x.s'$ to the same abstraction, taking place within non-closing surface contexts only.

Thus $\lambda x.s' \; (\widehat{\lesssim}_b)_0 \; t$ follows from $s \; (\widehat{\lesssim}_b)_0 \; t$ by lemma 4.3.11. By corollary 4.2.6, there is a closed $\lambda x.t'$ such that $t \Downarrow \lambda x.t'$ and $s \; \left[(\widehat{\lesssim}_b)_0^{\;o} \right]_{\approx} \; t$ hold. $\quad\square$

Having proven $(\widehat{\lesssim}_b)_0$ a simulation, it is now a straightforward consequence that the inclusion $(\widehat{\lesssim}_b)_0 \subseteq \lesssim_b$ holds by coinduction. Together with the admissibility of $\lesssim_b^{\;o}$, this enables the application of theorem 2.2.13 in order to establish one of the main results.

Precongruence Theorem 4.3.13. *Open similarity $\lesssim_b^{\;o}$ is a precongruence.*

Proof. Proposition 4.2.10 shows that the open extension $\lesssim_b^{\;o}$ is admissible. Hence by theorem 2.2.13 it suffices to show that $(\widehat{\lesssim}_b)_0 \subseteq \lesssim_b$ holds. Recall that, as a greatest fixed point, $\lesssim_b$ contains all $[\cdot^o]_{\approx}$-dense sets, therefore the inclusion $(\widehat{\lesssim}_b)_0 \subseteq \left[(\widehat{\lesssim}_b)_0^{\;o} \right]_{\approx}$ is sufficient. By proposition 4.3.12 this inclusion is true, hence $(\widehat{\lesssim}_b)_0$ is $[\cdot^o]_{\approx}$-dense and thus $\lesssim_b^{\;o}$ a precongruence. $\quad\square$

As noted before, this is the essential precondition for showing that similarity complies with contextual preorder, which will be addressed in a later section. Thus, with mutual open similarity this chapter provides a powerful notion of program equivalence which is a suitable criterion for the correctness of program transformations.

By the concluding example, we may now get a better understanding of how non-determinism and open similarity interact in the $\boldsymbol{\lambda}_{\approx}$-calculus.

Example 4.3.14. *Let s and t be terms such that $s \lesssim_b^{\;o} t$ or $t \lesssim_b^{\;o} s$ holds. Then we have $\lambda x.\mathtt{pick}\; s\; t \lesssim_b^{\;o} \mathtt{pick}\; (\lambda x.s)\; (\lambda x.t)$, i.e., the potential to copy the non-deterministic choice of two $\lesssim_b^{\;o}$-related abstractions does not gain anything.*

W.l.o.g. assume $s \lesssim_b^{\;o} t$ from which we have $\mathtt{pick}\; s\; t \lesssim_b^{\;o} t$ by adopting the argument in example 4.1.10 to open similarity. Since open similarity $\lesssim_b^{\;o}$ is a precongruence, we obtain $\lambda x.\mathtt{pick}\; s\; t \lesssim_b^{\;o} \lambda x.t$ therefrom.

On the other hand, the relation $\lambda x.t \lesssim_b{}^o$ `pick` *$(\lambda x.s)$ $(\lambda x.t)$ follows directly from the top-level reduction* `pick` *$(\lambda x.s)$ $(\lambda x.t)$ $\xrightarrow{nd,\ right}$ $\lambda x.t$ and thus $\lambda x.$`pick`$\ s\ t \lesssim_b{}^o \lambda x.t \lesssim_b{}^o$* `pick` *$(\lambda x.s)$ $(\lambda x.t)$ is established.*

4.4   Future Work

As explained before, one motivation for developing a notion of similarity for the non-deterministic call-by-need lambda calculus $\boldsymbol{\lambda}_{\mathrm{ND}}$, was its use as a tool for proving contextual equivalences. Though the results of the preceding section show that this is indeed the case, apart from coinduction no proof strategy has been established so far.

This is a non-trivial task since the approximation reduction in the calculus $\boldsymbol{\lambda}_{\approx}$ comprises two further degrees of non-determinism beyond the `pick`-construct: For every reduction step there are usually various surface context available while rule $\circledcirc$ competes with all the other rules. So the prior question is whether there is a some kind of "standard" reduction or, at least, a "more deterministic" order for approximation reductions.

Moreover, like in [Kut99], it would be interesting to examine the deterministic fragment of the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus. These aspects will be addressed subsequently.

4.4.1   Reduction Strategies for the $\boldsymbol{\lambda}_{\approx}$-Calculus

In order to devise a reduction strategy for the $\boldsymbol{\lambda}_{\approx}$-calculus which is able to *determine* a converging approximation reduction if there is one, we will briefly discuss some ideas. Though the normalisation theorem 3.2.28 implements some kind of "leftmost-outermost" reduction order, it does *not* represent a suitable procedure for finding an abstraction.

One elementary requirement is that it should be detected when convergence is possible without using the (stop)-rule. It is easy to see that in general this cannot be accomplished by (olf)-reductions.

On the other hand, trying to simulate normal-order reduction to some extent, seems sensible. The crucial point is whether the rules (llet) and (cp) should be kept or replaced by (stop) and (cpa) from the $\boldsymbol{\lambda}_{\approx}$-calculus.

If only the rules of the approximation calculus $\boldsymbol{\lambda}_{\approx}$ were used, the effect of the rule (llet) would have to be emulated by descending recursively into the respective binding. To see that this situation is slightly different from the normalisation, a normal-order (llet)-redex `let` $x = ($`let` $y = t_y$ `in` $t_x)$ `in` $R[x]$ is considered. Since x is in a reduction context and thus needed for the reduction,

the whole term has no WHNF if `let` $y = t_y$ `in` t_x hasn't any. So it would make sense to proceed inside `let` $y = t_y$ `in` t_x with the reduction. But, it is still not clear when and where rule (stop) should be applied. The reason is that it is in advance unknown which abstraction from the answer set of `let` $y = t_y$ `in` t_x suffices for the whole term `let` $x = ($`let` $y = t_y$ `in` $t_x)$ `in` $R[x]$ to have an approximation reduction to an abstraction. The advantage of this method would be that all reductions which are involved belong to the $\boldsymbol{\lambda}_{\approx}$-calculus.

An alternative could be to switch back to normal-order reduction to weak head normal form. Then, using (stop) and (cpa) on the `let`-environment, an abstraction can be produced. In general such an abstraction will clearly possess less information than the corresponding weak head normal form and thus all possible abstractions have to be considered — this is just the way the approximation calculus $\boldsymbol{\lambda}_{\approx}$ works. The results from section 4.1.4 suggest that the use of (llet) and (cp) during this process is safe for a possible approximation reduction to an abstraction. Particularly, lemma 4.1.44 demonstrates that (llet) preserves the answer set of a term while an analogous result[2] seems possible for normal-order (cp)-reductions.

Hence the latter approach looks most promising since it can be based on many results already devised in this work.

4.4.2 Similarity Checking

Now possibilities and limitations will be discussed of how the technique presented in the previous section could be used to check for similarity. So a short illustration of the procedure using normal-order reduction will be given.

Assume two closed terms $s, t \in \Lambda_{\mathrm{ND}}^0$ for which $s \lesssim_b t$ is to be shown. If s has no WHNF the relation will trivially be true, hence suppose $s \xrightarrow[\boldsymbol{\lambda}_{\mathrm{ND}}]{n \ *} L_s[\lambda x.s']$ to hold. If t has *no* WHNF the relation will be false, so $t \xrightarrow[\boldsymbol{\lambda}_{\mathrm{ND}}]{n \ *} L_t[\lambda x.t']$ is assumed. As a first step, the abstractions $\lambda x.s'[\overline{\circledcirc}/\overline{x}]$ and $\lambda x.t'[\overline{\circledcirc}/\overline{x}]$ are produced, i.e., all variables which were bound in the `let`-environments, denoted by $\overline{x}$, are replaced by the $\circledcirc$-term. Note that $\lambda x.s'[\overline{\circledcirc}/\overline{x}] \lesssim_b \lambda x.t'[\overline{\circledcirc}/\overline{x}]$ is not necessary for $s \lesssim_b t$ to hold.

Therefore, the next stage proceeds within the `let`-environments L_s and L_t respectively. I.e., Let $\overline{s}_i$ and $\overline{t}_i$ respectively stand for the terms to which the variables $\overline{x}$ are bound to, then $\overline{s}_i$ and $\overline{t}_i$ are reduced in normal-order until a WHNF is reached, if possible. The goal is again to obtain abstractions from

---

[2]Note that in the case of (cpa), internal (stop)-reductions come into play, only because copy targets might be at non-surface locations.

these weak head normal forms. So unless there are abstractions already, the process is applied recursively. Finally, using (cpa) abstractions may be copied and thus the environments L_s and L_t are eliminated.

Backed up by (4.1.7) of corollary 4.1.30 we can establish $s \precsim_b t$ by showing that for every abstraction $\lambda x.s''$ derived from s in the above manner, there exists a corresponding abstraction $\lambda x.t''$ for t such that $\lambda x.s'' \precsim_b \lambda x.t''$ holds.

Though the answer set is computed in a systematic way, it is not yet clear how to accomplish the comparison of all possible answers — usually there are infinitely many. But the approach discussed above might offer the possibility to "short-circuit" proof obligations which occur repeatedly. The idea behind it is similar to the detection of a "loop", e.g. in [Sch00, SSSS04] where an infinite (cyclic) expansion of a proof tree can be avoided in some situations.

4.4.3   Deterministic Subterms

The treatment of deterministic subterms in [Kut99] extends over a whole chapter. Since the aim of this work is on similarity, we only glimpse at what deterministic essentially means and why those terms could be copied unrestrictedly.

Definition 4.4.1. *Let $s \in \Lambda_{\approx}^0$ be a closed term. Then s is* deterministic *if and only if s does not contain the* `pick`*-operator.*

If the above notion was given for the calculus $\boldsymbol{\lambda}_{\mathrm{ND}}$, a deterministic term automatically would have at most one weak head normal form. But, as explained before, in the $\boldsymbol{\lambda}_{\approx}$-calculus there exist further levels of non-determinism. Though it seems not immediately clear, whether a deterministic term might still have an infinite answer set, e.g. $\mathtt{ans}(\mathbf{K\,I})$ is obviously *not* a singleton. However, it contains $\lambda y.\odot$ only because (stop) could be applied to $\mathbf{I}$ although it is not necessary. Hence, analysing the answer set of deterministic terms should give the result that its elements differ only by internal stop reductions. More formally:

Conjecture. *Let s be a deterministic term. Then $\lambda y.p, \lambda z.q \in \mathtt{ans}(s)$ implies*

$$\lambda y.p \xleftrightarrow{\;i,\; \mathcal{C},\; stop\;}^{*} \lambda z.q$$

We will illustrate how this property could be used to show that closed deterministic terms may be copied, i.e., mutual similarity is retained. The argument is based on an additional assumption, namely: For all $\xleftrightarrow{\;i,\; \mathcal{C},\; \mathrm{stop}\;}^{*}$-related abstractions from the answer set, there exists a further element which is greater w.r.t. similarity.

Hence, suppose a surface contexts $S \in \mathcal{S}$ and a closed term $t \in \Lambda^0_{\approx}$ that is deterministic. We will only explain how $\mathtt{let}\ x = t\ \mathtt{in}\ S[t] \precsim_b \mathtt{let}\ x = t\ \mathtt{in}\ S[x]$ could be justified since the opposite relation should be even simpler. Therefore $\mathtt{let}\ x = t\ \mathtt{in}\ S[t] \Downarrow \lambda z.q$ is assumed. By the normalisation from lemma 3.2.26 in section 3.2.5 the converging approximation reduction sequence ending in $\lambda z.q$ may be required to reduce $\mathtt{let}\ x = t\ \mathtt{in}\ S[t]$ to $\mathtt{let}\ x = \lambda y.p\ \mathtt{in}\ S[t]$ first. Furthermore, suppose the second occurrence of t in $S[t]$ to be necessary to reach the abstraction $\lambda z.q$. Then t has to be reduced to some abstraction $\lambda y.p'$. By the premise there is a $t \Downarrow \lambda y.p''$ such that $\lambda y.p \precsim_b \lambda y.p''$ as well as $\lambda y.p' \precsim_b \lambda y.p''$ is valid. Choosing this abstraction $\lambda y.p''$ for the reduction of t within $\mathtt{let}\ x = t\ \mathtt{in}\ S[x]$ establishes the claim since open similarity is a precongruence.

It is remarkable that the argument is based on the condition that the answer set of a deterministic term has a greatest element. In section 5.1 this condition will appear again.

Chapter 5

Contextual and Denotational Semantics

The intention of this work is to provide the fundamentals for using similarity as a proof tool, e.g., to show program transformations correct. Since correctness of program transformations is defined in terms of contextual equivalence, the inclusion of (open) similarity in contextual preorder is vital. Fortunately, as will be seen in section 5.1.2, this is no more subtle once open similarity has been established a precongruence in the previous chapter.

However, for the time being this consequence is only valid w.r.t. the contextual preorder of the $\lambda_\approx$-calculus, because similarity is exclusively defined therein. Hence, section 5.1.1 will first have to show that the respective contextual preorders of the calculi λ_{ND} and $\lambda_\approx$ indeed coincide. This is accomplished by using the Approximation Theorem from section 3.3, by which convergence in the calculus $\lambda_\approx$ conforms with the one of the λ_{ND}-calculus.

By means of the subsequent discussion of syntactic continuity, it becomes evident that mutual open similarity fails to completely match contextual equivalence. The observation that similarity is strictly contained in contextual preorder is due to Lassen, though the counter-example for the $\lambda_\approx$-calculus is, compared to [Las98b], by far more involved.

Thus, section 5.3 describes which parts of the theory need contextual preorder and for which open similarity suffices. Moreover, a condition is given under which open similarity and contextual preorder agree. The chapter concludes with section 5.4 that sketches a term model based on contextual equivalence.

5.1   Contextual (Pre-) Congruence

This section presents the link for similarity in the calculus $\boldsymbol{\lambda}_{\approx}$ as a tool for proving contextual equivalence in the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus. Therefore, we first recall the definition of contextual preorder and equivalence as well as briefly compare it to the corresponding notions from [Kut99] and [MSC99a]. From definition 2.4.5 contextual preorder is known as the following relation.

$$s \lesssim_c t \iff (\forall C \in \mathcal{C} : \ C[s] \Downarrow \implies \ C[t] \Downarrow) \tag{5.1.1}$$

That definition is valid for the $\boldsymbol{\lambda}_{\approx}$-calculus too, though $\odot$ enriches the sets of terms and contexts. Hence section 5.1.1 will also address the issue that $\odot$ does not add any computational power in more detail.

However, the above definition does not cover divergence, i.e., non-terminating reduction sequences. So, w.r.t. the definition used in this work, the terms $\mathbf{I}$ and $\mathtt{pick\ I\ \Omega}$ are contextually equivalent. But this is *not* true for the contextual equivalence of neither [Kut99] nor [MSC99a].

