Congruence of Bisimulation in a Non-Deterministic Call-By-Need Lambda Calculus

Matthias Mann

Institut für Informatik
Johann Wolfgang Goethe-Universität
Postfach 11 19 32
D-60054 Frankfurt, Germany
mann@cs.uni-frankfurt.de

Abstract

We present a call-by-need λ-calculus $\lambda_{ND}$ with an erratic non-deterministic operator pick and a non-recursive let. A definition of a bisimulation is given, which has to be based on a further calculus named $\lambda_{=}$, since the naïve bisimulation definition is useless. The main result is that bisimulation in $\lambda_{=}^{=} \approx$ is a congruence and contained in the contextual equivalence.

The proof is a non-trivial extension of Howe’s method. This might be a step towards defining useful bisimulation relations and proving them to be congruences in calculi that extend the $\lambda_{ND}$-calculus.

Key words: Bisimulation, Congruence, Contextual Equivalence, Non-determinism, Call-by-need Lambda Calculus

1 Introduction

Equality plays a prominent role in reasoning about programs. Thus specifically for λ-calculi, there is a certain range of concepts when two terms should be considered equal. First, there is the notion of convertibility, i.e. two terms are equivalent if they could be transformed to each other according to the conversion rules of the calculus. Usually conversion is permitted inside arbitrary contexts, i.e. program fragments, hence convertibility is a congruence.

For deterministic calculi, there is a large number of reasonable equations, e.g. useful program transformations, which neither are provable by, nor stand in contradiction to, convertibility. Hence there is a serious interest in λ-theories (cf. [5, Part IV]), that is, consistent extensions of the λ-calculus which are closed under derivation.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science

URL: www.elsevier.nl/locate/entcs
The contextual equivalence due to [23], which discriminates terms by their behaviour in all contexts, falls into this category. Typically, termination is observed (cf. [19]). Thus, with \( \downarrow \) denoting termination while \( C \) stands for program contexts, contextual equivalence \( \simeq_c \) can be expressed as

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

This obviously establishes a congruence and since it is based on the observation of some behaviour, it is often called observational congruence. We consider contextual equivalence more significant than convertibility for several reasons. First, contextual equivalence does not directly depend on the reduction rules of a calculus and therefore can be used as a separate justification for its design. Secondly, contextual equivalence validates more meaningful equations than convertibility, the latter of which may, e.g., only relate terms of the same asymptotic complexity in some cases (cf. [29]).

In a non-deterministic setting, on the other hand, convertibility in general leads to an inconsistent theory. Hence there are, of course, convertible terms which are not contextual equivalent. But in many cases, the deterministic part of a calculus can be proven sound w.r.t. contextual equivalence.

Thus contextual equivalence is of great interest in the important field of correct program transformations, for both, deterministic and non-deterministic calculi. However, proofs of contextual equivalence could turn out to be non-trivial, since all contexts have to be taken into account. Hence it is common to reduce the number of contexts by a context lemma (cf. [19]), e.g. in [20] the observation of specific machine configurations is sufficient whereas [21,14,13,31] use evaluation contexts. This is unquestionably a very useful, and for many tasks adequate, approach. But it does not resolve the issue in principle, since still a generally infinite number of contexts has to be considered.

Bisimulation, the origins of which date back to the work of Park [25] and Milner [18], provides a more stepwise approach for proving equations and hence bisimulation techniques have been applied frequently to functional programming (e.g. [2,27,8,9]) since then. In this area, there is some variety of relations, but in general the definition of bisimilarity involves the greatest fixed point of some monotonic operator.

Because of this, a bisimulation proof can be very concise where the corresponding proof of contextual equivalence is subtle, as example 4.9 will show. So it lends itself to a powerful proof instrument, but in order to employ it for proving correctness of program transformations, one has to ensure that it is a congruence. E.g. Groote and Vaandrager, in the introduction of [10], emphasise that proving bisimulation a congruence is vital. But as the effort in [2,11,12] shows, this is in general not a trivial task. Moreover, we demonstrate in example 3.3, that within the scope of non-determinism combined with sharing, bisimulation has to be designed carefully.

The aim of this work now is to establish a sensible definition of bisimulation for a non-deterministic call-by-need \( \lambda \)-calculus and to show that it is
sound w.r.t. contextual equivalence. Therefore the structure of the paper is as follows: In section 2, after we survey several $\lambda$-calculi with non-determinism and/or sharing, we discuss the major techniques for proving bisimulation a congruence.

The $\lambda_{ND}$-calculus, the subject of our study, will be introduced in section 3 then. We have intentionally chosen a very basic calculus to act as a starting point for further studies. Since for several reasons we will explain there, a definition of bisimulation in $\lambda_{ND}$ working directly with let-environments is problematic, we develop in section 4 with the $\lambda_\approx$-calculus a way to prune the evaluation in environments at an arbitrary finite depth. We accomplish this by adapting the reduction rules, so that bisimulation may be based upon reduction to pure abstractions without a surrounding let-environment while recording every possible outcome of the original environment. This empowers us to define bisimulation and eventually prove it a congruence in theorem 5.2 by an extension of Howe’s method in [11,12], i.e. that the so-called “precongruence candidate” is preserved under reduction. The section concludes with its main result, namely that in $\lambda_\approx$ bisimulation implies contextual equivalence.

In section 6, the link between the $\lambda_{ND}$- and the $\lambda_\approx$-calculus is established. The achievement of theorem 6.3 is, that the contextual equivalence of the $\lambda_\approx$-calculus agrees with the one of the $\lambda_{ND}$-calculus. Its proof relies heavily on the diagram method as e.g. [14,13] use it. Due to space limitations we present sketches for the most important proofs and will refer to [16] for supplying complete evidence.

2 Related Work

Of course, there has been a lot of research on extended $\lambda$-calculi and also a fair amount on how to prove bisimulation a congruence in this area. Since it seems impossible to take all of the publications on this subject into account, we will briefly discuss only some of the related work.

2.1 Non-Determinism and Sharing in $\lambda$-Calculi

When introducing a non-deterministic construct into a programming language or, e.g. a $\lambda$-calculus, a number of questions have to be clarified. Apart from the classification of non-determinism as e.g. in [32], we consider a major topic the decision, what kind of terms should be permitted to be copied.

