A Non-Deterministic Call-by-Need Lambda Calculus

Arne Kutzner, Manfred Schmidt-Schauß

Fachbereich Informatik
Johann Wolfgang Goethe-Universität
Postfach 11 19 32
D-60054 Frankfurt, Germany
email: {schauss,arne}@ki.cs.uni-frankfurt.de
Tel: (+49) 69 798 28597; Fax: (+49) 69 798 28919

Abstract

In this paper we present a non-deterministic call-by-need (untyped) lambda calculus λ_{nd} with a constant `choice` and a `let`-syntax that models sharing. Our main result is that λ_{nd} has the nice operational properties of the standard lambda calculus: confluence on sets of expressions, and normal order reduction is sufficient to reach head normal form. Using a strong contextual equivalence we show correctness of several program transformations. In particular of lambda-lifting using deterministic maximal free expressions. These results show that λ_{nd} is a new and also natural combination of non-determinism and lambda-calculus, which has a lot of opportunities for parallel evaluation.
An intended application of λ_{nd} is as a foundation for compiling lazy functional programming languages with I/O based on direct calls. The set of correct program transformations can be rigorously distinguished from non-correct ones. All program transformations are permitted with the slight exception that for transformations like common subexpression elimination and lambda-lifting with maximal free expressions the involved subexpressions have to be deterministic ones.

1 Introduction

Currently, the preferred methods in non-strict functional languages to implement I/O and other interactions with the environment are monadic programming as in Haskell ([PHA+97]) or direct calls that are embedded in a system of unique types as in Clean [NSvP91, Ach96]. In the commercial non-strict functional programming language Natural EL [HNSSH97] these interactions were implemented as direct calls.
The intention of this paper is twofold: On the one hand a non-deterministic lambda calculus λ_{nd} is described that is different from most other non-deterministic lambda calculi insofar as it is lazy, i.e., call-by-need, and has all the advantageous properties of a lambda-calculus, where instead of confluence a generalized notion (set-confluence) is used. On the other hand we want to demonstrate that the naïve approach to I/O in non-strict functional languages can be justified, based on a different set of transformations of the non-deterministic lambda-calculus.
The following well-known example demonstrates the problems in adding non-determinism to the lambda calculus: Let $\|$ be non-deterministic choice and let the function `double` be defined as `double` $x = x + x$. The issue is: "What is the result of reducing the expression `double` $(1 \| 2)$?" Using β-reduction there are two different reduction sequences that conflict with set-confluence.

- `double` $(1 \| 2) \longrightarrow (1 \| 2) + (1 \| 2)$, which may result in 2, 3, or 4.
- `double` $(1 \| 2) \longrightarrow$ `double` n, where $n = 1 \vee 2$, which may result in 2 or 4.

This means that depending on the selected redex to be reduced, the set of possible results is different after one reduction step, which means that this kind of beta-reduction makes an "implicit choice". This is not only counterintuitive, but leads to inconsistencies. One remedy is to restrict the permitted redexes, usually by only permitting a fixed reduction strategy like strict evaluation or normal order evaluation.
The first possibility is the method chosen for strict functional programming languages and also for Clean. It has the disadvantage that it severely restricts the permitted program transformations, since the sequence of evaluation is highly fixed.
Another remedy, which we will pursue, is to modify the calculus by introducing sharing such that the calculus does not unnecessarily copy expressions. In the example above, the expression $(1 \| 2)$ is not copied, such that the set of possible results is 2 or 4. We are strongly influenced by the let-calculi described in [AFM+95, AF97, MOW98] which model sharing. The only syntactic addition is to add `choice` as a constant.
In order to be able to built a compiler using program transformations like lambda-lifting, partial evaluation, inlining, etc., it is important that the calculus is rather permissive in the applicability of its reduction rules, such that a wide range of program transformations can be shown to be correct. The sequence of evaluations should have a maximal degree of freedom, such that parallel evaluation is possible. On the other hand, the normal order reduction should be rather close to a possible implementation. The calculus λ_{nd} meets these requirements.

ICFP '98 Baltimore, MD USA

There are two criteria that play a role in the correctness proofs: One is set-confluence, the other is the existence of (in)finite normal order reductions. Set-confluence corresponds to the intuition that the initial program determines the set of possible results, and that exactly these results can be computed using reduction rules. Moreover, it also captures the intuition that all choices are explicit. Thus it is a natural generalization of confluence.