To see this, we consider the definition of contextual preorder in the calculus Λ_{let} of [Kut99] and in the $\Lambda_{\mathrm{NEED}}^{\oplus}$-calculus of [MSC99a] respectively. Let the symbol $\Downarrow\!\!\Downarrow$ stand for must convergence and $\Uparrow\!\!\Uparrow$ for may divergence.

$$
\begin{aligned}
s \ \lesssim_{\Lambda_{let}} \ t \ &\overset{\text{def}}{\iff} \ (\forall C \in \mathcal{C} : \ C[s] \Downarrow \implies \ C[t] \Downarrow) \ \wedge \\
&\qquad (\forall C \in \mathcal{C} : \ C[s] \Downarrow\!\!\Downarrow \implies \ C[t] \Downarrow\!\!\Downarrow)
\end{aligned}
$$

$$
\begin{aligned}
s \ \lesssim_{\Lambda_{\mathrm{NEED}}^{\oplus}} \ t \ &\overset{\text{def}}{\iff} \ (\forall C \in \mathcal{C} : \ C[s] \Downarrow \impliedby C[t] \Downarrow) \ \wedge \\
&\qquad (\forall C \in \mathcal{C} : \ C[s] \Uparrow\!\!\Uparrow \impliedby C[t] \Uparrow\!\!\Uparrow)
\end{aligned}
$$

As explained in the introduction to section 3.1.3, must convergence $\Downarrow\!\!\Downarrow$ and may divergence $\Uparrow\!\!\Uparrow$ are just logical counterparts, i.e., $s \Downarrow\!\!\Downarrow \iff s \not\Uparrow\!\!\Uparrow$ holds. So it is merely a matter of taste which one to choose thereof. Here, contrary to [Kut99] the representation with must convergence was preferred, because it provides a uniform look of the implications. Because of the above mentioned relation between must convergence and may divergence, both contextual preorders $\lesssim_{\Lambda_{let}}$ and $\lesssim_{\Lambda_{\mathrm{NEED}}^{\oplus}}$ apparently induce the same style of contextual equivalence. That is, two terms are contextually equivalent if and only if identical *converging and diverging* behaviour is observed in all contexts.

This is apparently not the case for the terms $\mathbf{I}$ and $\mathtt{pick\ I\ \Omega}$ as the latter has a non-terminating reduction within the empty context. It is interesting to note that in the $\Lambda_{\mathrm{NEED}}^{\oplus}$-calculus $\mathtt{pick\ I\ \Omega} \ \lesssim_{\Lambda_{\mathrm{NEED}}^{\oplus}} \ \mathbf{I}$ is true and also in

the Λ_{let}-calculus the relation $\texttt{pick I } \Omega \lesssim_{\Lambda_{let}} \textbf{I}$ holds. This is so because $\textbf{I}$ and $\texttt{pick I } \Omega$ are distinguished only by their non-terminating behaviour. The situation is completely different for the non-deterministic choice of two incomparable terms, e.g., $\texttt{pick K K2}$ and $\textbf{K}$ are not related by $\lesssim_{\Lambda_{let}}$ in any fashion but $\texttt{pick K K2} \lesssim_{\Lambda^{\oplus}_{\text{NEED}}} \textbf{K}$ holds. Though, the motivation for the design of the contextual preorder in [Kut99] is convincing: A term t should only be considered "better" than a term s if it does not additionally introduce non-termination.

However, such a view would cause $\texttt{pick K K2} \xrightarrow{\text{nd, left}} \textbf{K}$ to be a counterexample to terms becoming smaller w.r.t. contextual preorder during reduction. And throughout chapter 4 this property has been used extensively, simplifying many of the proofs. So when the calculus $\boldsymbol{\lambda}_{\text{ND}}$ would be extended to incorporate divergence, adopting the opposite relation $\gtrsim_{\Lambda^{\oplus}_{\text{NEED}}}$ of the contextual preorder from the $\Lambda^{\oplus}_{\text{NEED}}$-calculus seems most promising.

By the achievement of the aforementioned chapter 4, i.e., that it is a precongruence, open similarity will be shown to imply contextual preorder in section 5.1.2. But before this, the equivalence of the contextual preorders in $\boldsymbol{\lambda}_{\text{ND}}$ and $\boldsymbol{\lambda}_{\approx}$ is established.

5.1.1 Correspondence of Equality in $\boldsymbol{\lambda}_{\text{ND}}$ and $\boldsymbol{\lambda}_{\approx}$

From the Approximation Theorem in section 3.3 it has been learned that convergence in the calculi $\boldsymbol{\lambda}_{\text{ND}}$ and $\boldsymbol{\lambda}_{\approx}$ coincides. There, the normal-order reduction of the $\boldsymbol{\lambda}_{\text{ND}}$-calculus was already extended to $\boldsymbol{\lambda}_{\approx}$-terms by considering $\odot$ as a term without any normal-order reduction. Furthermore, it is not difficult to see that $\odot$ and $\boldsymbol{\Omega}$ are mutual similar in the $\boldsymbol{\lambda}_{\approx}$-calculus and therefore contextually equivalent by the results of the subsequent section. Hence, not only inside terms but also in contexts all occurrences of $\odot$ can be replaced with $\boldsymbol{\Omega}$ without changing their discriminating power. This means that apart from normal-order reduction also the contextual preorder may be transferred from the Λ_{ND}- to the $\Lambda_{\approx}$-language. Therefore, we may assume to completely operate within the language $\Lambda_{\approx}$ so that the lemma below becomes applicable.

It accomplishes the step from correspondence of convergence to contextual preorder. This is even possible in the general setting of a lazy computation system while its proof benefits from the fact that only may convergence has to be considered.

Lemma 5.1.1. *Let $\mathcal{L}_1, \mathcal{L}_2$ be two lazy computation systems, whose term sets are identical, i.e. $\mathcal{T}(\mathcal{L}_1) = \mathcal{T}(\mathcal{L}_2)$, with respective evaluation $\Downarrow_{\mathcal{L}_1}$ and $\Downarrow_{\mathcal{L}_2}$ and*

contextual approximation $\lesssim_{\mathcal{L}_1,c}$ and $\lesssim_{\mathcal{L}_2,c}$. Then $\lesssim_{\mathcal{L}_1,c} = \lesssim_{\mathcal{L}_2,c}$ if and only if $r \Downarrow_{\mathcal{L}_1} \iff r \Downarrow_{\mathcal{L}_2}$ for every term $r \in \mathcal{T}$ holds.

Proof. The "only-if"-part is trivial by choosing $C = [\ ]$ in the right hand side of (5.1.1). By symmetry, it suffices to show $\lesssim_{\mathcal{L}_1,c} \subseteq \lesssim_{\mathcal{L}_2,c}$ for the "if"-part. So assume terms $s, t \in \mathcal{T}$, arbitrary but fixed, and a context $C \in \mathcal{C}$ such that $s \lesssim_{\mathcal{L}_1,c} t$ and $C[s] \Downarrow_{\mathcal{L}_2}$ hold. By the premise $\forall r : \ r \Downarrow_{\mathcal{L}_1} \iff r \Downarrow_{\mathcal{L}_2}$, we also have $C[s] \Downarrow_{\mathcal{L}_1}$ and hence $C[t] \Downarrow_{\mathcal{L}_1}$ since $s \lesssim_{\mathcal{L}_1,c} t$ implies $C[s] \lesssim_{\mathcal{L}_1,c} C[t]$. Thus $C[t] \Downarrow_{\mathcal{L}_2}$ by the premise $\forall r : \ r \Downarrow_{\mathcal{L}_1} \iff r \Downarrow_{\mathcal{L}_2}$ again. $\qquad\square$

In conjunction with the Approximation Theorem it is now an easy consequence that the contextual congruences in $\boldsymbol{\lambda}_{\mathrm{ND}}$ and $\boldsymbol{\lambda}_{\approx}$ agree.

Theorem 5.1.2. *Let $s, t \in \Lambda_{\approx}$ be terms. Then $s \simeq_{\boldsymbol{\lambda}_{\mathrm{ND}},c} t$ iff $s \simeq_{\boldsymbol{\lambda}_{\approx},c} t$ is true.*

Proof. As explained before, $\Lambda_{\approx}$ is considered the common set of terms. Thus the claim follows from lemma 5.1.1 since $\forall r : \ r \Downarrow_{\boldsymbol{\lambda}_{\mathrm{ND}}} \iff r \Downarrow_{\boldsymbol{\lambda}_{\approx}}$ holds by the Approximation Theorem. $\qquad\square$

We will therefore drop the distinction between the two contextual preorders in $\boldsymbol{\lambda}_{\mathrm{ND}}$ and $\boldsymbol{\lambda}_{\approx}$ for the rest of this work.

5.1.2   Open Similarity implies Contextual Preorder

The last step for establishing mutual open similarity as a method to show contextual equivalences in the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus is now the inclusion of open similarity in the contextual preorder. The prerequisites for this are provided in chapter 4, namely by the Precongruence Theorem, which states that open similarity is a precongruence, and by proposition 4.1.37 about convergence on open terms.

Main Theorem 5.1.3. *For all terms $s, t \in \Lambda_{\approx}$ the following holds:*

$$s \lesssim_b{}^o t \implies s \lesssim_c t$$

Proof. Since $\lesssim_b{}^o$ is a precongruence by the Precongruence Theorem and $\lesssim_b$ is by definition a simulation, according to theorem 2.4.7 the prerequisite that $(\cdot)^o$ qualifies for preservation of convergence has to be established only.

The proof is then as follows. Assuming terms $s, t \in \Lambda_{\approx}$ and an arbitrary context $C \in \mathcal{C}$ satisfying $s \lesssim_b{}^o t$ and $C[s] \Downarrow$ it is to show that $C[t] \Downarrow$ holds, too. From $s \lesssim_b{}^o t$ we have $C[s] \lesssim_b{}^o C[t]$ since $\lesssim_b{}^o$ is a precongruence. By definition 4.1.3 of the open extension this means $\sigma(C[s]) \ \lesssim_b \ \sigma(C[t])$ for all closing substitutions σ that map only to $\circledcirc$ or a closed abstraction.

This is true in particular for the substitution $\sigma_\odot$ that yields $\odot$ for all variables. Furthermore, $C[s] \Downarrow$ implies $\sigma_\odot(C[s]) \Downarrow$ by proposition 4.1.37 from which, in connection with $\sigma_\odot(C[s]) \precsim_b \sigma_\odot(C[t])$, we have $\sigma_\odot(C[t]) \Downarrow$ in turn. Thence, by proposition 4.1.37 again, $C[t] \Downarrow$ holds. $\square$

It easily follows that approximation reduction makes terms smaller w.r.t. contextual preorder.

Corollary 5.1.4. *Let* $s, t \in \Lambda_\approx$ *be terms with* $s \xrightarrow{\mathcal{S}}{}^{*}_{\boldsymbol{\lambda}_\approx} t$. *Then* $t \precsim_c s$ *is true.*

Proof. By corollary 4.1.40 in connection with the Main Theorem. $\square$

By the results from section 3.3 and section 4.1.4 the previous corollary may be carried over to normal-order reduction.

Corollary 5.1.5. *Let* $s, t \in \Lambda_\approx$ *be terms with* $s \xrightarrow{n}{}^{*}_{\boldsymbol{\lambda}_{\mathrm{ND}}} t$. *Then* $t \precsim_c s$ *is true.*

However, the current section is entitled that open similarity *implies* contextual preorder which means that the inclusion $\precsim_c \subseteq \precsim_b{}^o$ is in general *not* valid. If it was, then also $(\precsim_c)_0 \subseteq (\precsim_b{}^o)_0 = \precsim_b$ would be, since $(\cdot)_0$ is monotonic and $(\cdot)^o$ admissible. So, by the following proposition open similarity is strictly contained in contextual preorder.

Proposition 5.1.6. *There exist closed terms* $s, t \in \Lambda_\approx^0$ *such that* $s \; (\precsim_c)_0 \; t$ *holds but* $s \precsim_b t$ *does not.*

Involving the notion of syntactic continuity, the proof of this proposition is quite complex and thus deferred to the next section.

5.2   Syntactic Continuity

It is well-known, cf. [MST96], that *not* every least upper bound in $\Lambda_\approx$ is continuous w.r.t. contexts. I.e., there exist infinitely ascending chains (s_i) which have a least upper bound, say t, but $C[t]$ is not the least upper bound of the chain $(C[s_i])$ where C is some context. When least upper bounds should be considered only, which are continuous w.r.t. contexts, the notion of a *contextual least upper bound* proves useful, cf. [Sch00, SS03a]. Suppose $\precsim$ to be some preorder.

Definition 5.2.1. *Let* (s_i) *be an ascending chain. Then* t *is the* contextual least upper bound *of* (s_i), *denoted* $t = \bigsqcup_i^{\mathcal{C}} s_i$, *iff the following condition is met.*

$$\forall C : \; (\forall i : \; C[s_i] \precsim C[s]) \; \wedge \; (\forall t : \; (\forall i : \; C[s_i] \precsim t) \implies C[s] \precsim t)$$

Usually the combinator $\mathbf{Y}$ shall be used to represent recursion. Therefore, it is crucial that $\mathbf{Y}\,f$ forms the contextual least upper bound of the chain $(f^i\,\circledcirc)$ where f^i denotes the i-fold application of the term f which will be defined below. This property is called *syntactic continuity*, i.e., $\mathbf{Y}\,f$ is the least upper bound of the chain $(f^i\,\circledcirc)$ and continuous w.r.t. contexts, cf. [Pit97, San97, Las98b].

Definition 5.2.2. *Let $s, t \in \Lambda_\approx$ be terms and $i \in \mathbb{N}$ a natural number. Then $s^i\,t$, the i-fold application of s to t, is inductively defined as follows:*

$$s^0\,t \equiv t$$
$$s^{i+1}\,t \equiv s\,(s^i\,t)$$

The remainder of this section treats syntactic continuity w.r.t. contextual preorder and open similarity respectively. For the former it holds, whereas regarding the latter a counter-example is being devised.

5.2.1   Contextual Preorder

In order to show that syntactic continuity is valid w.r.t. the contextual preorder, a context lemma for the $\boldsymbol{\lambda}_\approx$-calculus will be established first. By means of the Surface Context Lemma it is sufficient to consider the converging behaviour of terms within surface contexts only.

Lemma 5.2.3 (Surface Context Lemma). *Let $T \subseteq \Lambda_\approx$ be a countable set of terms and $s \in \Lambda_\approx$ a term satisfying the following: For every surface context $S \in \mathcal{S}$ with $S[s] \Downarrow$ there is a term $t \in T$ such that $S[t] \Downarrow$ holds.*

Then this property is also valid for general contexts, i.e. for every context $C \in \mathcal{C}$ with $C[s] \Downarrow$ there exists $t \in T$ such that $C[t] \Downarrow$ is true.