Non-determinism in languages without sharing, i.e. those that retain a copying ($\beta$)-rule like e.g. [24,7,30], is completely different from our work because it will distinguish $\lambda x. (x + x)$ from $\lambda x. (2 * x)$. Likewise is the situation with [6], since, as usual also in explicit substitution-calculi (cf. [1]), substitutions are distributed over applications and hence duplicated.

The deterministic call-by-need calculi of [4,3,17] realise explicit sharing using a let-construct (or special syntactic entities, as is the case in [3]) and
restrict copying to abstractions. However, their equational theory is based on convertibility rather than on contextual equivalence.

Thus the calculi in [21,14,13,31] which all provide a non-deterministic choice, sharing and contextual equivalence roughly represent the direction of our investigations. Though there are a few differences. Since these papers do not discuss bisimulation, it seemed sensible to carry out our studies in a rather elementary calculus, as this should increase readability, too.

Hence, like the work of [14,13], the $\lambda_{ND}$-calculus only has a non-recursive \texttt{let}, whereas the calculi in [21,31] provide recursive bindings. Furthermore, like [14,13] but in contrast to [21,31], the $\lambda_{ND}$-calculus neither has a \texttt{case} nor data constructors.

2.2 Proving Bisimilarity a Congruence

As indicated before, for non-deterministic $\lambda$-calculi in combination with sharing there has not been much research on bisimulation in relation to contextual equivalence. The lazy lambda calculus of [2] is a deterministic, and in fact call-by-name $\lambda$-calculus. Denotational approaches of this kind are connected to operational techniques by Pitts, but [26] does not incorporate non-determinism.

Of the purely operational methods, the rule format of [28] is deterministic, while the approach of Howe [11,12] in principle permits non-deterministic evaluation. Also in [10] bisimulation is shown a congruence, but their rule format is too restricted to represent our calculus. As Howe already remarks, the definition of the precongruence candidate in [10] is too weak to be proven stable under substitutions [12, Lemma 3.2]. Like his earlier work [11], the technique of Howe assumes that every term may be copied, and hence has to be adapted in order to cope with sharing.

Even though not dealing with sharing, Sands demonstrates in [27] the extensibility of Howe’s approach by applying it to express improvements; and also [8,9] makes use of the method, but for typed programs. So the decision to base our work on [11] looked most promising.

3 $\lambda_{ND}$ – a Non-Deterministic $\lambda$-Calculus with Sharing

The $\lambda_{ND}$-calculus closely resembles the one of [14], apart from the difference that the nondeterministic choice is modelled by the syntactic construct \texttt{pick} rather than a constant. In the grammar of figure 1, let $V$ denote a non-terminal for variables. Hence the terms of the language, referred to as $\Lambda_{ND}$, are variables or formed by application as well as the operators $\lambda$, \texttt{let} and

\[
E ::= V \mid (\lambda x. E) \mid (E \; E) \mid (\texttt{let} \; x = E \; \texttt{in} \; E) \mid (\texttt{pick} \; E \; E)
\]

Fig. 1. Syntax for expressions in the language $\Lambda_{ND}$
pick. Since the symbol $=$ is part of the let-construct, we use $\equiv$ for syntactic equality up to renaming of bound variables. Furthermore, we write $s[t/x]$ for substituting every free occurrence of $x$ in $s$ by $t$ and adopt the distinct variable convention, i.e. suppose all bound variables to be distinct from each other and the free variables. We implicitly assume this convention to take effect after every reduction step, so e.g. the double occurrence of the term $\lambda y.r$ in the specification of the (cp)-rule below, does not pose a problem.

$$\begin{align*}
\text{let } x &= (\text{let } y = t_y \text{ in } t_x) \text{ in } s \xrightarrow{\text{let}} \text{let } y = t_y \text{ in } (\text{let } x = t_x \text{ in } s) \quad (\text{llet}) \\
(\text{let } x = t_x \text{ in } s) t &\xrightarrow{\text{lapp}} \text{let } x = t_x \text{ in } (s t) \quad (\text{lapp}) \\
(\lambda x.s) t &\xrightarrow{\text{lbeta}} \text{let } x = t \text{ in } s \quad (\text{lbeta}) \\
\text{pick } s t &\xrightarrow{\text{ndl}} s \quad (\text{ndl}) \\
\text{pick } s t &\xrightarrow{\text{ndr}} t \quad (\text{ndr}) \\
\text{nd} &\xrightarrow{=} \text{ndl} \cup \text{ndr} \quad (\text{nd}) \\
\text{let } x = \lambda y.r \text{ in } D[x] &\xrightarrow{\text{cp}} \text{let } x = \lambda y.r \text{ in } D[\lambda y.r] \quad (\text{cp})
\end{align*}$$

Fig. 2. The reduction rules of the $\lambda_{\text{ND}}$-calculus

As usual, a context is a term with a single hole and with $C[s]$ we denote filling the hole of a context $C$ with the term $s$. Note, that the distinct variable convention does not apply to variables which become bound in the hole of a context. The $\lambda_{\text{ND}}$-calculus will be equipped with an operational semantics based on a small-step reduction relation. We will give a succinct account of these rules from figure 2. The purpose of (llet) and (lapp) is mainly to rearrange let-bindings for subsequent reductions. The ordinary ($\beta$)-rule is superseded by (lbeta) which just creates a let-binding. The rules (ndl) and (ndr) implement the non-deterministic choice and are combined into (nd).

With (cp) one occurrence of a variable bound to an abstraction may be replaced with a copy of this abstraction. Note that in contrast to the $\lambda_{\approx}$-calculus of section 4, the rule (cp) copies an abstraction only to one location at a time. This not only conforms with earlier work on call-by-need $\lambda$-calculi (cf. [4,3,17]) but is also closer to the implementation of lazy functional languages than a simultaneous substitution would be.