The achievements of this paper are that the non-deterministic call-by-need lambda calculus λ_{nd} is set-confluent (Theorem 3.8) and that whenever an expression can be reduced to a HNF, there is a normal order reduction to a HNF (Theorem 4.9). We show correctness of lambda-lifting with maximal free expressions wrt contextual equivalence, a result which seems of practical value (Theorem 8.2). As a corollary it follows by simply dropping the choice, that the calculus λ_{let} [AFM+95, AF97] may have more permissive reduction rules without destroying confluence.

In [SS92] twelve different kinds of non-determinism in non-strict functional languages are identified. Using their categories, λ_{nd} has an erratic, restrained, singular non-determinism. However, λ_{nd} justifies unfolding, (i.e., program transformations) without incompatibilities between singular semantics and unfoldability. Hence, λ_{nd} is a new kind of non-determinism in non-strict functional languages. Unfortunately, it is a real pain to work through case-analyses and reductions of lots of diagrams. Due to lack of space, we often give only the results and omit the tedious computations. The diagrams are also mechanically checked (by a Haskell-program) up to a certain size of expressions. This forces us to correct several errors in the diagrams. An extended version of this paper with full proofs is in preparation.

Our calculus λ_{nd} fits into the work on implementing and compiling non-strict functional programming languages [PJ87, PJL91], since the calculus is compatible with super-combinator reduction.

The practical application of our calculus is to provide a basis for program transformations (i.e. compilation) of lazy functional (higher order) programming languages that use direct calls. The correct program transformations can be derived by simulating the result of an I/O-function by a big choice-expression. For example, the result of an I/O-function that may return $True$ or $False$ can be modeled by the expression (choice$\bot$(choice $True$ $False$)).

The paper is structured as follows. First the calculus is described in section 2. Then the basic properties like set-confluence and existence of a normal order reduction are investigated in sections 3 and 4. Contextual equivalence and deterministic expressions are defined in sections 5 and 7. The correctness of a set of extended rules and of lambda-lifting is proved in sections 6, 7 and 8, and the relationship with the usual lambda calculus is clarified in section 9.

2 The Language

Λ-expressions may be variables x, the constant choice, applications $(s\ t)$, let expressions $(\texttt{let}\ x = s\ \texttt{in}\ t)$ and abstractions $\lambda x.t$, where x is a variable, and s, t are Λ-expressions. As a convention we shall assume that all bound variables are different, which can be achieved by a consistent renaming of bound variables. The set of variables in a closed expression t is denoted as $V(t)$, the set of let-bound variables as $V_{let}(t)$. A *closed* expression is one without free variables. We consider expressions as equal (denoted $\equiv$), if there is some consistent renaming of bound variables that makes them equal, i.e., if they are α-convertible. We use the convention that application is left associating, i.e., $e_1\ e_2\ e_3$ means the expression $((e_1\ e_2)\ e_3)$.

The constant choice is intended to be a function that may reduce (choice s t) to either s or t. We use several kinds of reduction relations. We shall use the $+, *$-notation for the transitive and the transitive-reflexive closure. The symbol "?" is used for reduction consisting of 0 or 1 steps. A context $C[.]$ is a closed Λ-expression with exactly one hole, where the hole can be at every position where an expression is permitted, i.e., the syntax is $C ::= [\cdot] \mid C\ s \mid s\ C \mid (\texttt{let}\ x = C\ \texttt{in}\ s) \mid (\texttt{let}\ x = S\ \texttt{in}\ s) \mid \lambda x.C$, where s is a Λ-expression and x a variable. The notation $C[s]$ stands for the expression, where s is plugged into the hole. A reduction rule $\longrightarrow$ is compatible, iff $t \longrightarrow t'$ implies $C[t] \longrightarrow C[t']$ for every context $C[\cdot]$.

2.1 Non-Deterministic Reduction

We use the reduction rules in table 1, which are a generalization of the ones in [AFM+95, AF97]. The calculus is almost the same as the one in [MOW98], but we leave out the garbage collection rule (ldel), see 2.