Proof. Given a term s such that $\forall S \in \mathcal{S}: S[s] \Downarrow \implies \exists t \in T: S[t] \Downarrow$ holds. For all terms $r \in \Lambda_\approx$ we have $r \Downarrow_{\mathrm{ND}} \iff r \Downarrow_{\boldsymbol{\lambda}_\approx}$ by the Approximation Theorem. This is why the above implication is equivalent to

$$\forall S \in \mathcal{S}: \ S[s] \Downarrow_{\mathrm{ND}} \implies \exists t \in T: \ S[t] \Downarrow_{\mathrm{ND}}$$

As every reduction context is also a surface context, lemma 3.1.16 applies.   □

Remark 5.2.4. *Note that, even if restricted to closed terms, the above lemma cannot be extended to the case that $C[s]$ and $C[t]$, respectively $S[s]$ and $S[t]$, converge to the* same *abstractions.*

A counter-example is $s \equiv \mathtt{pick}\ \mathbf{K}\ \mathbf{K2}$ *and* $T = \{\mathbf{K}, \mathbf{K2}\}$. *By lemma 3.2.20, since* s *is closed, for every* $S \in \mathcal{S}$ *such that* $S[s] \Downarrow \lambda z.q$ *either* $S[\mathbf{K}] \Downarrow \lambda z.q$ *or* $S[\mathbf{K2}] \Downarrow \lambda z.q$ *holds. But* $\lambda x.(\mathtt{pick}\ \mathbf{K}\ \mathbf{K2})$ *is not contextually equivalent to* $\mathtt{pick}\ (\lambda x.\mathbf{K})\ (\lambda x.\mathbf{K2})$ *and hence neither* $\lambda x.\mathbf{K} \Downarrow \lambda x.s$ *nor* $\lambda x.\mathbf{K2} \Downarrow \lambda x.s$ *is true.*

Now we will proceed as follows. Basically, the subsequent lemma will establish that $\mathbf{Y}\,f$ can be computed by the chain $(f^i\,\circledcirc)$, i.e., for all possible outcomes of $\mathbf{Y}\,f$ there is also an index i such that $f^i\,\circledcirc$ converges to the same abstraction. Afterwards, this result is extended to reductions within surface contexts. Using the Surface Context Lemma, it is eventually transferred to arbitrary contexts which yields syntactic continuity.

Lemma 5.2.5. *Let* $f \equiv \lambda x.s \in \Lambda^0_{\approx}$ *designate a closed abstraction. Then for every abstraction the term* $(\lambda y.f\,(y\,y))\,(\lambda z.f\,(z\,z))$ *converges to, there is a natural number* $i \in \mathbb{N}$ *such that* $f^i\,\circledcirc$ *reduces to the same abstraction.*

Proof. By induction on the length n of the converging approximation sequence for the term $(\lambda y.f\,(y\,y))\,(\lambda z.f\,(z\,z))$. We therefore distinguish on the first reduction of this sequence:

- Clearly $(\lambda y.f\,(y\,y))\,(\lambda z.f\,(z\,z)) \xrightarrow{\mathcal{S},\ \mathrm{stop}} \circledcirc$ may be ruled out, since $\circledcirc$ has no approximation reduction to an abstraction.

- By lemma 3.2.14, the case $(\lambda y.f\,(y\,y))\,(\lambda z.f\,(z\,z)) \xrightarrow{\mathcal{S},\ \mathrm{stop}} \circledcirc\,(\lambda z.f\,(z\,z))$ behaves identically to the previous one.

- The reduction $(\lambda y.f\,(y\,y))\,(\lambda z.f\,(z\,z)) \xrightarrow{\mathcal{S},\ \mathrm{stop}} (\lambda y.f\,(y\,y))\,\circledcirc$ may only be followed by (lbeta) to reach an abstraction — otherwise one of the previous cases would arise. This sequence $\xrightarrow{\mathcal{S},\ \mathrm{stop}} \cdot \xrightarrow{\mathcal{S},\ \mathrm{lbeta}}$ obviously commutes, i.e. leads to the same abstraction as $\xrightarrow{\mathcal{S},\ \mathrm{lbeta}} \cdot \xrightarrow{\mathcal{S},\ \mathrm{stop}}$ which forms the induction base.

  The common result of both sequences is $\mathtt{let}\ y = \circledcirc\ \mathtt{in}\ f\,(y\,y)$ whose reduction may continue with (cpa) without changing the length of the sequence by lemma 3.2.26. Thus $\mathtt{let}\ y = \circledcirc\ \mathtt{in}\ f\,(y\,y) \xrightarrow{\mathrm{cpa}} f\,(\circledcirc\,\circledcirc)$ has the same approximation reduction to an abstraction and so has $f^1\,\circledcirc$ by lemma 3.2.14, which establishes the claim.

- The induction step is given when the use of (stop) is avoided at this early stage. Then only (lbeta) is possible at first:

$$(\lambda y.f\,(y\,y))\,(\lambda z.f\,(z\,z)) \xrightarrow{[\ ],\ \mathrm{lbeta}} \mathtt{let}\ y = (\lambda z.f\,(z\,z))\ \mathtt{in}\ f\,(y\,y)$$

By lemma 3.2.26, we now may perform (cpa) without changing the length of the sequence. Recalling $f \equiv \lambda x.s$, the possible non-(stop)-reductions are all of type (lbeta) and may clearly be commuted. Hence we may assume:

$$(\lambda y.f\,(y\,y))\,(\lambda z.f\,(z\,z))$$

$$\xrightarrow{[\,],\ \text{lbeta}} \texttt{let } y = (\lambda z.f\,(z\,z)) \texttt{ in } f\,(y\,y)$$

$$\xrightarrow{[\,],\ \text{cpa}} f\,((\lambda z'.f\,(z'\,z'))\,(\lambda z''.f\,(z''\,z'')))$$

$$\xrightarrow{[\,],\ \text{lbeta}} \texttt{let } x = ((\lambda z'.f\,(z'\,z'))\,(\lambda z''.f\,(z''\,z''))) \texttt{ in } s$$

By lemma 3.2.26 again, we may assume that reduction first proceeds inside the context $\texttt{let } x = [\,] \texttt{ in } s$ without changing the length of the sequence at all. But then, since $(\lambda z'.f\,(z'\,z'))\,(\lambda z''.f\,(z''\,z''))$ is syntactically equal to the original term, we may apply the induction hypothesis to it.

Hence there is a natural number i such that $f^i \odot$ reduces to the same abstraction as $(\lambda z'.f\,(z'\,z'))\,(\lambda z''.f\,(z''\,z''))$ converges to. Thus the claim is shown since $f^{i+1} \odot \xrightarrow{\text{lbeta}} \texttt{let } x = (f^i \odot) \texttt{ in } s$ holds.   ☐

The reductions performed in the above lemma may also be embedded into surface contexts, as can be shown using evaluation at strict positions.

Lemma 5.2.6. *Let $f \equiv \lambda y.p \in \Lambda^0_\approx$ be a closed abstraction and $S \in \mathcal{S}$ a surface context. Then $S[\mathbf{Y}\,f] \Downarrow \lambda z.q$ implies that there is a natural number $i \in \mathbb{N}$ such that also $S[f^i \odot]$ has a reduction to $\lambda z.q$, i.e. to the same abstraction.*

Proof. Assuming $S[\mathbf{Y}\,f] \Downarrow \lambda z.q$ lemma 3.2.20 is applicable, since $\mathbf{Y}$ and f both are closed. Hence we have $\lambda x.s$ such that $\mathbf{Y}\,f \Downarrow \lambda x.s$ and $S[\lambda x.s] \Downarrow \lambda z.q$ hold. Since for $\mathbf{Y}\,f \Downarrow \lambda x.s$ all sensible reduction sequences have to start with $\mathbf{Y}\,f \xrightarrow{\mathcal{S}\ *}_{\boldsymbol{\lambda}_\approx} (\lambda y.f\,(y\,y))\,(\lambda z.f\,(z\,z))$ lemma 5.2.5 may be used. Therefore, there is a natural number i such that $f^i \odot \xrightarrow{\mathcal{S}\ *}_{\boldsymbol{\lambda}_\approx} \lambda x.s$ too. This approximation reduction may be performed within the surface context S, thus $S[f^i \odot] \xrightarrow{\mathcal{S}\ *}_{\boldsymbol{\lambda}_\approx} S[\lambda x.s] \xrightarrow{\mathcal{S}\ *}_{\boldsymbol{\lambda}_\approx} \lambda z.q$ proves the claim.   ☐

Combining the previous lemma with the Surface Context Lemma comprises an essential step towards syntactic continuity.

Lemma 5.2.7. *Let $f \equiv \lambda x.s \in \Lambda^0_\approx$ be a closed abstraction and $C \in \mathcal{C}$ an arbitrary context. Then $C[\mathbf{Y}\,f] \Downarrow \implies \exists i \in \mathbb{N}: C[f^i \odot] \Downarrow$ is valid.*

Proof. Assume $f \equiv \lambda x.s \in \Lambda^0_\approx$ to be a closed abstraction. Then, by lemma 5.2.6, the implication $\forall S \in \mathcal{S} : S[\mathbf{Y} f] \Downarrow \implies \exists i \in \mathbb{N} : S[f^i \odot] \Downarrow$ holds. Therewith, the preconditions for lemma 5.2.3 are met. $\qquad \square$

We are now in a position to prove the following formulation of syntactic continuity which is also found in [Pit97, San97, Las98b]. Deriving $\mathbf{Y} f$ to be the contextual least upper bound of the ascending chain $(f^i \odot)$ is easy therefrom.

Theorem 5.2.8 (Syntactic Continuity for $\lesssim_c$). *Let $f \equiv \lambda x.s \in \Lambda^0_\approx$ be a closed abstraction. Then for every context $C \in \mathcal{C}$ and for every term $t \in \Lambda_\approx$ the equivalence $C[\mathbf{Y} f] \lesssim_c t \iff \forall i : C[f^i \odot] \lesssim_c t$ is true.*

Proof. The "if"-part of the equivalence is straightforward, hence solely the "only-if"-part is shown. Therefore, assume the implication $\forall i : C[f^i \odot] \lesssim_c t$ to hold. Furthermore, suppose $D[C[\mathbf{Y} f]] \Downarrow$ in order to establish $D[t] \Downarrow$ for some arbitrary context D as required by the definition of the contextual preorder. From lemma 5.2.7 a natural number i is obtained such that $D[C[f^i \odot]] \Downarrow$ is true. By the premise, $D[t] \Downarrow$ is implied for every i, thus the claim holds. $\qquad \square$

What follows, is the discussion of a special case corresponding to the counter-example that will be given for open similarity. Since similarity tests for abstractions, we will put $\mathbf{Y} f$ under an abstraction $\lambda z.[\ ]$ and exploit that the contextual least upper bound of the ascending chain $\lambda z.(f^i \odot)$ is obtained.

Corollary 5.2.9. *Let $f \equiv \lambda x.s \in \Lambda^0_\approx$ be closed. Then for all contexts $C \in \mathcal{C}$ with $C[\lambda z.(\mathbf{Y} f)] \Downarrow$ there is a $\lambda z.(f^i \odot)$ such $C[\lambda z.(f^i \odot)] \Downarrow$ holds.*

Proof. Let C be an arbitrary context such that $C[\lambda z.(\mathbf{Y} f)]$ converges. For the context $D \equiv C[\lambda z.[\ ]]$ we obtain $D[\mathbf{Y} f] \Downarrow$ since $C[\lambda z.(\mathbf{Y} f)] \equiv D[\mathbf{Y} f]$. Hence, by lemma 5.2.7, there exists a $f^i \odot$ such that $D[f^i \odot] \Downarrow$ which in fact is $C[\lambda z.(f^i \odot)] \Downarrow$ as desired. $\qquad \square$

A term is needed that produces just the chain $\lambda z.(f^i \odot)$ as its answer set.

Notation. *Whenever f denotes some closed abstraction, let G_f stand for the closed term $G_f \equiv (\mathbf{Y} (\lambda g.\lambda x.\mathtt{pick} (\lambda z.x) (g (f x)))) \odot$ in the following.*

Lemma 5.2.10. *Let $f \in \Lambda^0_\approx$ be a closed abstraction. Then for $\mathtt{ans}(G_f)$, the answer set, $\mathtt{ans}(G_f) = \{ \lambda z.(f^i \odot) \mid i \in \mathbb{N} \}$ holds.*

Proof. By induction on the number i of (nd, right)-reductions in a converging approximation reduction $G_f \xrightarrow{\mathcal{S}}{}^*_{\boldsymbol{\lambda}_\approx} \lambda z.q$ we will show that $q \equiv f^i \circledcirc$ holds. We therefore unfold the beginning of such a sequence, where we disregard (stop)-reductions, since they will not contribute anything to an abstraction:

$$G_f \equiv (\mathbf{Y}\,(\lambda g.\lambda x.\mathtt{pick}\,(\lambda z.x)\,(g\,(f\,x))))\,\circledcirc$$

$$\xrightarrow{\text{lbeta}} (\mathtt{let}\ h = (\lambda g.\lambda x.\mathtt{pick}\,(\lambda z.x)\,(g\,(f\,x)))\ \mathtt{in}\ (\lambda y.h\,(y\,y))\,(\lambda y'.h\,(y'\,y')))\,\circledcirc$$

$$\xrightarrow{\text{cpa}} ((\lambda y.(\lambda g.\lambda x.\mathtt{pick}\,(\lambda z.x)\,(g\,(f\,x)))\,(y\,y))\,(\lambda y'.(\lambda g....)\,(y'\,y')))\,\circledcirc$$

$$\xrightarrow{\text{lbeta}} \ldots$$

$$\xrightarrow{\text{cpa}} ((\lambda g.\lambda x.\mathtt{pick}\,(\lambda z.x)\,(g\,(f\,x)))\,((\lambda y'.(\lambda g....)\,(y'\,y'))\ldots))\,\circledcirc$$

$$\xrightarrow{\mathcal{S}}{}^*_{\boldsymbol{\lambda}_\approx} \mathtt{pick}\,(\lambda z.\circledcirc)\,w$$

The induction base is given by $\lambda z.\circledcirc$ whereas the term w is of a form so that the induction hypothesis may be applied. □

Now it can be shown that G_f may approximate $\lambda z.(\mathbf{Y}\,f)$ arbitrarily precise.

Lemma 5.2.11. *Let f be a closed abstraction. Then $\lambda z.(\mathbf{Y}\,f) \lesssim_c G_f$ is true.*

Proof. Assuming $f \equiv \lambda x.s \in \Lambda^0_\approx$ and a context C satisfying $C[\lambda z.(\mathbf{Y}\,f)] \Downarrow$, we have to show that $C[G_f] \Downarrow$ holds, too. By corollary 5.2.9 there is a number i such that $C[\lambda z.f^i \circledcirc]$ converges. Furthermore, by lemma 5.2.10, the reduction $G_f \xrightarrow{\mathcal{S}}{}^*_{\boldsymbol{\lambda}_\approx} \lambda z.(f^i \circledcirc)$ exists, hence $\lambda z.(f^i \circledcirc) \lesssim_b G_f$ by corollary 4.1.12. From this we have $C[\lambda z.(f^i \circledcirc)] \lesssim_b C[G_f]$ since $\lesssim_b$ is a precongruence, and thus, by the definition of $\lesssim_b$, the claim holds. □

For similarity, the specific case $f \equiv \mathbf{K}$ will be considered as well.