In order to obtain call-by-need evaluation, the normal-order reduction defined later will always take place in reduction contexts. These do not introduce a hole in the argument of an application, nor in the binding of a let, nor within a $\lambda$-term either. In figure 3 the sets $\mathcal{R}$ and $\mathcal{S}$ of reduction and surface contexts are designated by the symbols $R$ and $S$ respectively. Surface contexts do not possess a hole under an abstraction and will become more important in section 4. Note, that every reduction context is also a surface context.
\[ A_L ::= [ ] e \quad \quad A_L^* ::= [ ] \mid A_L[A_L^*] \]

\[ L_R ::= \text{let } x = e \text{ in } [ ] \quad \quad L_R^* ::= [ ] \mid L_R[L_R^*] \]

\[ R ::= L_R^*[A_L^*] \mid L_R^*[\text{let } x = A_L^* \text{ in } R[x]] \]

\[ S ::= [ ] \mid S e \mid e S \mid \text{let } x = e \text{ in } S \mid \text{let } x = S \text{ in } e \mid \text{pick } S e \mid \text{pick } e S \]

Fig. 3. Major context classes for \( \Lambda_{\text{ND}} \)

Let \( a \in \{ \text{let, lapp, lbeta, ndl, ndr, cp} \} \) be any of the reduction rules in figure 2. We then denote with \( \overset{\mathcal{R}, a}{\rightarrow} \) the application of the rule \( (a) \) in any reduction context \( R \in \mathcal{R} \) and write \( \overset{*}{\rightarrow} \) for the reflexive-transitive closure of reduction relations. The normal-order reduction of the following definition uniquely identifies a normal-order redex and is, except for the non-deterministic rules, also unique.

**Definition 3.1** A reduction \( s \overset{\mathcal{R}, a}{\rightarrow} t \) is called normal-order and depicted by \( s \overset{n, a}{\rightarrow} t \) if it is one of the following.

(i) If \( s \equiv L_R^*[A_L^*[r]] \) and rule (lapp), (lbeta), (ndl) or (ndr) is applied to \( r \).

(ii) If \( s \equiv L_R^*[\text{let } x = A_L^*[r] \text{ in } R[x]] \) with some reduction context \( R \) such that rule (lapp), (lbeta), (ndl) or (ndr) is applied to \( r \).

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

(iv) If rule (llet) is applied as follows:

\[ s \equiv L_R^*[\text{let } x = (\text{let } y = t_y \text{ in } t_x) \text{ in } R[x]] \overset{n, \text{llet}}{\rightarrow} L_R^*[\text{let } y = t_y \text{ in } (\text{let } x = t_x \text{ in } R[x])] \equiv t \]

The above definition complies with [13] and slightly differs from [4] as discussed in [13, p. 42]. Intuitively, it can be described as follows. Descend into contexts of the form \( L_R \) and subsequently \( A_L \) until (ndl), (lapp) or (lbeta) becomes applicable, the case (i). If during this process a variable is encountered, follow its binding. Whenever possible, perform (cp) or (llet) for the variable in question, i.e. cases (iii) and (iv) respectively. Otherwise, in case (ii), if the variable is bound to an application, descend into the \( A_L^* \)-context as far as possible in order to apply (ndl), (lapp) or (lbeta).

The notion of convergence is then defined by a normal-order reduction sequence to a term of the form \( L_R[\lambda x.t] \), i.e. a weak head normal form, WHNF for short. So we write \( s \downarrow t \) if and only if \( s \overset{n, \rightarrow}{\rightarrow} t \) and \( t \) is a WHNF, \( s \not\downarrow \) if there exists such a \( t \) and \( s \not\downarrow \) if not. Apparently, the normal-order reduction is neither confluent nor terminating, i.e. a term may reduce to multiple weak head normal forms or none at all.
The procedure to determine the normal-order redex is quite complex, so it is not obvious how to represent the normal-order reduction directly by a structural operational semantics. E.g. the structured evaluation systems of [12], apart from being geared to big-step operational semantics, seem not capable of this. This arises from the fact that both, normal-order reducible terms and weak head normal forms, could be formed with the let-operator.

3.1 Contextual Equivalence

Convergence, as defined in the previous section, 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. The notion of “must convergence”, i.e. that all normal-order reduction sequences starting with \( s \) lead to a WHNF, also makes sense for a non-deterministic calculus (cf. [21,13,31]). However, for reason of simplicity, the following definition only regards “may convergence”.

Definition 3.2 The contextual approximation \( \trianglelefteq_{\text{ND,}e} \) is defined by

\[
s \trianglelefteq_{\text{ND,}e} t \iff \forall C : C[s] \Downarrow \Rightarrow C[t] \Downarrow
\]

and contextual equivalence \( \simeq_{\text{ND,}e} \) by \( s \simeq_{\text{ND,}e} t \iff s \trianglelefteq_{\text{ND,}e} t \land t \trianglelefteq_{\text{ND,}e} s \).

The goal is to define bisimulation in such a manner that it implies contextual equivalence. The following example makes clear that it is impossible to employ the usual “reduce to weak head normal form and apply to fresh arguments”-approach like e.g. in [2].

Example 3.3 Let the combinators \( K \equiv \lambda x_1.\lambda x_2.x_1 \) and \( K2 \equiv \lambda y_1.\lambda y_2.y_2 \) as well as the non-converging term \( \Omega \equiv (\lambda z.z \, z) \, (\lambda z.z \, z) \) be as usual.

Then \( s \equiv \text{let } v = \text{pick } K \, K2 \text{ in } \lambda w.v \) and \( t \equiv \lambda w.\text{pick } K \, K2 \) could be distinguished by the context \( C \equiv \text{let } f = [ \] in \((f \, K) \,(f \, K) \, \Omega \, \Omega \, K)\) in the following way: Concerning \( t \) we may construct a normal-order reduction sequence \( C[t] \xrightarrow{\ast} L_n[K] \) 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 previous example also reveals that in the \( \lambda_{\text{ND}} \)-calculus the transformation \( \lambda y.\text{let } x = s \text{ in } t \ \leadsto \text{let } x = s \text{ in } \lambda y.t \), i.e. shifting let over \( \lambda \), in general is not correct w.r.t. contextual equivalence. This is so, because the term \( \text{let } v = \text{pick } K \, K2 \text{ in } \lambda w.v \) becomes \( \lambda w.\text{let } v = \text{pick } K \, K2 \text{ in } v \) by a reverse application of this transformation. One could simply play through the example with these two terms or, alternatively, argue that the latter is contextual equivalent (cf. [13, rule (ucp)]) to the term \( t \) in the example.
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. There may be several ways to adjust bisimulation so that examples of the above sort work, but it is not clear which one will really produce a suitable definition of bisimulation.