Corollary 5.2.12. *The inequation $\lambda z.(\mathbf{Y}\,\mathbf{K})\,(\lesssim_c)_0\,G_{\mathbf{K}}$ is true.*

5.2.2   Similarity

In order to show $s\ [\eta]_\approx\ t$ the definition of an experiment requires to *fix* for the term t a corresponding abstraction for every abstraction s converges to. This obviously will not be possible if $\lambda z.(\mathbf{Y}\,\mathbf{K})$ and the ascending chain $(\lambda z.(\mathbf{K}^i \circledcirc))$ are compared. The crucial point is that the term $G_{\mathbf{K}}$ constructs this chain.

Lemma 5.2.13. *The inequation $\lambda z.(\mathbf{Y}\,\mathbf{K}) \lesssim_b G_{\mathbf{K}}$ is false.*

Proof. Assume $\lambda z.(\mathbf{Y}\,\mathbf{K}) \lesssim_b G_{\mathbf{K}}$ for a proof by contradiction. Since $\lambda z.(\mathbf{Y}\,\mathbf{K})$ already is an abstraction, there has to be a closed abstraction $\lambda z.q \in \Lambda^0_{\approx}$ such that $G_{\mathbf{K}} \Downarrow \lambda z.q$ and $\lambda z.(\mathbf{Y}\,\mathbf{K}) \lesssim_b \lambda z.q$ hold. By lemma 5.2.10, q has to be of the form $q \equiv \lambda z.(\mathbf{K}^i\,\circledcirc)$. Fixing some i we obtain $(\lambda z.(\mathbf{Y}\,\mathbf{K}))\,\circledcirc\ldots\circledcirc \Downarrow$ for $i+2$ many applications to $\circledcirc$, but $(\lambda z.\mathbf{K}^i\,\circledcirc)\,\circledcirc\ldots\circledcirc$, with $\circledcirc\ldots\circledcirc$ representing the same argument sequence, does not converge. $\qquad\square$

     This has an easy but significant consequence.

Corollary 5.2.14. *Syntactic continuity is not valid w.r.t. similarity.*

     Furthermore, we may restate and finally prove proposition 5.1.6 which implies that open similarity is strictly contained in contextual preorder.

Proposition. There are $s, t \in \Lambda^0_{\approx}$ so that $s\,(\lesssim_c)_0\,t$ holds but $s \lesssim_b t$ does not.

Proof of Proposition 5.1.6. By corollary 5.2.12 and lemma 5.2.13. $\qquad\qquad\square$

5.3 (In-) Equational Theory

This section describes the theory generated by contextual preorder and equivalence respectively. So an overview will be given which equations and inequations respectively are valid in the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus. For this, open similarity is a quite powerful tool. However, because of its strict inclusion in contextual preorder, logical implications that are valid w.r.t. open similarity can become false for contextual preorder.

     Therefore, we will also look for conditions under which contextual preorder is sufficient to establish open similarity. E.g., it should be stressed that open similarity matches contextual preorder whenever the answer set already contains its own contextual least upper bound, i.e., the answer set has a greatest element. In order to establish this result, it has to be shown first, that such an answer is contextually equivalent to the original term in this case.

Lemma 5.3.1. *Let $s \in \Lambda^0_{\approx}$ be a closed term such that $\mathtt{ans}(s)$ possesses a maximum w.r.t. $\lesssim_c$, denoted by $\lambda x.t$. Then $s\,(\simeq_c)_0\,\lambda x.t$ is true.*

Proof. We have $\lambda x.t \lesssim_c s$ by corollary 5.1.4, so only the converse needs to be proven. By the surface context lemma 5.2.3 it is sufficient to prove the implication $\forall S:\ S[s] \Downarrow \implies S[\lambda x.t] \Downarrow$. Therefore, we assume an arbitrary surface context $S \in \mathcal{S}$ such that $S[s] \Downarrow$ holds. We then obtain, by lemma 3.2.20, some abstraction $\lambda y.p$ such that both $s \Downarrow \lambda y.p$ and $S[\lambda y.p] \Downarrow \lambda z.q$ are satisfied.

By the premise, $\lambda x.t$ is the maximum of $\mathtt{ans}(s)$ w.r.t. $\lesssim_c$ and we thus have $\lambda y.p \lesssim_c \lambda x.t$ from which, by the definition of $\lesssim_c$, the claim follows. $\square$

Proposition 5.3.2. *Let $s', t' \in \Lambda^0_{\approx}$ be closed terms such that $\mathtt{ans}(t')$ has a greatest element w.r.t. the $\lesssim_c$-ordering. Then $s'\ (\lesssim_c)_0\ t' \implies s' \lesssim_b t'$ holds.*

Proof. By $\lambda x.t$ let the maximum of $\mathtt{ans}(t')$ be denoted. Proving $s' \lesssim_b t'$ by coinduction requires the condition $(\lesssim_c)_0 \subseteq [(\lesssim_c)_0]_{\approx}$ to be established. So assume $s'\ (\lesssim_c)_0\ t'$ and according[1] to (4.1.4) show

$$\forall \lambda y.p \in \mathtt{ans}(s') :\ \exists \lambda z.q \in \mathtt{ans}(t') :\ \forall r \in \Lambda^0_{\approx} :\ (\lambda y.p)\, r\ (\lesssim_c)_0\ (\lambda z.q)\, r$$

Let $s' \Downarrow \lambda y.p$ and as $t' \Downarrow \lambda x.t$ holds, only $\forall r \in \Lambda^0_{\approx} :\ (\lambda y.p)\, r\ (\lesssim_c)_0\ (\lambda x.t)\, r$ remains to be shown. From $s'\ (\lesssim_c)_0\ t'$ we obtain $\lambda y.p\ (\lesssim_c)_0\ t'$ by corollary 5.1.4. Since $\lambda x.t$ is the maximum of $\mathtt{ans}(t')$ we thus have $\lambda y.p\ (\lesssim_c)_0\ \lambda x.t$ by lemma 5.3.1 and the claim follows since $\lesssim_c$ is a precongruence. $\square$

Remark 5.3.3. *The above proof is based on the argument that $\lesssim_b$ could equally well have been defined as the greatest fixed point of an experiment*

$$s\ [\eta]_{\approx}\ t \iff \forall \lambda x.s' \in \mathtt{ans}(s) :\ \exists \lambda x.t' \in \mathtt{ans}(t) :$$
$$\forall p \in \Lambda^0_{\approx} :\ (\lambda x.s')\, p \quad \eta \quad (\lambda x.t')\, p$$

Instead, for $\lesssim_c$ also a counterpart to proposition 4.1.27 could have been established. But that is not the intention of this section since a lot of the foregoing argumentation would be repeated.

For illustration purposes we give a separate proof for the corresponding statement of lemma 5.3.1 w.r.t. similarity.

Lemma 5.3.4. *Let $s \in \Lambda^0_{\approx}$ be a closed term such that $\mathtt{ans}(s)$ possesses w.r.t. $\lesssim_b$ a greatest element, denoted by $\lambda x.t$. Then $s \simeq_b \lambda x.t$ is valid.*

Proof. We have $\lambda x.t \lesssim_b s$ by corollary 4.1.12, so only the converse needs to be proven. We therefore assume $s \Downarrow \lambda x.s'$ and will show that $(\lambda x.s')\, r \lesssim_b (\lambda x.t)\, r$ for an arbitrary closed $r \in \Lambda^0_{\approx}$ holds. By the premises $\lambda x.s' \lesssim_b \lambda x.t$ since $\lambda x.s' \in \mathtt{ans}(s)$ and thus by corollary 4.1.28 the claim follows. $\square$

---

[1]As mentioned there, already the *experiment* also could have been formulated this way.

The implication $r \lesssim_b t \wedge s \lesssim_b t \implies \mathtt{pick}\ r\ s \lesssim_b t$ from example 4.1.10 is one of the typical cases where similarity must *not* be replaced simply by contextual preorder. Though the result may be transferred if $\mathtt{ans}(t)$ has a maximum, it would be more appropriate to rework the example w.r.t. contextual preorder. In particular, as all results of section 3.2.5 concerning rearrangement of reduction sequences likewise apply to direct proofs of contextual equivalence. The same goes for $(s \lesssim_c t \vee t \lesssim_c s) \implies \lambda x.\mathtt{pick}\ s\ t \lesssim_c \mathtt{pick}\ (\lambda x.s)\ (\lambda x.t)$ according to example 4.3.14. This condition can be shown to hold for closed terms w.r.t. contextual preorder by means of evaluation at strict positions in conjunction with the Surface Context Lemma.

However, for the opposite inequation it does work, since no restrictions are imposed on the terms.

$$\mathtt{pick}\ (\lambda x.s)\ (\lambda x.t) \lesssim_c \lambda x.\mathtt{pick}\ s\ t \tag{5.3.1}$$

Because of compatibility with contexts, this is just an instance of the following statement and its symmetric variant which both follow from corollary 5.1.4.

$$\forall t: \ s \lesssim_c \mathtt{pick}\ s\ t \tag{5.3.2}$$

Establishing $\mathtt{pick}\ t\ t \lesssim_b^o t$ for open similarity is trivial and thus $\mathtt{pick}$ could be said to be idempotent by the equation below.

$$t \simeq_c \mathtt{pick}\ t\ t \tag{5.3.3}$$

Similar to example 3.1.18 and 4.1.22 respectively, shifting λ over $\mathtt{pick}$ is obviously not correct in general, since[2] abstractions are the only terms that may be copied. So more than (5.3.1) and the conditional version mentioned above could not be expected in this regard.

But when switching from a λ-context to surface contexts, the equality for a $\mathtt{pick}$-shift holds unconditionally.

$$S[\mathtt{pick}\ s\ t] \simeq_c \mathtt{pick}\ S[s]\ S[t] \tag{5.3.4}$$

The proof is an easy application of the results from section 3.2.5 on the rearrangement of reduction sequences. Another interesting example is

$$s \lesssim_c t \implies \mathtt{pick}\ s\ t \simeq_c t \tag{5.3.5}$$

which follows from (5.3.2) in connection with (5.3.3) when the premise $s \lesssim_c t$ is inserted into the context $\mathtt{pick}\ [\ ]\ t$. More equations and inequations could be derived using the ideas which have been presented in this section.

---

[2]in the original $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus

5.4   Denotational Semantics

This section aims at describing how a denotational semantics, i.e., a term model based on contextual equivalence, for the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus could look like. An in-depth treatment of denotational semantics goes beyond the scope of this work. The reader is therefore referred to the literature, e.g. [Sch86, Mos90] for a more general introduction to denotational semantics and [Bar84, Dav89] for the construction of lambda calculus models.

Roughly, in denotational semantics every language construct is mapped to a mathematical object. So our primary task will be to talk about which domain these objects should be taken from. For a so-called *term model*, that will be discussed here, this domain is usually the quotient of the set of terms w.r.t. some equivalence relation. The equivalence, which will be used here, is contextual equivalence, of course. This should relieve some of the work necessary to obtain a fully abstract denotational semantics, i.e., a model whose equalities agree with contextual equivalence.

However, solely from dealing with terms modulo contextual equivalence, one does not gain more information. The reason is that a key feature of denotational semantics is not supported very well, namely compositionality. The notion means that the value of compound terms should only depend on the values of its components. This is essential, since denotational semantics emphasises the mathematical value a term is mapped to rather than the way it is computed. But if a term was mapped to its $\simeq_c$-equivalence class, how would e.g. the equivalence class of an application $s\,t$ be computed from the equivalence classes of the subterms? We will return to this issue after defining the necessary notions where $\mathbb{D}$ stands for some suitable domain.

Definition 5.4.1 (Variable Assignment). *A mapping $\rho : V \to \mathbb{D}$ from variables to objects of the domain $\mathbb{D}$ is called* variable assignment.

Definition 5.4.2 (Denotation). *Let $t \in \Lambda_{\approx}$ be a term and $\rho : V \to \mathbb{D}$ a variable assignment. Then $[\![t]\!]\,\rho \in \mathbb{D}$ is said to be the* denotation *of the term t under the variable assignment ρ. We call the denotations of two terms* equal, *written $[\![s]\!] = [\![t]\!]$ if and only if $[\![s]\!]\,\rho = [\![t]\!]\,\rho$ for all variable assignments ρ holds.*

Definition 5.4.3 (Adequacy). *A denotational semantics is called* adequate *if and only if all its equalities are justified by contextual equivalence, i.e., for all terms $s, t \in \Lambda_{\approx}$ the implication $[\![s]\!] = [\![t]\!] \implies s \simeq_c t$ is true.*

Definition 5.4.4 (Full Abstraction). *A denotational semantics is said to be* fully abstract *if it is adequate and does not distinguish terms which are contextually equal. That is to say,* $[\![s]\!] = [\![t]\!]$ *if and only if* $s \simeq_c t$ *holds.*

5.4.1 The Domain

In a lazy theory, cf. [Abr90, RDRP04], weak head normal forms are usually regarded as the meaning of terms. Like [Abr90], for the calculus $\boldsymbol{\lambda}_{\approx}$ a weak head normal form is simply an abstraction. But since $\boldsymbol{\lambda}_{\approx}$ is a non-deterministic calculus, $\Lambda_{\approx}$-terms could reduce to *several* weak head normal forms and thus sets of abstractions have to be used as elements of the domain.

However, a valuable insight for constructing a denotational domain is provided by the failure of open similarity to match contextual preorder completely. The reason of this failure demonstrates that it is *not* sufficient to consider only the abstractions a term converges to. More precise, the structure of the counter-example $\lambda z.(\mathbf{Y}\,\mathbf{K}) \not\lesssim_b G_{\mathbf{K}}$ and proposition 5.3.2 indicate that a suitable definition has to take into account contextual least upper bounds, cf. section 5.2.

Furthermore, because of the condition $s \lesssim_c t \implies \texttt{pick}\,s\,t \simeq_c t$ from (5.3.5), only down-closed sets should be used as denotations. I.e., a term is contained only if all of its derivatives, where a subterm is replaced with a smaller term, are included, too. Otherwise $[\![\texttt{pick}\,s\,t]\!]$ could be distinguished from $[\![t]\!]$ in the model. One advantage of the domain will be, that ordinary subset inclusion can be used as an ordering.

For a closed term $t \in \Lambda_{\approx}^0$, let $[t]_{\simeq_c}$ stand for its $(\simeq_c)_0$-equivalence class. The class $\perp \stackrel{\text{def}}{=} [\boldsymbol{\Omega}]_{\simeq_c}$ represents just the set $\{t \in \Lambda_{\approx}^0 \mid t \not\Downarrow\}$ of non-converging closed terms. To make notation easier, we understand the answer set as being extended to sets of terms, i.e., $\texttt{ans}(S) = \bigcup \{\texttt{ans}(s) \mid s \in S\}$ in the following. The domain $\mathbb{D}$ for the denotational semantics comprises the power-set of $\perp$ and all $(\simeq_c)_0$-equivalence classes of closed abstractions.

Definition 5.4.5. *The partial order* $(\mathbb{D}, \subseteq)$ *is defined by*

$$\mathbb{D} \stackrel{def}{=} \mathcal{P}\left(\left(\texttt{ans}(\Lambda_{\approx}^0)/(\simeq_c)_0\right) \cup \{\perp\}\right) \tag{5.4.1}$$

We omit the details of proving that $(\mathbb{D}, \subseteq)$ is indeed appropriate as a domain in favour of a more intuitive discussion. E.g., it seems evident that fundamental equations like $[\![\texttt{pick}\,s\,t]\!] = [\![s]\!] \cup [\![t]\!]$ can be satisfied in the model. Furthermore, the domain obviously permits the construction of infinite ascending chains, especially like $(f^i \circledcirc)$ from section 5.2, and thus seems capable of representing contextual least upper bounds.

5.4.2   Denotation

After defining the domain we turn straight to the denotation itself. Basically, a term will be mapped to the set of all $(\simeq_c)_0$-equivalence classes of abstractions that are contextually smaller. Note that $\perp$ is added, otherwise the denotation will not be down-closed. A closed term $r \in \Lambda^0_\approx$ should adhere to the equation

$$\llbracket r \rrbracket = \big\{\ [\lambda x.s]_{\simeq_c} \mid \lambda x.s \in \Lambda^0_\approx\ \wedge\ \lambda x.s \lesssim_c r\ \big\} \cup \{\perp\} \qquad (5.4.2)$$

In order to transfer this to open terms we need a mechanism to derive (sets of) closed terms from open terms equipped with a variable assignment. Every variable assignment ρ corresponds in a natural way to a set of pseudo-valued substitutions. We therefore interpret variable assignments as mapping *terms* to sets of equivalence classes in the following.

Definition 5.4.6. *Let $r \in \Lambda_\approx$ be a term. Then its denotation $\llbracket r \rrbracket\ \rho$ w.r.t some variable assignment ρ is defined as below.*

$$\llbracket r \rrbracket\ \rho \stackrel{def}{=} \big\{\ [\lambda x.s]_{\simeq_c} \mid \lambda x.s \in \Lambda^0_\approx\ \wedge\ \exists [t]_{\simeq_c} \in \rho(r):\ \lambda x.s \lesssim_c t\ \big\} \cup \{\perp\} \quad (5.4.3)$$

This is well-defined. Moreover, by use of the contextual preorder $\lesssim_c$, there is no need to add an extra club-operator or so. Taking the club is "built" into the contextual preorder, so to speak.

Instead of such a direct approach also an inductive definition would have been possible. However, the more concise representation above is preferred, because it gives a better clue of how to establish adequacy and full abstraction.

5.4.3   Adequacy and Full Abstraction

In this section we will prove the denotational semantics to be adequate and fully abstract. The proof will be for closed terms only, since for open terms it is rather technical but seems not very difficult because of the above mentioned correspondence between variable assignments and pseudo-valued substitutions.

Lemma 5.4.7 (Adequacy). *The denotation $\llbracket \cdot \rrbracket$ is adequate, i.e., for all closed terms $s, t \in \Lambda^0_\approx$ the equality $\llbracket s \rrbracket = \llbracket t \rrbracket$ implies $s\ (\simeq_c)_0\ t$.*

Proof. By symmetry it is enough to show $\llbracket s \rrbracket \subseteq \llbracket t \rrbracket \implies s \lesssim_c t$. So $\llbracket s \rrbracket \subseteq \llbracket t \rrbracket$ and a context C such that $C[s] \Downarrow$ is assumed. By the Surface Context Lemma C may be supposed to be a surface context. Then, by lemma 3.2.20, there a two possibilities: Either $C[\circledcirc] \Downarrow$ and nothing has to be shown. Or the evaluation of

s may be performed first, i.e., $C[s] \xrightarrow{\mathcal{S}\,C,\,*}_{\lambda_\approx} C[\lambda z.q] \Downarrow$ holds. As $\lambda z.q \lesssim_c s$ by corollary 5.1.4, its equivalence class is contained in $[\![s]\!]$ by definition. Hence from $[\![s]\!] \subseteq [\![t]\!]$ we also have $[\lambda z.q]_{\simeq_c} \in [\![t]\!]$ which implies that $\lambda z.q \lesssim_c t$ holds. Thus $C[\lambda z.q] \lesssim_c C[t]$ by compatibility in contexts and the claim is shown. $\qquad\square$

Theorem 5.4.8 (Full Abstraction). *The denotation $[\![\cdot]\!]$ is fully abstract: Let $s, t \in \Lambda^0_\approx$ be closed terms. Then $s\ (\simeq_c)_0\ t$ if and only if $[\![s]\!] = [\![t]\!]$ holds.*

Proof. Since the "if"-part is just the adequacy of lemma 5.4.7, we concentrate on the "only-if"-part. By symmetry, $s \lesssim_c t \implies [\![s]\!] \subseteq [\![t]\!]$ suffices. So assume $s \lesssim_c t$ and some closed abstraction $\lambda z.q \in \Lambda^0_\approx$ whose equivalence class is contained in $[\![s]\!]$, i.e., $[\lambda z.q]_{\simeq_c} \in [\![s]\!]$ holds. By definition, $\lambda z.q \lesssim_c s$ and from transitivity $\lambda z.q \lesssim_c s \lesssim_c t$ follows. Thus $[\lambda z.q]_{\simeq_c} \in [\![t]\!]$ is true. $\qquad\square$

5.4.4 Denotational Equations

As mentioned before, the definition of the denotation is not compositional. But this is not really a flaw, since we could prove the appropriate equations. The example $[\![\texttt{pick } s\ t]\!] = [\![s]\!] \cup [\![t]\!]$ was already given. It is listed together with the remaining equations which could be formulated as follows.

The case for an abstraction is omitted, since there is no obvious way to construct a set of terms from a map on the domain.

$$[\![x]\!]\ \rho = \rho(x) \quad \text{if } s \in V \text{ is a variable} \tag{5.4.4}$$

$$[\![\odot]\!]\ \rho = \{\bot\} \tag{5.4.5}$$

$$[\![\texttt{let } x = s \texttt{ in } t]\!]\ \rho = [\![t]\!]\ \rho[([\![s]\!]\ \rho)/x] \tag{5.4.6}$$

$$[\![s\,t]\!]\ \rho = \bigcup \{\ [\![s']\!]\ (\rho[([\![t]\!]\ \rho)/x]) \mid [\lambda x.s']_{\simeq_c} \in [\![s]\!]\ \rho\ \} \cup \{\bot\} \tag{5.4.7}$$

$$[\![\texttt{pick } s\ t]\!]\ \rho = ([\![s]\!]\ \rho)\ \cup\ ([\![t]\!]\ \rho) \tag{5.4.8}$$

However, the equations given above provide a sensible basis for further studies of the denotational semantics. We prove $[\![\texttt{pick } s\ t]\!] = [\![s]\!] \cup [\![t]\!]$ as an example.

Since $s \lesssim_c \texttt{pick } s\ t$ and $t \lesssim_c \texttt{pick } s\ t$ by the reductions $\texttt{pick } s\ t \xrightarrow{\text{nd, left}} s$ and $\texttt{pick } s\ t \xrightarrow{\text{nd, right}} t$ respectively, the inclusion $[\![s]\!] \cup [\![t]\!] \subseteq [\![\texttt{pick } s\ t]\!]$ is trivial. For the converse, suppose $\lambda z.q \lesssim_c \texttt{pick } s\ t$ and show that either $\lambda z.q \lesssim_c s$ or $\lambda z.q \lesssim_c t$ is true. We therefore apply the Surface Context Lemma and assume that $S[\lambda z.q] \Downarrow$ as well as $S[\texttt{pick } s\ t] \Downarrow$ holds. Then by lemma 3.2.20 either $S[\odot] \Downarrow$ for which $S[s] \Downarrow$ and $S[t] \Downarrow$ are clear. Or $\texttt{pick } s\ t \Downarrow \lambda y.p$ such that $S[\lambda y.p] \Downarrow \lambda z.q$

is valid. Hence either $\mathtt{pick}\ s\ t \xrightarrow{\text{nd, left}} s \Downarrow \lambda y.p$ or $\mathtt{pick}\ s\ t \xrightarrow{\text{nd, right}} t \Downarrow \lambda y.p$ and the claim is shown.

Exploiting the algebraic properties of set union it is now straightforward to derive further equations for $\mathtt{pick}$, e.g., commutativity and associativity.

$$[\![\mathtt{pick}\ s\ t]\!]\ \rho = [\![\mathtt{pick}\ t\ s]\!]\ \rho$$
$$[\![\mathtt{pick}\ r\ (\mathtt{pick}\ s\ t)]\!]\ \rho = [\![\mathtt{pick}\ (\mathtt{pick}\ r\ s)\ t]\!]\ \rho$$

5.4.5   Future Work

As we have seen in the case of $\mathtt{pick}$, the strength of the denotational equations from the previous section lies in enabling an algebraic style of reasoning. Hence it would be desirable to prove further of those equations.

Moreover, in the spirit of [Wad78], it might be possible to replace the condition $\lambda x.s \lesssim_c r$ in the definition of the denotation $[\![r]\!]$ by an approximation $\lambda x.s \in \mathtt{ans}(r)$. The key would be to establish a kind of result which is called "limit theorem" in [Dav89, Theorem 5.6.8], namely that the value of a term can be computed as the limit of its approximants.

$$[\![s]\!]\ \rho = \bigsqcup \{\, [\![\lambda x.t]\!]\ \rho \mid \lambda x.t \in \mathtt{ans}(s) \,\}$$

Since, as we have learned in section 5.2, not every least upper bound is continuous w.r.t. contexts, denotation and domain had to be adjusted accordingly.

Chapter 6

Possible Extensions of the Base Calculus

The previous chapters contain a quite complete treatment of a non-deterministic call-by-need lambda calculus. Section 3.1 has defined the small-step reduction semantics of the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus and with the approximation calculus $\boldsymbol{\lambda}_{\approx}$ of section 3.2 an alternative representation has been developed in which the outcomes of an evaluation do not have to carry around a `let`-environment.

Founding on the equivalence of these calculi from section 3.3, a sensible notion of similarity has been devised in chapter 4. There, using the lcs-framework of chapter 2, open similarity was shown to be a precongruence, enabling its use for proving correctness of program transformations. Furthermore, chapter 5 has demonstrated that the contextual preorder possesses a rich theory from which also a fully abstract denotational semantics was derived.

It has been pointed out before that the calculus $\boldsymbol{\lambda}_{\mathrm{ND}}$ was intentionally designed to be powerful while keeping the presentation of the rather complex material as simple as possible. Since it shall serve as a starting point, it is rather basic in at least two respects. First, it does not cover data types, i.e., constructors and a `case` common in modern functional programming languages like, e.g., Haskell [PJ03] and ML [MTH90]. Secondly, at least the aforementioned languages support (mutual) recursive bindings in their `let`-construct. The `let`-operator in the calculus $\boldsymbol{\lambda}_{\mathrm{ND}}$ is non-recursive and recursion has therefore to be expressed using fixed point combinators, e.g., the **Y**-combinator.

However, it is well-known, cf. [Lau93], that this causes a different behaviour

in the execution of functional programs. Hence the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus is not capable to reflect this behaviour accurately. Albeit not measuring space or time usage, an example in section 6.2 will make this visible exploiting non-determinism. There, we also discuss some of the representation issues. Since a `letrec` binds more than just one variable, it cannot be encoded directly as an operator in a lazy computation language.

Fortunately this is not a problem for constructors and `case` which will be treated in section 6.1. Before we describe the manner in which the base calculus could be extended, we briefly summarise the most relevant parts of the proof that open similarity is a precongruence. One of the crucial notions was the extension of relations to open terms, or more precisely, how it could be defined the right way for a calculus with sharing. The fact that the corresponding parts of the proof nicely fit together, demonstrates that the admissibility of the open extension is essential. Indeed it is based on the equivalence of closing `let`-environments and pseudo-valued substitutions, cf. proposition 4.1.27, and the Substitution Lemmas w.r.t. the precongruence candidate for just the pseudo-values. This suggests the following "recipe" for a `let`-calculus: First determine which terms may be copied – enriched by ⊙ these form the pseudo-values. Then, one should be able to show that pseudo-valued substitutions have the same power as arbitrary closing `let`-environments for distinguishing open terms. Moreover, for the precongruence candidate the corresponding substitution lemmas have to be established.

6.1   Lambda Calculi with `case` and Constructors

When introducing data types into a lambda calculus, some topics have to be clarified. First, in contrast to e.g. [Pit97, Gor99] we will not discuss a typed language. This means, though data types and a `case`-construct are provided, not every term will have to be well-typed. So we emphasise the intent of such a calculus as an intermediate product in the compilation of a high-level language: We assume that the program has already been type-checked, as e.g. for Haskell, and simply regard terms with type errors as non-converging. The reason is not only to simplify the presentation but also to ensure that no contextual equivalences would be excluded, solely because the type system was too restricted.

Furthermore, like in algebraic reasoning, there is some choice whether the data constructors should be partitioned into disjoint sets, types or sorts so to speak, where every type A possesses a distinct `case`$_{\mathtt{A}}$-construct. Some work, cf. [Sch00, SSSS04], pursues this view while other calculi, e.g. [SS03a], provide

only a single `case`-construct which all data constructors belong to. In the following, we will give multiple sorts priority over the approach with a single `case`, since the latter can always be implemented within the former.

Note that the converse is in general not true, i.e., just dropping the type information from the different `case`-constructs will make certain contextual equivalences false. The example below, inspired by [SSSS04], illustrates this.

Example 6.1.1. *Assume constructors* $c_{A,1}, c_{A,2} \in A$ *and* $c_B \in B$ *belonging to the two data types* A, B *respectively. Then the terms*

$$s \equiv \lambda f.\mathtt{case_A}\ (f\ c_{A,1})\ \mathtt{of}\ \ c_{A,1}\ \mathtt{->}\ c_{A,2}$$
$$c_{A,2}\ \mathtt{->}\ \mathtt{case_A}\ (f\ c_B)\ \mathtt{of}\ \ c_{A,1}\ \mathtt{->}\ c_{A,2}$$
$$c_{A,2}\ \mathtt{->}\ \mathbf{\Omega}$$

and

$$t \equiv \lambda f.\mathtt{case_A}\ (f\ c_{A,1})\ \mathtt{of}\ \ c_{A,1}\ \mathtt{->}\ \mathbf{\Omega}$$
$$c_{A,2}\ \mathtt{->}\ \mathtt{case_A}\ (f\ c_B)\ \mathtt{of}\ \ c_{A,1}\ \mathtt{->}\ \mathbf{\Omega}$$
$$c_{A,2}\ \mathtt{->}\ c_{A,2}$$

are contextually equivalent because every possible f *has to use either* $\mathtt{case}_A$ *or* $\mathtt{case}_B$ *to examine its function argument and thus runs into a type error.*

Dropping the distinction between the types A *and* B *using only a single* `case`-*construct for* $A \cup B$, *the terms* s *and* t *are obviously be distinguished by the context* $C \equiv ([\ ]\ (\lambda x.\mathtt{case_{A \cup B}}\ x\ \mathtt{of}\ (c_{A,1}\ \mathtt{->}\ c_{A,1})\ (c_{A,2}\ \mathtt{->}\ c_{A,1})\ (c_B\ \mathtt{->}\ c_{A,1}))).$

Another design issue is the existence of a strictness operator. In a calculus without constructors, adding such an operator has no impact on the contextual equalities, whereas the following example taken from [SS03b] shows that with data constructors and `case` the situation becomes different.