Our approach eliminating the environments has the additional benefit that proving the precongruence candidate stable under the rule (illet) becomes obsolete, a task which seems to be infeasible for the other variations of a definition we have tried.

So before we introduce the special calculus $\lambda_\approx$ which eliminates let-environments by collecting all possible outcomes, we illustrate by an example that in the $\lambda_{ND}$-calculus the rule (illet) in general is necessary to find a WHNF.

Example 3.4 Consider the term $s \equiv \text{let } x = (\text{let } y = t_y \text{ in } \lambda z.t) \text{ in } x$ which obviously has a WHNF by the following normal-order reduction:

\[
\begin{align*}
\text{let } x = (\text{let } y = t_y \text{ in } \lambda z.t) \text{ in } x \\
\text{n.,illet} & \quad \text{let } y = t_y \text{ in } (\text{let } x = \lambda z.t \text{ in } x) \\
\text{n.,cp} & \quad \text{let } y = t_y \text{ in } (\text{let } x = \lambda z.t \text{ in } \lambda z.t)
\end{align*}
\]

Apparently, the effect of (illet) cannot be accomplished neither by a different scope nor target for the (cp)-rule. Obviously, making a copy of the whole environment let $y = t_y \text{ in } \lambda z.t$ is in general no option either, since then e.g. for a term of the form let $f = (\text{let } y = \text{pick } K K2 \text{ in } \lambda x.x y) \text{ in } f(f \Omega)$ this would alter its value w.r.t. contextual approximation.

4 $\lambda_\approx$ – Approximating Expressions of the $\lambda_{ND}$-Calculus

As figure 4 shows, a special constant $\otimes$ is added to the language which is now designated by $\Lambda_\approx$. The reduction rules of the $\lambda_\approx$-calculus in figure 5 evolve

\[
E ::= V \mid \otimes \mid (\lambda x.E) \mid (E \ E) \mid (\text{let } x = E \text{ in } E) \mid (\text{pick } E \ E)
\]

Fig. 4. Syntax for expressions in the language $\Lambda_\approx$

from the ones in $\lambda_{ND}$ as follows. First, by the rule (stop) which may reduce every non-$\otimes$ term to $\otimes$, a further level of non-determinism is introduced. As there is no rule for $\otimes$, 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 copy terms that could not be reduced further, namely $\otimes$ and ab-
stractions, and garbage-collect their binding with the rule (cpa) in parallel. So we are able to show in section 6 that the original (cp)-rule becomes obsolete.

Furthermore, all these reductions will be permitted inside arbitrary surface contexts, which are denoted by the symbol $S$ as before. Hence there is no need for the rule (llet) either, since we could first reduce inside the binding of a let-environment before collapsing it using (cpa). We will give a more detailed account on this process in section 6 where we show that convergence in $\lambda_{ND}$ and $\lambda\approx$ coincides.

As indicated above, the reductions of the $\lambda\approx$-calculus may take place in surface contexts; hence $S \rightarrow_{\lambda\approx} \lambda_{\approx}$ stands for an application of the rule (a) inside any surface context $S \in S$. Since it is possible to evaluate up to an arbitrary depth before cutting off with the rule (stop), we call this an approximation reduction and will omit the subscript $\lambda\approx$ for the remainder of this section if no confusion arises. The notion of convergence in the $\lambda_{\approx}$-calculus is then defined by $s \Downarrow_{\lambda} \lambda x.t \iff s \rightarrow_{S, \approx}^{*} \lambda x.t$, i.e. if there exists an approximation reduction sequence to an abstraction.

### 4.1 Transformation on Reduction Sequences

In anticipation of bisimulation proofs, it can be shown that applications of the rules (cpa) and (lbeta) never do any harm to an approximation reduction sequence. In case of the former, this is valid only w.r.t. some (stop)-reductions inside arbitrary contexts.

**Definition 4.1** A $C, \text{stop} \rightarrow_{\lambda_{\approx}}$-reduction is called internal, depicted by $i, C, \text{stop} \rightarrow_{\lambda_{\approx}}$, if $C \in C$ is a context which is not a surface context, i.e. $C \notin S$.

These internal (stop)-reductions may always be moved to the end of an approximation reduction sequence.
Lemma 4.2 Let $s, \lambda x.t \in \Lambda_\approx$ be terms with $s \xrightarrow{(S, \lambda x.t \cup \ i, \ \text{stop})^*_{\lambda x.t}} \lambda x.t$. Then there is also a reduction $s \xrightarrow{(S, \lambda x.t \cup \ i, \ \text{stop})^*_{\lambda x.t}} \lambda x.t$ such that $\lambda x.t \xrightarrow{i, \ \text{stop}}^* \lambda x.t$ holds.

Internal (stop)-reductions may become necessary to clean up forking situations as follows. Consider the case that an ordinary (stop)-reduction is applied to an abstraction bound to a variable in a let-expression. If the rule (cpa) is used afterwards for this let-expression, the $\otimes$-terms previously introduced may be found under abstractions. Therefore a simple commutation of the above (stop)- and (cpa)-reductions cannot achieve the same effect.

Hence we can show that for every converging approximation reduction sequence the preference of reductions by rule (cpa) leads to abstractions with some internal (stop)-reductions delayed.

Lemma 4.3 Let $s, s', \lambda x.t$ be terms so that $s \xrightarrow{(S, \text{cpa})} s'$ and $s \xrightarrow{(S, \lambda x.t \cup \ i, \ \text{stop})^*_{\lambda x.t}} \lambda x.t$ hold. Then $s'$ has an approximation reduction to an abstraction $\lambda x.t'$ which differs from $\lambda x.t$ only by internal (stop)-reductions, i.e. $\lambda x.t' \xrightarrow{i, \ \text{stop}}^* \lambda x.t$ holds.