Example 6.1.2. *As in example 6.1.1, let again* A *be a type with data constructors* $c_{A,1}$ *and* $c_{A,2}$. *Now consider the terms* $s \equiv D[\mathbf{\Omega}]$ *and* $t \equiv D[\lambda x.\mathbf{\Omega}]$ *given by the context* $D \equiv \lambda f.\mathtt{case_A}\ (f\ [\ ])\ \mathtt{of}\ (c_{A,1}\ \mathtt{->}\ c_{A,1})\ (c_{A,2}\ \mathtt{->}\ c_{A,1})$. *Assuming sequential evaluation by the operator* `seq` *obeys the equalities*

$$s\ \mathtt{seq}\ t\ \simeq_c \begin{cases} \mathbf{\Omega} & \textit{if } s \simeq_c \mathbf{\Omega} \\ t & \textit{otherwise} \end{cases}$$

the terms s *and* t *will be distinguished by the context* $C \equiv [\ ]\ (\lambda y.(y\ \mathtt{seq}\ c_{A,1})).$

Intuitively, the reason for the failure of contextual equivalence lies in "hiding" the term $\mathbf{\Omega}$ and $\lambda x.\mathbf{\Omega}$ respectively in the first argument of a `case`. As $\lambda x.\mathbf{\Omega}$

is a functional expression, choosing $f \equiv \mathbf{I}$ would clearly produce a type error[1]. Hence `seq` is the only way to obtain $c_{A,1}$ or $\mathbf{\Omega}$ depending on the term under the `case`. One could say that `seq` acts like a `case` but for "function types". That's why `case` in FUNDIO [SS03a] has a an extra pattern for abstractions.

6.1.1   A λ-`let`-calculus with `case`, Constructors and `pick`

After we have shed some light on design issues, we will propose a concrete non-deterministic call-by-need lambda calculus with `case` and constructors. Figure 6.1 describes its syntax where `c` stands for arbitrary data constructors. Furthermore, we write $|A|$ for the cardinality of a data type, i.e. the number of data constructors associated with it while $ar(c)$ denotes the arity of a constructor, i.e. the number of its arguments. The expression $(c\ E_1 \ldots E_{ar(c)})$ in the gram-

$$
\begin{aligned}
E ::= {}& V \ \mid\ (\lambda V.E) \ \mid\ (E\ E) \ \mid\ (\texttt{let}\ V = E\ \texttt{in}\ E) \\
& \mid\ (\texttt{pick}\ E\ E) \ \mid\ (E\ \texttt{seq}\ E) \\
& \mid\ (\texttt{c}\ E_1 \ldots E_{ar(\texttt{c})}) \\
& \mid\ (\texttt{case}_{\texttt{A}}\ E\ \texttt{of}\ P_1 \ldots P_{|A|})
\end{aligned}
$$

$$
P ::= (\texttt{c}\ V_1 \ldots V_{ar(\texttt{c})})\ \texttt{->}\ E
$$

Figure 6.1: Syntax for expressions in the language $\Lambda_{\mathrm{ND+con}}$

mar of figure 3.1 indicates that constructor applications are always saturated. Analogously for the `case`-construct, i.e., $\texttt{case}_{\texttt{A}}\ E\ \texttt{of}\ P_1 \ldots P_{|A|}$ means that every `case` is equipped with a complete list of all patterns which are possible for its type. We therefore enumerate the constructors of A by $c_{A,1}, \ldots, c_{A,|A|}$.

Since data constructors do not bind variables, the arity of a constructor directly corresponds to its arity in the representation of a lazy computation language. Also the `case`-construct perfectly fits in this framework, all except the first argument binding as many variables as the corresponding constructor

---

[1]So the terms $D[\mathbf{\Omega}]$ and $D[\lambda x.\mathbf{\Omega}]$ can already be distinguished without using `seq`, when type errors and $\mathbf{\Omega}$ are *not* identified. However, this will not be discussed here.

has arguments. So the arities for the new language operators are given below.

$$\alpha(\texttt{c}) \stackrel{\text{def}}{=} \langle \underbrace{0, \ldots, 0}_{ar(\texttt{c})} \rangle$$

$$\alpha(\texttt{case}_\texttt{A}) \stackrel{\text{def}}{=} \langle 0, ar(\texttt{c}_{\texttt{A},1}), \ldots, ar(\texttt{c}_{\texttt{A},|\texttt{A}|}) \rangle$$

We will frequently have to deal with large nested let-environment and therefore introduce the following notation.

Notation. *Let I be an index set. We write $\{\!\!\{$ let $x_i = t_i$ in s $\}\!\!\}$ to designate the term* let $x_1 = t_1$ in (let $x_2 = t_2$ in ... (let $x_n = t_n$ in s)). *For an empty index set I the term s is denoted.*

Although it might seem that an interpreter can provide more information for a non-well-typed term than for a term whose reduction does not terminate, those terms will not be distinguished. The reason is simply that those terms do not converge and that the contextual equivalence we have in mind will only consider convergence. It was mentioned before that, apart from this, we understand our calculus as a core language for functional programs which have already been type-checked.

Since it represents a call-by-need calculus, we have to explain when or which kind of constructor terms may be copied. Certainly there is also an interdependence w.r.t. the style of the case-rules. Usually case just decomposes its first argument and chooses the right pattern. So for a case-expression of a constructor application one would expect the rule below.

case$_\texttt{A}$ ($\texttt{c}_{\texttt{A},\texttt{i}}$ $t_1 \ldots t_n$) of ... ($\texttt{c}_{\texttt{A},\texttt{i}}$ x_{i_1} ... x_{i_n}) -> s_i ...

$$\xrightarrow{\text{case-c}} \{\!\!\{ \text{ let } z_j = t_j \text{ in } s_i[z_j/x_{i_j}] \}\!\!\} \quad (\text{case-c})$$

Note that the let-bindings are necessary to preserve sharing. However, what if case is applied to a variable which is bound to a constructor expression in a surrounding let-environment? We will not discuss the issue at full length here, but notice that unrestricted copying of full constructor terms breaks sharing. This somewhat relates to the fact that not every constructor term may be considered "cheap" in the sense of [GHC03]. So there are chiefly two possibilities:

- To make the rule (case-c) applicable, at least the constructor must be known to the case-expression. Therefore, constructor applications of variables may be copied. To obtain such a constructor term the arguments

of a general constructor application may be "abstracted". This works as follows: A constructor term of the form $(c\ t_1 \ldots t_{ar(c)})$ may be replaced by $\{\!\!\{\ \texttt{let}\ x_i = t_i\ \texttt{in}\ (c\ x_1 \ldots x_{ar(c)})\ \}\!\!\}$ where $x_i = t_i$ represent the appropriate bindings for the arguments of the original constructor application. In this way, we obtain two rules, one for the abstraction operation and the other for copying constructor applications of variables.

Note that albeit the abstraction operation appears a bit unusual, it has a counterpart w.r.t. function application: E.g. [MSC99a] consider a language with a restricted syntax where only variables are allowed in the argument positions of functions.

- The two steps which have previously been described may also be performed at once. Thus obtaining a rule which decomposes the constructor term bound to the variable to be cased. But since these two steps could generally happen at different locations, the formulation of such a reduction rule may be tedious unless `letrec` is used.

  The reason is that in order to find the binding to the constructor term several indirections might have to be followed. Whereas in languages with `letrec` usually a normalisation takes place in that only one large environment has to be considered. This makes lookup for bindings easier.

The first approach consists of introducing two rules, in particular the constructor abstraction, for which reduction diagrams have to be constructed potentially. So we prefer the second approach while simplifying the presentation by resolving indirections in advance: Since corollary 4.1.4 has established the correctness of copying variables, it is sensible to adopt this as a reduction rule. Of course, a rule to shift `case` through `let`-expressions has to be provided, too. Figure 6.2 shows the reduction rules which make up the $\boldsymbol{\lambda}_{\mathrm{ND+con}}$-calculus. We briefly sketch how the normal-order and reduction context would be affected:

- The rule (cp) is applied in the same context as before, but additionally if the term the variable is bound to is another variable. Compared to $\boldsymbol{\lambda}_{\mathrm{ND}}$ this changes the normal-order reduction also for terms of the form

$$\texttt{let}\ x = \lambda z.q\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ R[x])$$

$$\text{let } x = (\text{let } y = t_y \text{ in } t_x) \text{ in } s \xrightarrow{\text{llet}} \text{let } y = t_y \text{ in } (\text{let } x = t_x \text{ in } s) \quad \text{(llet)}$$

$$(\text{let } x = t_x \text{ in } s) \, t \xrightarrow{\text{lapp}} \text{let } x = t_x \text{ in } (s \, t) \quad \text{(lapp)}$$

$$(\lambda x.s) \, t \xrightarrow{\text{lbeta}} \text{let } x = t \text{ in } s \quad \text{(lbeta)}$$

$$(\text{let } x = r \text{ in } s) \text{ seq } t \xrightarrow{\text{lseq}} \text{let } x = r \text{ in } (s \text{ seq } t) \quad \text{(lseq)}$$

$$(\lambda x.s) \text{ seq } t \xrightarrow{\text{eseq}} t \quad \text{(eseq)}$$

$$\text{pick } s \, t \xrightarrow{\text{nd, left}} s \quad \text{(nd, left)}$$

$$\text{pick } s \, t \xrightarrow{\text{nd, right}} t \quad \text{(nd, right)}$$

$$\text{let } x = r \text{ in } D[x] \xrightarrow{\text{cp}} \text{let } x = r \text{ in } D[r]$$
$$\text{where } r \text{ is a variable or an abstraction} \quad \text{(cp)}$$

$$\text{case } (\text{let } x = s \text{ in } t) \text{ of } a \xrightarrow{\text{lcase}} \text{let } x = s \text{ in } (\text{case } t \text{ of } a) \quad \text{(lcase)}$$

$$\text{case}_{\texttt{A}} \, (\texttt{c}_{\texttt{A,i}} \, t_1 \ldots t_n) \text{ of } \ldots \, (\texttt{c}_{\texttt{A,i}} \, x_{i_1} \, \ldots \, x_{i_n}) \text{ -> } s_i \, \ldots$$
$$\xrightarrow{\text{case-c}} \{\!\{ \text{ let } z_j = t_j \text{ in } s_i[z_j/x_{i_j}] \, \}\!\} \quad \text{where } z_j \text{ are fresh} \quad \text{(case-c)}$$

$$\text{let } x = (\texttt{c}_{\texttt{A,i}} \, t_1 \ldots t_n) \text{ in } D[\text{case}_{\texttt{A}} \, x \text{ of } \ldots \, (\texttt{c}_{\texttt{A,i}} \, x_{i_1} \, \ldots \, x_{i_n}) \text{ -> } s_i \, \ldots]$$
$$\xrightarrow{\text{case-in}} \{\!\{ \text{ let } z_j = t_j \text{ in } (\text{let } x = (\texttt{c}_{\texttt{i}} \, z_1, \ldots, z_n) \text{ in } D[s_i[z_j/x_{i_j}]]) \, \}\!\}$$
$$\text{(case-in)}$$

Figure 6.2: The reduction rules of the $\boldsymbol{\lambda}_{\text{ND}+\texttt{con}}$-calculus

which in the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus reduces normal-order like

$$\texttt{let } x = \lambda z.q \texttt{ in } (\texttt{let } y = x \texttt{ in } R[x])$$
$$\xrightarrow{n,\ \mathrm{cp}} \texttt{let } x = \lambda z.q \texttt{ in } (\texttt{let } y = \lambda z.q \texttt{ in } R[x])$$
$$\xrightarrow{n,\ \mathrm{cp}} \texttt{let } x = \lambda z.q \texttt{ in } (\texttt{let } y = \lambda z.q \texttt{ in } R[\lambda z.q])$$

but involves less copies of λ-terms in the $\boldsymbol{\lambda}_{\mathrm{ND+con}}$-calculus:

$$\texttt{let } x = \lambda z.q \texttt{ in } (\texttt{let } y = x \texttt{ in } R[x])$$
$$\xrightarrow{n,\ \mathrm{cp}} \texttt{let } x = \lambda z.q \texttt{ in } (\texttt{let } y = x \texttt{ in } R[y])$$
$$\xrightarrow{n,\ \mathrm{cp}} \texttt{let } x = \lambda z.q \texttt{ in } (\texttt{let } y = x \texttt{ in } R[\lambda z.q])$$

- reduction contexts now also exhibit `case`-expressions, i.e.

$$X ::= \texttt{case}_{\texttt{A}} [\,] \texttt{ of } \dots$$
$$Z ::= A_L \ \mid \ X$$
$$R ::= L_R^*[Z^*] \ \mid \ L_R^*[\texttt{let } x = Z^* \texttt{ in } R[x]]$$

The Approximation Calculus

It is clear that in the $\boldsymbol{\lambda}_{\mathrm{ND+con}}$-calculus surface contexts also extend over constructor terms, i.e., if the rule for constructor abstraction was used the arguments of constructors would be located in `let`-bindings.

So approximation reductions have to be performed inside all these contexts. Furthermore, in the $\boldsymbol{\lambda}_{\mathrm{ND+con}}$-calculus, abstractions, variables and, implicitly in rule (case-in), constructor applications to variables are copied by the normal-order reduction. Hence it seems possible to show that it is correct to copy variables, abstractions or constructor terms built from these terms, i.e. terms according to the grammar below.

$$B ::= V \ \mid \ \lambda x.T \ \mid \ (\texttt{c } B_1 \dots B_n)$$

Therefore the corresponding (cpa)-rule in the approximation variant of the calculus $\boldsymbol{\lambda}_{\mathrm{ND+con}}$ should be defined to copy these terms. It is worth mentioning that this corresponds to the notion of a "cheap" term in [GHC03].

6.2 Lambda Calculi with recursive `let`

It was mentioned before that the `let`-operator in the calculus $\boldsymbol{\lambda}_{\mathrm{ND}}$ is non-recursive. Thus in $\boldsymbol{\lambda}_{\mathrm{ND}}$, recursion must be represented by a fixed point combinator, e.g. **Y**. In functional programming this would be tedious and is usually avoided by permitting recursive bindings via `letrec`, cf. [PJ03, MTH90].