For reductions by rule (lbeta), a stronger statement applies. That is to say, if (lbeta) is applicable in a surface context, it does not matter whether a different reduction is performed first.

Lemma 4.4 Let $s, t$ be terms such that $s \xrightarrow{(S, \text{lbeta})} t$ holds. Then for all abstractions $\lambda z.q$ we have $s \xrightarrow{(S, \lambda z.q \cup \ i, \ \text{cpa})^*_{\lambda z.q}} \lambda z.q$ if and only if $t \xrightarrow{(S, \lambda z.q \cup \ i, \ \text{cpa})^*_{\lambda z.q}} \lambda z.q$ holds.

The proofs for lemma 4.3 and 4.4 use the technique of complete sets of forking and, in the case of lemma 4.2, commuting diagrams (cf. [13,31]). Further details can be found in [16, section 2.3.1].

Another essential result consists in reordering converging approximation reduction sequences so that reduction first takes place inside the let-bindings.

Theorem 4.5 For every reduction $\text{let } x = s \in t \xrightarrow{(S, \lambda z.q \cup \ i, \ \text{cpa})^*_{\lambda z.q}} \lambda z.q$ there is also an approximation reduction sequence of the following form:

$$\text{let } x = s \in t \xrightarrow{(S, \lambda z.q \cup \ i, \ \text{cpa})^*_{\lambda z.q}} \lambda z.q$$

where $s'$ represents $\otimes$ or an abstraction.

Proof. Induction on the length of the approximation reduction sequence. \qed

4.2 Similarity

Owing to the rules (stop) and (cpa), we now have the potential to equip abstractions with the information about their let-environments up to an arbitrary depth. This fact will be exploited through non-determinism, i.e. by considering all possible approximation reductions to abstractions.
We would like to stress the point that we are looking for a method to prove contextual equivalence in the $\lambda_{\text{ND}}$-calculus. So we are not interested in the usual notion of similarity for $\lambda_{\text{ND}}$ per se — which by example 3.3 has already turned out to be incorrect w.r.t. contextual approximation — but rather in a more operational style for determining contextual equivalences. Hence like e.g. Abramsky [2] and Pitts [26] we use the familiar terms “bisimulation” and “bisimilarity”. However, the relation we subsequently will develop would have been classified by Lassen and Pitcher [15] as “mutual similarity” which is different from the bisimilarity defined therein.

**Definition 4.6** The operation $\cdot \approx : \Lambda^0_\approx \times \Lambda^0_\approx \rightarrow \Lambda^0_\approx \times \Lambda^0_\approx$ over relations on closed terms is defined by

$$s' [\eta] \approx t' \iff \forall \lambda x.s : (s' \downarrow \lambda x.s \implies \exists \lambda y.t : (t' \downarrow \lambda y.t \land \forall r \in \Lambda^0_\approx \implies (\lambda x.s) r \eta (\lambda y.t) r))$$

and called an experiment. A relation $\eta \subseteq \Lambda^0_\approx \times \Lambda^0_\approx$ is a simulation if $\eta \subseteq [\eta]_\approx$.

It is clear that $\cdot \approx$ is monotonic, i.e. $\eta_1 \subseteq \eta_2 \implies [\eta_1]_\approx \subseteq [\eta_2]_\approx$, hence its greatest fixed point exists.

**Definition 4.7** Define the similarity $\preceq_b$ to be the greatest fixed point of $\cdot \approx$, i.e. $\preceq_b = \text{gfp}(\cdot \approx)$, and the bisimilarity $\sim_b$ by $s \sim_b t \iff s \preceq_b t \land t \preceq_b s$.

So two terms $s$ and $t$ are considered bisimilar as long as their approximation reduction leads to sets of abstractions such that there are elements from each set which are bisimilar if applied to arbitrary arguments. The next example underpins that this is exactly what we need to obtain the same capability in distinguishing terms as with contexts.

**Example 4.8** As is known, the two terms $s \equiv \text{let } v = \text{pick } K \ K2 \text{ in } \lambda w. v$ and $t \equiv \lambda w. \text{pick } K \ K2$ of example 3.3 could be distinguished by contexts.

Now we can show that $t \not\preceq_b s$ 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, \text{ nd} \Downarrow} \text{let } v = K \text{ in } \lambda w. v \xrightarrow{\text{[]}, \text{ cp} \Downarrow} \lambda w. K$$

$$s \xrightarrow{\text{let } v=\[] \text{ in } \lambda w.v, \text{ nd} \Downarrow} \text{let } v = K2 \text{ in } \lambda w. v \xrightarrow{\text{[]}, \text{ cp} \Downarrow} \lambda w. 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 $\Omega, \Omega, K$ and $\Omega, K, \Omega$, while $\lambda w. K$ does not converge for the former, nor does $\lambda w. K2$ for the latter.

What follows is an example of a proof which is straightforward for similarity but seems rather involved using the definition of contextual approximation.
Example 4.9 Let \( r, s, t \in \Lambda_{\approx}^0 \) be arbitrary closed terms. Then we have

\[
\begin{align*}
    r \lesssim_b t \land s \lesssim_b t & \implies \text{pick} \ r \ s \lesssim_b t
\end{align*}
\]

i.e. if \( t \) behaves “better” than both \( r \) and \( s \), then it is immaterial which one is chosen thereof. So assume \( \text{pick} \ r \ s \Downarrow \lambda y.p \), then \( r \Downarrow \lambda y.p \) or \( s \Downarrow \lambda y.p \). Since by the premise we have \( r \lesssim_b t \) and \( s \lesssim_b t \), the proposition is shown.

We will now extend similarity to open terms. The motivation for doing so is twofold. First, the notion of a congruence is less meaningful when dealing with closed terms. E.g., inferring \( \lambda x.s \sim_b \lambda x.t \) from \( s \sim_b t \) for closed \( s \) and \( t \) does not gain much, since \( x \) is only a dummy variable. Secondly, the proof method interacts closely with the extension of \( \lesssim \) to open terms anyway.

So we have to bear in mind which terms may be copied in the \( \lambda_{\approx} \)-calculus. Since this is the case for \( \odot \) and abstractions only, the technique to use all closing substitutions is not applicable, as the following example substantiates.

Example 4.10 Consider the open terms \( f \ f \) and let \( x = f \) in \( x \ x \) which are contextual equivalent in the \( \lambda_{\ND} \)-calculus since copying variables is permitted (cf. correctness of rule (lcv) in \([13]\)).

But demanding the terms to be bisimilar for every closing substitution is not possible: \( (f \ f)[\text{pick} \ K \ K2/f] \) may yield \( K \ K2 \) which, along the lines of example 3.3, converges if successively applied to the arguments \( \Omega, \Omega \) and \( K \), whereas \( (\text{let} \ x = f \ in \ x \ x)[\text{pick} \ K \ K2/f] \) clearly does not.

Hence what we need is a restriction of the substitutions such that free variables are mapped only to \( \odot \) or closed abstractions.

Definition 4.11 Let \( s, t \in \Lambda_{\approx} \) be (possibly open) terms. We then write \( s \lesssim_{bo} t \) if and only if \( \sigma(s) \lesssim_b \sigma(t) \) holds for all closing substitutions \( \sigma \) whose range \( \text{rng}(\sigma) \) satisfies \( \text{rng}(\sigma) \subseteq \{ p \in \Lambda_{\approx}^0 | p \equiv \odot \lor p \equiv \lambda z.q \} \).

In \([16, \text{section 4.7}]\), we show that an equivalent notion may be defined by considering all closing \text{let}-environments.

4.3 The Precongruence Candidate

In this section, let \( \tau \) stand for any operator of the \( \Lambda_{\approx} \)-language (i.e. \( \odot, \lambda, \text{let}, \text{pick} \) or application) and \( \overline{a}_i \) for a sequence of its operands. With \( \overline{a}_i \), \( \eta \overline{b}_i \), we denote the condition that \( a_i \eta \overline{b}_i \) for every \( i \) holds. A relation \( \eta \subseteq \Lambda_{\approx} \times \Lambda_{\approx} \) is then called \text{operator-respecting}, or \text{compatible}, if and only if \( \overline{a}_i \eta \overline{b}_i \) implies \( \tau(\overline{a}_i) \eta \tau(\overline{b}_i) \) for all operators. A \text{precongruence} is a compatible preorder.

The following defines a relation which is compatible by definition but not necessarily transitive. The intention is to show that it coincides with \( \lesssim_{bo} \) for which the criteria will be developed in this section.
Definition 4.12 Let $\eta \subseteq \Lambda^0_\approx \times \Lambda^0_\approx$ be a preorder. Then define its precongruence candidate $\lesssim_b \subseteq \Lambda_\approx \times \Lambda_\approx$ by

- $x \lesssim_b b$ if $x \in V$ is a variable and $x \lesssim_b^o b$.
- $\tau(\overline{a}_i) \lesssim_b b$ if there exists $\overline{a}_i'$ such that $\overline{a}_i \lesssim_b \overline{a}_i'$ and $\tau(\overline{a}_i) \lesssim_b^o b$.

Howe [11, p. 201] aptly gives the informal account that $a \lesssim_b b$ if $b$ can be obtained from a via one bottom-up pass of replacements of subterms by terms that are larger under $\lesssim_b^o$. As noted before, in the $\lambda$-calculus only $\otimes$ and abstractions may be copied. Hence the following two lemmata reflect our counterpart to [11, Lemma 1] and [12, Lemma 3.2] respectively.

Lemma 4.13 Let $b, b' \in \Lambda_\approx$ be terms. Then $b \lesssim_b b'$ implies $b[\otimes/x] \lesssim_b b'[\otimes/x]$.

Lemma 4.14 For all $b, b' \in \Lambda_\approx$ and closed abstractions $\lambda z.r, \lambda z.r' \in \Lambda^0_\approx$ the following holds: $b \lesssim_b b' \land \lambda z.r \lesssim_b \lambda z.r' \implies b[\lambda z.r/x] \lesssim_b b'[\lambda z.r'/x]$.

Both lemmata are proven by induction on the definition of the precongruence candidate in which we take advantage of how $\lesssim_b^o$ is defined.

Let in the following $\eta_0$ stand for the restriction of a preorder $\eta \subseteq \Lambda_\approx \times \Lambda_\approx$ to closed terms, i.e. $\eta_0 = \eta \cap \Lambda^0_\approx \times \Lambda^0_\approx$, then from the above we can show:

Theorem 4.15 The relation $(\lesssim_b)_0 \subseteq \lesssim_b$ holds, if and only if $\lesssim_b \subseteq \lesssim_b^o$, if and only if $\lesssim_b^o$ is a precongruence.

We will establish the first set inclusion $(\lesssim_b)_0 \subseteq \lesssim_b$ which, by co-induction, follows from $(\lesssim_b)_0 \subseteq [(\lesssim_b)_0]_\approx$, since $\lesssim_b$ is the greatest fixed point of $[\cdot]_\approx$ and contains all simulations. To use induction on the length of converging approximation reductions sequences, we therefore have to show that $(\lesssim_b)_0$ is preserved under every single-step reduction.

5 Proving Similarity a Precongruence

Preservation of $(\lesssim_b)_0$ under every single-step reduction amounts to the condition that $s (\lesssim_b)_0 t \land s \xrightarrow[^s]{\Lambda^\approx} s'$ implies $s' (\lesssim_b)_0 t$ in which the terms $s, s', t$ all are closed. Note that the only closing surface contexts, i.e. those with which open terms can be closed, involve a $L_R$-context somewhere.

It can be shown that for every converging approximation reduction there is also an approximation reduction sequence to the same abstraction, where no approximation reduction takes place in a closing surface context. Hence in the following, it is sufficient to examine top-level reductions on closed terms.

Lemma 5.1 Let $r, s \in \Lambda^0_\approx$ be closed terms such that $r \xrightarrow[^r]{\Lambda^\approx} s$ holds. Then for every closed term $t \in \Lambda^0_\approx$ we have: $r (\lesssim_b)_0 t$ implies $s (\lesssim_b)_0 t$.