However, it is well-known, that by means of recursive bindings more sharing can be obtained in contrast to the use of fixed point combinators. In [Lau93], Launchbury gives the term `letrec` $t = ($if e then `Nil` else `Cons` $1\ t)$ in t as an example for this. Here, the evaluation of e is only performed once and shared between all recursive calls to t. If **Y** $(\lambda t.$if e then `Nil` else `Cons` $1\ t)$ was used as an encoding in the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus, the term e would be evaluated repeatedly as the abstraction $\lambda t.\ \dots$ is copied during reduction.

Since we do not measure space or time usage but only consider convergence, the above terms would be contextually equivalent. Though, non-determinism provides a way to discover that a straightforward translation of `letrec` into the calculus $\boldsymbol{\lambda}_{\mathrm{ND}}$ is not possible. In addition, the following example does *not* involve data types and thus demonstrates that the issue already comes up in basic calculi without `case` or constructors.

Example 6.2.1. *Consider the term* `letrec` $t = $ `pick` **K2** $(\lambda g.g\,t)$ in t *when successively applied to the arguments* $\mathbf{I} \equiv \lambda x.x$ *and* $\boldsymbol{\Omega}$*:*

$$(\texttt{letrec}\ t = \texttt{pick}\ \mathbf{K2}\ (\lambda g.g\,t)\ \texttt{in}\ t)\,\mathbf{I}\,\boldsymbol{\Omega}$$

$$\xrightarrow{\mathit{lapp}} \cdot \xrightarrow{\mathit{lapp}} \cdot \xrightarrow{\mathit{nd,\ left}} \texttt{letrec}\ t = \mathbf{K2}\ \texttt{in}\ t\,\mathbf{I}\,\boldsymbol{\Omega}$$

$$\xrightarrow{\mathit{cp}} \texttt{letrec}\ t = \mathbf{K2}\ \texttt{in}\ \mathbf{K2}\,\mathbf{I}\,\boldsymbol{\Omega} \quad \simeq_c \quad \boldsymbol{\Omega}$$

$$(\texttt{letrec}\ t = \texttt{pick}\ \mathbf{K2}\ (\lambda g.g\,t)\ \texttt{in}\ t)\,\mathbf{I}\,\boldsymbol{\Omega}$$

$$\xrightarrow{\mathit{lapp}} \cdot \xrightarrow{\mathit{lapp}} \cdot \xrightarrow{\mathit{nd,\ right}} \texttt{letrec}\ t = (\lambda g.g\,t)\ \texttt{in}\ t\,\mathbf{I}\,\boldsymbol{\Omega}$$

$$\xrightarrow{\mathit{cp}} \cdot \xrightarrow{\mathit{lbeta}} \texttt{letrec}\ t = (\lambda g.g\,t)\ \texttt{in}\ (\texttt{letrec}\ g = \mathbf{I}\ \texttt{in}\ g\,t)\,\boldsymbol{\Omega}$$

$$\xrightarrow{\mathit{lapp}} \cdot \xrightarrow{\mathit{llet}} \cdot \xrightarrow{\mathit{cp}} \texttt{letrec}\ t = (\lambda g.g\,t),\ g = \mathbf{I}\ \texttt{in}\ \mathbf{I}\,t\,\boldsymbol{\Omega}$$

$$\xrightarrow{\mathit{lbeta}} \cdot \xrightarrow{\mathit{cp}} \cdot \xrightarrow{\mathit{lapp}} \cdot \xrightarrow{\mathit{llet}} \texttt{letrec}\ t = (\lambda g.g\,t),\ g = \mathbf{I},\ x = t\ \texttt{in}\ x\,\boldsymbol{\Omega}$$

$$\xrightarrow{\mathit{cp}} \cdot \xrightarrow{\mathit{cp}} \texttt{letrec}\ t = (\lambda g.g\,t),\ g = \mathbf{I},\ x = (\lambda g.g\,t)\ \texttt{in}\ (\lambda g.g\,t)\,\boldsymbol{\Omega}$$

Thus `letrec` $t = $ `pick` **K2** $(\lambda g.g\,t)$ in t *does not converge in the context* $[\ ]\,\mathbf{I}\,\boldsymbol{\Omega}$*, since both alternatives yield terms where* $\boldsymbol{\Omega}$ *is the normal-order redex.*

Now regard the naive translation $s \equiv \mathbf{Y}\,(\lambda t.\mathtt{pick}\,\mathbf{K2}\,(\lambda g.g\,t))$ using the $\mathbf{Y}$-combinator. Plugged into the context $[\,]\,\mathbf{I}\,\mathbf{\Omega}$, it may converge as follows:

$$\mathbf{Y}\,(\lambda t.\mathtt{pick}\,\mathbf{K2}\,(\lambda g.g\,t))\,\mathbf{I}\,\mathbf{\Omega}$$

$$\xrightarrow{\text{lbeta}}\cdot\xrightarrow{\text{cpa}} (\lambda x.(\lambda t.\mathtt{pick}\,\mathbf{K2}\,(\lambda g.g\,t))\,(x\,x))\,(\lambda y.\lambda t.\dots(y\,y))\,\mathbf{I}\,\mathbf{\Omega}$$

$$\xrightarrow{\text{lbeta}}\cdot\xrightarrow{\text{cpa}} (\lambda t.\mathtt{pick}\,\mathbf{K2}\,(\lambda g.g\,t))\,((\lambda y.\dots)\,(\lambda y'.\dots))\,\mathbf{I}\,\mathbf{\Omega}$$

$$\xrightarrow{\text{lbeta}}\cdot\xrightarrow{\text{nd, right}} \mathtt{let}\ t = ((\lambda y.\dots)\,(\lambda y'.\dots))\ \mathtt{in}\ (\lambda g.g\,t)\,\mathbf{I}\,\mathbf{\Omega}$$

Starting from here, after a few further reduction steps, a term which is contextually equivalent to $\mathtt{let}\ t = ((\lambda y.\dots)\,(\lambda y'.\dots))\ \mathtt{in}\ t\,\mathbf{\Omega}$ can be produced. Since the binding for t still has the potential of a non-deterministic choice, we may obtain a term like $\mathbf{K2}\,\mathbf{\Omega}$ which obviously converges.

Hence, this example illustrates that the original `letrec`-term is contextually smaller than its naive translation. However, this should not be misunderstood: It does in no way represent a deficiency of the `letrec`-calculus, since also the corresponding $\mathbf{Y}$-term can be represented. Therefore, rather the converse is true, i.e., the `letrec`-calculus is more expressive and the example points out that its contextual equivalence possesses a finer structure than in a calculus where `let` is non-recursive.

6.2.1 Representation in a Lazy Computation Language

So we have seen the motivation for a calculus with recursive bindings but we have not yet discussed how `letrec` could be represented in the framework of lazy computation languages.

As is known, the non-recursive `let` $x = s$ `in` t represented as $\mathtt{let}(x.t, s)$ has arity $\langle 1, 0\rangle$ which means that its first operand binds just a single variable. But if the binding was recursive, the variable x would be bound in s too.

Obviously, a construct like $\mathtt{let}(x.s, x.t)$ appears *not* to be an appropriate representation. Apart from this, the issue of mutual recursive bindings as, e.g., in `letrec` $x = C[x, y],\ y = D[x, y]$ `in` t has to be solved. However, this would be possible if a countably infinite set of `letrec`-operators, i.e., for each number of bindings one with the corresponding arity, was used instead of a single one.

6.2.2 Approximating Recursive Bindings

Because in `letrec` $x = s$ `in` t the term s may contain an occurrence of the variable x, the binding cannot be deleted even if s is an abstraction. This means,

the rule (cpa) in its original form is not applicable to a `letrec`-calculus. It might be conceivable to add an alternative for (cpa) which keeps the environment, thus obtaining three variants of the rule as below.

$$\texttt{let } x = \odot \texttt{ in } t \xrightarrow{\text{cpa}-\odot} t[\odot/x] \tag{cpa-$\odot$}$$

$$\texttt{let } x = \lambda y.s \texttt{ in } t \xrightarrow{\text{cpa}-\lambda} t[\lambda y.s/x] \quad \text{if } x \notin \mathcal{FV}(s) \tag{cpa-λ}$$

$$\texttt{let } x = \lambda y.s \texttt{ in } t \xrightarrow{\text{cpa}-\mu} \texttt{let } x = \lambda y.s \texttt{ in } t[\lambda y.s/x] \tag{cpa-μ}$$

If we had done so, we would just run into the problem of loosing sharing as explained in the foregoing section by means of example 6.2.1. This demonstrates that the present technique of the approximation calculus, i.e., using (cpa) to eliminate `let`-bindings, is no longer feasible.

On the other hand, we have already learned by example 3.1.18 that reduction to weak head normal forms followed by application to arguments does *not* possess the same power in distinguishing terms as contextual equivalence. So in order to develop a suitable similarity for a `letrec`-calculus an appropriate way to represent terms by sets of approximants is still necessary.

For their `letrec`-calculus with `case` and constructors, Schmidt-Schauß et al. propose in [SSSS04] to keep the environments and successively reduce within them. That is, there exists an analogous rule to (stop) and evaluation stops when terms have reached a certain form. This process is continued recursively, creating some kind of "stratified" `letrec`-bindings in this manner.

However, it seems that in a `letrec`-calculus even a method like this cannot eliminate the necessity for the (llet)-rule. Recall that proving the precongruence candidate stable under just this rule seemed infeasible. Hence it is unknown whether the rule (llet) becomes more tractable in a `letrec`-calculus because of its simpler structure there. Moreover, the approach in [SSSS04] introduces several special rules and reduction steps, e.g., garbage collection and normalisation of `letrec`-environments. It would be probably easier to introduce only (stop) as a new rule and permit reductions within surface contexts like in this work.

Chapter 7

Conclusion and Future Work

The subject of this work was similarity in a non-deterministic call-by-need lambda calculus and how to prove it a precongruence. Building upon the seminal work of Howe [How89, How96], chapter 2 has set out the framework of lazy computation systems for proving similarity a precongruence. For this purpose, the original work had to be extended to reflect sharing properly. This chiefly concerns the issue how a relation on closed terms should be extended to open terms. We have therefore introduced the admissibility of an open extension in order to decouple the precongruence proof from a certain view to open terms. This has turned out to be the vital notion for the proof of theorem 2.2.13 which has established substantial conditions for determining when a relation is a precongruence. This result is achieved by means of the precongruence candidate relation and corresponds to [How96, Theorem 3.1]. Thus we have proven in chapter 2 that it is possible to generalise the already highly abstract lazy computation system framework insofar that it is not necessary to fix the open extension in advance.

Of course, admissibility of the open extension has to be established for every concrete calculus in order to make theorem 2.2.13 applicable. To lay the foundation for this, chapter 3 has presented a thorough treatment of two non-deterministic call-by-need lambda calculi. In section 3.1, the operational semantics of the λ_{ND}-calculus has been defined in terms of small-step reduction and contextual equivalence. One key result is the Context Lemma, which states

167

that in order to determine contextual equivalence, observing the termination of evaluation within reduction contexts is sufficient. Whereas from example 3.1.18 we have learned that a sensible definition of similarity in the calculus $\boldsymbol{\lambda}_{\text{ND}}$ is impossible since application to arguments is not enough to distinguish weak head normal forms. Thus, an alternative representation had to be developed: The approximation calculus $\boldsymbol{\lambda}_{\approx}$ of section 3.2 in which `let`-environments are eliminated from weak head normal forms. The Approximation Theorem in section 3.3 is one of the main results. It not only has established the equivalence of the calculi $\boldsymbol{\lambda}_{\text{ND}}$ and $\boldsymbol{\lambda}_{\approx}$ but has also pointed out how terms may be represented by sets of abstractions.

The Approximation Theorem was the essential precondition for basing the development of similarity on the $\boldsymbol{\lambda}_{\approx}$-calculus. Thus proving open similarity a precongruence, one of the main contributions of this work, has demonstrated that the approximation calculus $\boldsymbol{\lambda}_{\approx}$ is in fact sensible. The proof was carried out in chapter 4 and benefitted greatly from the results on the rearrangement of reduction sequences in section 3.2.5. Moreover, it should not be forgotten how the enhanced lcs-framework of chapter 2 has guided the proof. In particular, we have seen that the open extension in a call-by-need calculus must *not* be defined using *all* closing substitutions. Rather pseudo-values only may be substituted, i.e., those terms that may be copied anyway. Proposition 4.1.27 has shown how this corresponds nicely to the use of all closing `let`-environments and to the style of Abramsky's applicative bisimulation. Admissibility of the open extension has been obtained together with Substitution Lemmas for the pseudo-values. We thus have demonstrated admissibility to be a viable notion. Furthermore, the Substitution Lemmas were of significance for proving the precongruence candidate stable under (cpa)-reductions. In section 4.3.1 stability of the precongruence candidate under reduction was treated. It formed the basis for the proof of the Precongruence Theorem.

In order to establish mutual open similarity as a proof method for contextual equivalence, the Main Theorem in section 5.1.2 has combined two of the major results, namely the Precongruence Theorem and the Approximation Theorem. Apart from this we have learned in chapter 5 that open similarity fails to match contextual equivalence completely, since it does not support syntactic continuity. Moreover, section 5.4 has demonstrated the usefulness of the approximation calculus $\boldsymbol{\lambda}_{\approx}$ in defining a fully abstract denotational semantics.

However, in order to really turn the denotational semantics into a handy tool further steps have to be taken. At first more denotational equations have to be established. Secondly, the results from the $\boldsymbol{\lambda}_{\approx}$-calculus could probably be used to represent a term as the limit of a set of abstractions in the spirit of

Wadsworth's "approximate normal forms" [Wad78].

Some further topics for future research have already been mentioned. E.g., comprising an (automatable) method for equality analysis as discussed in section 4.4.2. The extensions of the base calculus described in chapter 6 seem even more desirable. On the basis of the explanations in section 6.1.1 we feel confident that the method presented in this work is powerful enough to treat a calculus with `case` and constructors. Moreover, implementing recursive bindings natively in terms of a `letrec`-construct is of importance, as the discussion in section 6.2 has pointed out that the naive translation of `letrec` via the fixed point combinator $\mathbf{Y}$ is inappropriate. However, neither the representation as a lazy computation language nor the design of the respective approximation calculus seem a straightforward task.

What has gained only little attention up to now is the treatment of infinite reduction sequences in the contextual semantics. In this respect it seems quite promising to define a similarity relation for divergence, or must convergence respectively, and to show its inclusion in the corresponding contextual preorder separately. The "upper similarity" in [Las98b] represents this kind of approach.

Bibliography

[Abr90]   Samson Abramsky. The lazy lambda calculus. In David A. Turner, editor, *Research Topics in Functional Programming*, University of Texas at Austin Year of Programming Series, chapter 4, pages 65–116. Addison-Wesley, 1990.

[ACCL91]   Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1991.