Proof. In the case of $a \in \{ndl, nrd, stop\}$ the claim is obvious. The remaining reduction rules are by induction on the definition of the precongruence
candidate, where for (lapp) and (lbeta) compatibility of \( \lessapprox_b \) w.r.t. contexts of the form \([\ ]c\) is applied.

The proof of the rule (cpa) is done distinguishing the cases for \( \otimes \) and abstractions, while exploiting lemma 4.13 and 4.14, respectively. \( \square \)

It is remarkable, that by virtue of Howe’s method we obtain a modular proof. If new reduction rules are added to the calculus, the proof will extend easily because it could be done separately for each of the rules.

Furthermore, using (cpa) instead of (cp) greatly simplifies matters, since by the integrated garbage collection there is no need to keep track of the copied term at its target location w.r.t to its binding in the let-environment.

**Theorem 5.2** The similarity \( \lessapprox_b^o \) is a precongruence.

**Proof.** With lemma 5.1, the premises for theorem 4.15 are satisfied. \( \square \)

Since the language \( \Lambda_{\approx} \) contains strictly more contexts than \( \Lambda_{ND} \) due to the constant \( \otimes \), we define contextual approximation for \( \lambda_{\approx} \) separately. We will show in section 6 that these contexts do not add any computational power.

**Definition 5.3** The contextual approximation \( \lessapprox_{\Lambda_\approx, c} \) for \( \lambda_{\approx} \) is defined by

\[
s \lessapprox_{\Lambda_\approx, c} t \iff \forall C: \ C[s] \downarrow \implies C[t] \downarrow
\]

and contextual equivalence \( \simeq_{\Lambda_\approx, c} \) by \( s \simeq_{\Lambda_\approx, c} t \iff s \lessapprox_{\Lambda_\approx, c} t \land t \lessapprox_{\Lambda_\approx, c} s \).

By virtue of theorem 5.2 it becomes straightforward to show that the similarity \( \lessapprox_b^o \) is contained in the contextual approximation of the \( \lambda_\approx \)-calculus.

**Theorem 5.4** Let \( s, t \in \Lambda_{\approx} \) be terms. Then \( s \lessapprox_b^o t \implies s \lessapprox_{\Lambda_\approx, c} t \) holds.

### 6 Correspondence of Equality in \( \lambda_{ND} \) and \( \lambda_{\approx} \)

Since our aims were a method to prove the contextual equivalence \( \simeq_{\Lambda_{ND, c}} \) in the \( \lambda_{ND} \)-calculus, we have an obligation to show that this is indeed the same as the contextual equivalence \( \simeq_{\Lambda_\approx, c} \) in the \( \lambda_\approx \)-calculus. Hence, we first recall how the contextual approximation is defined:

\[
s \lessapprox_c t \iff (\forall C: \ C[s] \downarrow \implies C[t] \downarrow)
\]

It is quite evident that the correspondence of \( \lessapprox_{\Lambda_{ND, c}} \) and \( \lessapprox_{\Lambda_\approx, c} \) requires the following property: For every term \( t \in \Lambda_{\approx} \) there is a normal-order reduction to a weak head normal form if and only if \( t \) has an approximation reduction to an abstraction. We therefore understand the notion of normal-order reduction in \( \lambda_{ND} \) as extended to terms from \( \Lambda_{\approx} \) in the obvious way, i.e. regarding \( \otimes \) as a constant which has no normal-order reduction.
Moreover, it can be shown that for every $\Lambda_\approx$-context $C$ which distinguishes two terms $s, t \in \Lambda_{\text{ND}}$ there is also a $\Lambda_{\text{ND}}$-context $C'$ such that $C'[s]$ converges but $C'[t]$ does not, or vice versa. For this purpose we may obtain $C'$ from $C$ by simply replacing all occurrences of $\odot$ with $\Omega$, whose normal-order reduction does not terminate.

Thus, the contexts of $\Lambda_{\text{ND}}$ and $\Lambda_\approx$ are equally powerful in distinguishing terms and we also obtain a sufficient condition from the above mentioned property. I.e., we may confine attention to the transformation of converging reduction sequences between the two calculi in the following.

6.1 Transforming $\frac{S}{\Lambda_\approx}$- into $\frac{n}{\Lambda_{\text{ND}}}$-reduction sequences

The process of constructing a normal-order reduction to a WHNF is by induction on the length of a converging approximation reduction. Since $(\text{cpa})$ and $(\text{nd})$ are the only approximation reductions to reach an abstraction within a single step, the induction base should be clear.

For the induction step it is then to show that every approximation reduction may be moved to the end of a normal-order reduction sequence. For approximation reductions which are performed inside surface contexts that are not reduction contexts, this is an obvious task, since the corresponding contexts are disjoint. So it turns out that only reductions by rule $(\text{cpa})$ inside reduction contexts are of particular interest. For these, in [16, section 5.1] a complete set of commuting diagrams w.r.t. normal-order reductions is established. These diagrams do not duplicate $R_{\text{cpa}}$-reductions and hence could be composed by induction which leads to the following result.

Lemma 6.1 Let $r, \lambda x.s \in \Lambda_\approx$ be terms such that $r \frac{S}{\Lambda_\approx}^* \lambda x.s$ holds. Then there is also a normal order reduction $r \frac{n}{\Lambda_{\text{ND}}}^* t$ where $t$ is a WHNF.

Proof. Using the arguments discussed above for an induction on the length of an approximation reduction sequence to an abstraction. □

6.2 Transforming $\frac{n}{\Lambda_{\text{ND}}}$- into $\frac{S}{\Lambda_\approx}$-reduction sequences

Since every reduction context is also a surface context, the only normal-order reductions which are no approximation reductions are those by the rules $(\text{cp})$ and $(\text{llet})$. Since with $(\text{cpa})$ the former has a counterpart in the $\lambda_\approx$-calculus, its treatment is not difficult.

But in order to make the latter superfluous, the reduction strategy has to be adapted so that for a term like $\text{let } x = (\text{let } y = t_y \text{ in } t_x) \text{ in } R[x]$ the approximation reduction first proceeds inside $\text{let } y = t_y \text{ in } t_x$ until $\odot$ or an abstraction is reached, which could be copied into $R[x]$ using $(\text{cpa})$ then. Because of theorem 4.5, this procedure is always possible and may also be applied recursively to the subterm $\text{let } y = t_y \text{ in } t_x$ of the above scenario.
Lemma 6.2 Let \( r, s \in \Lambda_\approx \) be terms such that \( s \) is a WHNF and \( r \xrightarrow{n*}^\Lambda_{\text{ND}} s \) holds. Then there is also an approximation reduction \( r \xrightarrow{\approx^*}^\Lambda_{\text{ND}} \lambda x . t \) to some abstraction.

Proof. By induction on the length of a normal-order reduction sequence. \( \Box \)

Putting all these parts together we achieve the soundness of similarity with respect to contextual approximation, both in the \( \lambda_{\text{ND}} \)- and the \( \lambda_{\approx} \)-calculus.

Theorem 6.3 Let \( s, t \in \Lambda_\approx \) be arbitrary \( \lambda_\approx \)-terms. Then the relation \( s \preccurlyeq^o b \otimes t \) implies that \( s \preccurlyeq_{\lambda_\approx, c} t \) as well as \( s \preccurlyeq_{\Lambda_{\text{ND}}, c} t \) is valid.

Our main objective of proving contextual equivalences in the \( \lambda_{\text{ND}} \)-calculus now becomes a simple consequence.

Corollary 6.4 For all terms \( s, t \in \Lambda_{\text{ND}} \) we have: \( s \sim^o b \otimes t \) implies \( s \simeq_{\Lambda_{\text{ND}}, c} t \).

7 Conclusion and Future Work

To the best of our knowledge, for the first time a sensible bisimulation has been defined for a non-deterministic call-by-need calculus and shown to be sound w.r.t. contextual equivalence. The proof that bisimulation is a congruence extended Howe’s method, where two points emerged to be of significance.

First, we have seen that testing terms by just reducing them to weak head normal form and applying these WHNF’s to arbitrary arguments is not appropriate. Instead, the terms to be tested have rather be equipped with all the information about which choices have to be shared and which may be copied. We accomplished this by performing evaluation inside surface contexts up to every arbitrary depth, in which also choices in \texttt{let}-environments may be forced. Since we non-deterministically collect all these possible outcomes, we therefore have enough potential to discriminate terms.

The other aspect concerns the kind of terms that may be copied. As we have seen, the precongruence candidate or, strictly speaking, the extension of the bisimilarity to open terms had to be adapted such that only \( \odot \) and abstractions are considered. This might point out a general way for the proof of the fundamental substitution lemma to go through, i.e. for \[11\], Lemma 1] and \[12, Lemma 3.2\] respectively, or lemma 4.13 and 4.14 in our case.

On the basis of these explanations, we feel confident that the technique demonstrated in this paper is powerful enough for the treatment of a language extending the \( \lambda_{\text{ND}} \)-calculus with a \texttt{case} and data constructors. It could also be worth to apply the results of this paper to the design and development of generic and purely syntactic systems of structural operational semantics, e.g. like the structured evaluation systems of \[12\] but suited for non-determinism combined with sharing.

As remarked earlier, the contextual equivalence does not regard must convergence nor, on a par with it, divergence. Like the work of \[21,13,31\] suggests,
it is quite reasonable in a non-deterministic calculus to regard possibly infinite reduction sequences. Hence as a further extension of the \(\lambda_{ND}\)-calculus, also divergent behaviour might be incorporated. So, writing \(s \uparrow\) if \(s\) has a non-terminating normal-order reduction, a possible — and sensible — definition of the contextual equivalence might be given by

\[
s \simeq_{c} t \iff ((\forall C : C[s] \Downarrow \iff C[t] \Downarrow) \land (\forall C : C[s] \Uparrow \iff C[t] \Uparrow))
\]

Using contextual approximation, the above contextual equivalence may be established in several ways. It may seem appealing to adopt a definition like

\[
s \preceq_{c} t \iff (\forall C : (C[s] \Downarrow \implies C[t] \Downarrow) \land (C[t] \Uparrow \implies C[s] \Uparrow))
\]

from [13] for the contextual approximation. But for our method, this will pose technical difficulties in showing that similarity implies contextual approximation. This is, because then e.g. \(K \preceq_{c} \text{pick } \Omega K\) will not hold anymore and therefore \(s \overset{\approx}{\longrightarrow}_{\lambda} t \implies t \preceq_{b} s\) neither. We preferably would like to retain this property, since it has turned out to be extremely helpful in the proof. It appears to us that, by the duality of convergence and divergence, it is feasible to define a separate “approximation” relation for divergence. For that relation a method similar to the one presented in this paper seems possible.

There is another aspect concerning the omission of divergence: As we have indicated before, there are equalities in the \(\lambda_{ND}\)-calculus which are not true in a calculus regarding divergence, e.g. [13]. These include the following equivalences, where \(\bot\) stands for an arbitrary term which does not have a weak head normal form:

\[
\begin{align*}
\text{pick } s \bot & \simeq_{\lambda_{ND}, c} s \\
\text{pick } \bot t & \simeq_{\lambda_{ND}, c} t
\end{align*}
\]

So in the \(\lambda_{ND}\)-calculus the operator \(\text{pick}\) behaves bottom-avoiding which suggests that it could be worthwhile to apply our results to this kind of calculi.

Further enhancements may be devoted to making bisimulation proofs easier to handle. Since the approximation reduction in the \(\lambda_{\approx}\)-calculus is highly non-deterministic, a direct definition of bisimulation in \(\lambda_{ND}\) is desirable, which provides more information on how to proceed comparing two terms.

Moreover, because sharing does not change the termination behaviour of terms in a deterministic setting, an application of our results to the improvement theory of [20], where terms could be distinguished if they differ in the number of reductions necessary to reach a weak head normal form, may be of interest for future research.
8 Acknowledgements

I would like to express my gratitude to Manfred Schmidt-Schauß and David Sabel for all their valuable comments and constructive criticism. I am particularly indebted to Manfred Schmidt-Schauß for scrutinising several proofs and having directed my attention to non-deterministic call-by-need \(\lambda\)-calculi.

References