[AF97]   Zena M. Ariola and Matthias Felleisen. The call-by-need lambda calculus. *Journal of Functional Programming*, 7(3):265–301, 1997.

[AFM+95]   Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 233–246. ACM Press, January 1995.

[And02]   Peter B. Andrews. *Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*, volume 27 of *Applied Logic Series*. Kluwer Academic Publishers, Dordrecht, second edition, 2002.

[Aug84]   Lennart Augustsson. A compiler for lazy ML. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 218–227, New York, NY, USA, 1984. ACM Press.

[Bac78]   John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[Bar84]   Hendrik Pieter Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. Elsevier Science Publishers, 1984.

[Bar91]   H. P. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, April 1991.

[Ber98]   Karen L. Bernstein. A congruence theorem for structured operational semantics of higher-order languages. In *Logic in Computer Science*, pages 153–164, 1998.

[BKvO98]   Marc Bezem, Jan Willem Klop, and Vincent van Oostrom. Diagram techniques for confluence. *Information and Computation*, 141(2):172–204, March 1998.

[BN98]   Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[Bou94]   Gérard Boudol. Lambda-calculi for (strict) parallel functions. *Information and Computation*, 108(1):51–127, January 1994.

[BS02]   Manfred Broy and Johannes Siedersleben. Objektorientierte Programmierung und Softwareentwicklung: Eine kritische Einschätzung. *Informatik Spektrum*, 25(1):3–11, February 2002.

[CH88]   Thierry Coquand and Gerard Huet. The calculus of constructions. *Information and Computation*, 76(2-3):95–120, February/March 1988.

[Chu36]   Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936. Reprinted in [Dav04].

[Chu41]   Alonzo Church. The calculi of lambda conversion. *Annals of Mathematics Studies*, 6:1–37, 1941. Princeton University Press.

[Dav89]   Ruth E. Davis. *Truth, Deduction, And Computation*. Computer Science Press, 1989.

[Dav04]   Martin Davis, editor. *The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions*. Dover Publications, February 2004.

[DP92]   B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, 1992.

[dP95]   Ugo de'Liguoro and Adolfo Piperno. Nondeterministic extensions of untyped λ-calculus. *Information and Computation*, 122(2):149–177, November 1995.

[DP01]   Nachum Dershowitz and David A. Plaisted. Rewriting. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 1. Elsevier Science, 2001.

[FH88]   A.J. Field and P.G. Harrison. *Functional Programming*. Addison Wesley, 1988.

[GHC03]   The GHC Team. *The Glasgow Haskell Compiler User's Guide, Version 5.04*, 2003. http://haskell.cs.yale.edu/ghc/docs/5.04.3.

[Gor94a]   Andrew D. Gordon. *Functional programming and input/output*. Distinguished Dissertations in Computer Science. Cambridge University Press, September 1994.

[Gor94b]   Andrew D. Gordon. A tutorial on co-induction and functional programming. In *Functional Programming, Glasgow 1994*, Workshops in Computing, pages 78–95. Springer-Verlag, 1994.

[Gor99]   Andrew D. Gordon. Bisimilarity as a theory of functional programming. *Theoretical Computer Science*, 228(1-2):5–47, October 1999.

[GP98]   Andrew D. Gordon and Andrew M. Pitts, editors. *Higher Order Operational Techniques in Semantics*. Publications of the Newton Institute. Cambridge University Press, 1998.

[GV92]   Jan Friso Groote and Frits Vaandrager. Structured operational semantics and bisimulation as a congruence. *Information and Computation*, 100(2):202–260, October 1992.

[Hen90]   Matthew Hennessy. *The Semantics of Programming Languages: An Elementary Introduction using Structural Operational Semantics*. John Wiley & Sons, Inc., 1990.

[HNSSH97] Nigel W. O. Hutchison, Ute Neuhaus, Manfred Schmidt-Schauß, and Cordelia V. Hall. Natural expert: A commercial functional programming environment. *Journal of Functional Programming*, 7(2):163–182, March 1997.

[HO80]   Gérard Huet and Derek C. Oppen. Equations and rewrite rules:
            A survey. In Ronald V. Book, editor, *Formal Language Theory,
            Perspectives and Open Problems*, pages 349–405. Academic Press,
            1980.

[How89]   Douglas J. Howe. Equality in lazy computation systems. In *Proceedings, Fourth Annual Symposium on Logic in Computer Science*,
            pages 198–203, Asilomar Conference Center, Pacific Grove, California, 5–8 June 1989. IEEE Computer Society Press.

[How96]   Douglas J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*,
            124(2):103–112, 1 February 1996.

[Hue80]   Gérard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, October 1980.

[KBdV03]   Jan Willem Klop, Marc Bezem, and Roel de Vrijer, editors. *Term Rewriting Systems*. Number 55 in Cambridge Tracts in Theoretical
            Computer Science. Cambridge University Press, 2003.

[Klo92]   J. W. Klop. Term rewriting systems. In S. Abramsky, Dov M. Gabbay, and S. E. Maibaum, editors, *Handbook of Logic in Computer
            Science*, volume 2 – Background: Computational Structures, pages
            1–116. Oxford University Press, Inc., 1992.

[KSS98]   Arne Kutzner and Manfred Schmidt-Schauß. A nondeterministic
            call-by-need lambda calculus. In *International Conference on Functional Programming 1998*, pages 324–335. ACM Press, 1998.

[Kut99]   Arne Kutzner. *Ein nichtdeterministischer call-by-need Lambda-Kalkül mit erratic Choice: Operationale Semantik, Programmtransformationen und Anwendungen.* PhD thesis, Johann Wolfgang
            Goethe-Universität, Frankfurt, October 1999.

[KvOvR93]   Jan Willem Klop, Vincent van Oostrom, and Femke van Raamsdonk. Combinatory reduction systems: introduction and survey.
            Theoretical Computer Science, 121(1–2):279–308, December 1993.

[Lan65a]   P. J. Landin. A correspondence between ALGOL 60 and church's
            lambda-notation: Part I. *Communications of the ACM*, 8(2):89–101, February 1965.

[Lan65b] P. J. Landin. A correspondence between ALGOL 60 and church's lambda-notation: Part II. *Communications of the ACM*, 8(3):158–167, March 1965.

[Las98a] S. B. Lassen. Relational reasoning about contexts. In Gordon and Pitts [GP98], pages 91–136.

[Las98b] Søren Bøgh Lassen. *Relational Reasoning about Functions and Non-determinism*. PhD thesis, University of Aarhus, Department of Computer Science, Ny Munkegade, building 540, DK-8000 Aarhus C, December 1998.

[Lau93] John Launchbury. A natural semantics for lazy evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 144–154. ACM Press, 1993.

[LP00] Søren Bøgh Lassen and Corin Pitcher. Similarity and bisimilarity for countable non-determinism and higher-order functions. In Andrew Gordon, Andrew Pitts, and Carolyn Talcott, editors, *Electronic Notes in Theoretical Computer Science*, volume 10. Elsevier, 2000.

[Man04] Matthias Mann. Towards Sharing in Lazy Computation Systems. Frank report 18, Institut für Informatik, J.W. Goethe-Universität Frankfurt am Main, December 2004.

[Man05] Matthias Mann. Congruence of bisimulation in a non-deterministic call-by-need lambda calculus. *Electronic Notes in Theoretical Computer Science*, 128(1):81–101, May 4, 2005.

[Mil71] Robin Milner. An algebraic definition of simulation between programs. In D. C. Cooper, editor, *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, pages 481–489, London, September 1971. William Kaufmann.

[Mil77] Robin Milner. Fully Abstract Models of Typed lambda-Calculi. *Theoretical Computer Science*, 4(1):1–22, 1977.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Mil89]   Robin Milner. *Communication and Concurrency*. Prentice Hall
            International Series in Computer Science. Prentice Hall, 1989.

[MN98]   Richard Mayr and Tobias Nipkow. Higher-order rewrite systems
            and their confluence. *Theoretical Computer Science*, 192:3–29, 1998.

[Mor68]   J.H. Morris. *Lambda-Calculus Models of Programming Languages*.
            PhD thesis, MIT, 1968.

[Mos90]   Peter D. Mosses. Denotational semantics. In Jan van Leeuwen,
            editor, *Handbook of Theoretical Computer Science*, volume B, pages
            575–631. Elsevier Science Publishers, 1990.

[MOW98]   John Maraist, Martin Odersky, and Philip Wadler. The call-by-need
            lambda calculus. *Journal of Functional Programming*, 8(3):275–
            317, May 1998.

[MS99]   Andrew Moran and David Sands. Improvement in a lazy context:
            an operational theory for call-by-need. In *POPL '99: Proceedings
            of the 26th ACM SIGPLAN-SIGACT symposium on Principles of
            programming languages*, pages 43–56. ACM Press, January 1999.

[MSC99a]   A. K. Moran, D. Sands, and M. Carlsson. Erratic Fudgets: A se-
            mantic theory for an embedded coordination language. In *the Third
            International Conference on Coordination Languages and Models;
            COODINATION'99*, number 1594 in Lecture Notes in Computer
            Science, pages 85–102. Springer-Verlag, April 1999. Extended avail-
            able: [MSC99b].

[MSC99b]   A. K. Moran, D. Sands, and M. Carlsson. Erratic Fudgets: A
            semantic theory for an embedded coordination language (extended
            version). Extended version of [MSC99a], February 1999.

[MST96]   Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. From oper-
            ational semantics to domain theory. *Information and Computation*,
            128(1):26–47, July 1996.

[MTH90]   Robin Milner, Mads Tofte, and Robert Harper. *The Definition of
            Standard ML*. IT Press, Cambridge, Massachusetts, 1990.

[NNH99]   Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Princi-
            ples of Program Analysis*. Springer-Verlag, 1999.

[Ong93]   C.-H. Luke Ong. Non-determinism in a functional setting. In *Logic in Computer Science*, pages 275–286, 1993.

[Par81]   David Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 1981.

[Pau89]   Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, 1989.

[Pit97]   A. M. Pitts. Operationally-based theories of program equivalence. In P. Dybjer and A. M. Pitts, editors, *Semantics and Logics of Computation*, Publications of the Newton Institute, pages 241–298. Cambridge University Press, 1997.

[PJ87]   Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall International, 1987.

[PJ03]   Simon Peyton Jones, editor. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.

[PJS89]   Simon L. Peyton Jones and Jon Salkild. The spineless tagless g-machine. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 184–201, New York, NY, USA, 1989. ACM Press.

[Pla93]   David A. Plaisted. Equational reasoning and term rewriting systems. In Dov M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 1 – Logical Foundations. Oxford University Press, Inc., 1993.

[Plo75]   G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(?):125–159, 1975.

[Plo77]   Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, December 1977.

[PS00]   Benjamin C. Pierce and Davide Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. *Journal of the ACM*, 47(3):531–584, May 2000.

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

[PvE99]   Rinus Plasmeijer and Marko van Eekelen. Keep it Clean: A unique approach to functional programming. *SIGPLAN Notices*, 34(6):23–31, June 1999.

[RDRP04]   Simona Ronchi Della Rocca and Luca Paolini. *The Parametric Lambda Calculus, A Meta-Model for Computation*. Texts in Theoretical Computer Science. An EACTS Series. Springer-Verlag, Berlin, Heidelberg, New-York, 2004.

[Sab03]   David Sabel. Realising nondeterministic I/O in the Glasgow Haskell Compiler. Frank report 17, Institut für Informatik, J.W. Goethe-Universität Frankfurt am Main, December 2003.

[San91]   D. Sands. Operational theories of improvement in functional languages (extended abstract). In *Proceedings of the Fourth Glasgow Workshop on Functional Programming*, Workshops in Computing Series, pages 298–311, Skye, August 1991. Springer-Verlag.

[San94]   Davide Sangiorgi. The lazy lambda calculus in a concurrency scenario. *Information and Computation*, 111(1):120–153, 15 May 1994.

[San97]   David Sands. From SOS rules to proof principles. Technical report, Chalmers University of Technology and Göteborg University, 1997.

[San98]   D. Sands. Improvement theory and its applications. In Gordon and Pitts [GP98], pages 275–306.

[Sch86]   David A. Schmidt. *Denotational semantics: a methodology for language development*. Allyn and Bacon, 1986.

[Sch00]   Marko Schütz. *Analyzing Demand in Non-Strict Functional Programming Languages*. Dissertation, Johann Wolfgang Goethe-Universität, Frankfurt, 2000.

[SGM02]   David Sands, Jörgen Gustavsson, and Andrew Moran. Lambda calculi and linear speedups. In T. Æ. Mogensen, D.A. Schmidt, and I. Hal Sudborough, editors, *The essence of computation: complexity, analysis, transformation*, number 2566 in Lecture Notes in Computer Science, pages 60–82. Springer-Verlag, Berlin, 2002.

[SS92]   Harald Søndergard and Peter Sestoft. Non-determinism in Functional Languages. *The Computer Journal*, 35(5):514–523, 1992.

[SS03a]   Manfred Schmidt-Schauß. FUNDIO: A Lambda-Calculus with a `letrec`, `case`, Constructors, and an IO-Interface: Approaching a Theory of `unsafePerformIO`. Frank report 16, Institut für Informatik, J.W. Goethe-Universität Frankfurt, September 2003.

[SS03b]   Manfred Schmidt-Schauß. Funktionale Programmierung 1. www.ki.informatik.uni-frankfurt.de, Summer 2003. Lecture Notes.

[SSSS04]   Manfred Schmidt Schauß, Marko Schütz, and David Sabel. On the Safety of Nöcker's Strictness Analysis. Frank report 19, Institut für Informatik, J.W.Goethe-Universität Frankfurt, Germany, 2004.

[Tur36]   Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936. Reprinted in [Dav04].

[Tur79]   D. A. Turner. A new implementation technique for applicative languages. *Software—Practice and Experience*, 9(1):31–49, January 1979.

[Tur85]   D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 1–16. Springer-Verlag, 1985.

[Wad78]   Christopher P. Wadsworth. Approximate reduction and lambda calculus models. *SIAM Journal of Computing*, 7(3):337–356, August 1978.

[Win93]   Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. Foundations of Computing. MIT Press, Cambridge, Massachusetts, 1993.

Index

Curriculum Vitae

Matthias Mann

Weinstr. 8c
D-60435 Frankfurt
Germany

Date of birth: 9th of April, 1972
Place of birth: Frankfurt/Main, Germany

Education

1999–2005	Postgraduate under supervision of Prof. Dr. Manfred Schmidt-Schauß, Artificial Intelligence and Software Technology
1992–1999	Studies of computer science at Johann-Wolfgang-Goethe Universität Frankfurt, Degree in computer science. Thesis:
	"Gleichheitsanalyse von Ausdrücken in nicht-strikten funktionalen Programmiersprachen unter Verwendung der Kontextanalyse" supervised by Prof. Dr. Manfred Schmidt-Schauß
1982–1991	Heinrich-von-Gagern Gymnasium Frankfurt (comprehensive secondary school), University-entrance diploma
1978–1982	Werner-von-Siemens Grundschule (primary school), Maintal