The rule (nd) is not compatible, which is justified by the intuition that a compiler should not be able to "optimize" functions by evaluate I/O's at compile time. It could be made compatible by formulating the reduction rules on sets of expressions. The presented formulation is close to the operational view that the choice-decisions are taken outside the functional program, and that once a decision is made, you can forget the alternatives (committed choice).

Definition 2.1 *Let s denote an expression. A* reduction context $R[.]$ *is defined by the syntax:* $R ::= [\cdot] \mid (R\ s) \mid (s\ R) \mid (\texttt{let}\ x = s\ \texttt{in}\ R) \mid (\texttt{let}\ x = R\ \texttt{in}\ s)$. *A* let-context *is defined by the syntax:* $L ::= [\cdot] \mid (\texttt{let}\ x = s\ \texttt{in}\ L)$. *We will use the symbols R, L only with this meaning. Moreover, let* $L_L ::= (\texttt{let}\ x = \cdot\ \texttt{in}\ s)$, $A_L ::= \cdot\ s$, *and* $W ::= L \mid L[L_L[A_L^*]]$

Definition 2.2 *Based on the reductions in table 1, we define:* $s \xrightarrow{let} t$, *if* $s \xrightarrow{\rho} t$ *for* $\rho \in \{llet, lbeta, lapp\}$. $s \xrightarrow{loa} t$, *iff* $s \xrightarrow{llet} t$ *or* $s \xrightarrow{lapp} t$.

Definition 2.3 *We define relations on sets of closed expressions. Let s, t be closed expressions and let M be a set of closed expressions.*

- *If* $s \xrightarrow{\rho} t$, *then* $M \cup \{s\} \xrightarrow{set,\rho} M \cup \{t\}$ *for all labels* $\rho \in \{cp, llet, lapp, lbeta\}$.
- $M \cup \{R[(\texttt{choice}\ s)]\} \xrightarrow{set,nd} M \cup \{R[((\lambda\ x, y\ .\ y)\ s)], R[((\lambda\ x, y\ .\ x)\ s)]\}$.
- $\xrightarrow{set,let,nd} := \xrightarrow{set,let} \cup \xrightarrow{set,nd}$.
- $\xrightarrow{set}_{\lambda,nd} := \xrightarrow{set,let,nd} \cup \xrightarrow{set,cp}$.

We define redexes as immediately reducible subexpressions within an expression.

Let C, D be arbitrary contexts.

(llet)	$C[(\texttt{let } x = (\texttt{let } y = t_y \texttt{ in } t_x) \texttt{ in } s)]$	$\longrightarrow$	$C[(\texttt{let } y = t_y \texttt{ in } (\texttt{let } x = t_x \texttt{ in } s))]$
(lapp)	$C[((\texttt{let } x = t_x \texttt{ in } s)\ t)]$	$\longrightarrow$	$C[(\texttt{let } x = t_x \texttt{ in } (s\ t))]$
(lbeta)	$C((\lambda\ x\ .\ t)\ s)]$	$\longrightarrow$	$C[(\texttt{let } x = s \texttt{ in } t)]$
(nd)	$R[(\texttt{choice } s)]$	$\longrightarrow$	$R[((\lambda\ x\ .\ (\lambda\ y\ .\ y))\ s)]$ where x, y are fresh variables.
		$\longrightarrow$	$R[((\lambda\ x\ .\ (\lambda\ y\ .\ x))\ s)]$ where x, y are fresh variables.
(cp)	$C[(\texttt{let } x = s \texttt{ in } D[x])]$	$\longrightarrow$	$C[(\texttt{let } x = s \texttt{ in } D[s'])]$

If s is a lambda-abstraction or choice and s' is a renamed copy of s

Table 1: The non-deterministic calculus λ_{nd}

3 Confluence of Reduction on Sets

Usually, confluence of reduction is interpreted as the independence of the result of a computation from the sequence of reductions. For a non-deterministic reduction, this notion has to be generalized. For example, choice s t may be reduced to s as well as t. Since the expressions s, t are arbitrary, we cannot hope that there is any sensible relationship between s and t. The criterion that we will use instead is that the set of possible results is an invariant of reduction: For non-nd reductions this means that the set of possible results does not change, i.e., there will be no implicit choices. For an nd-reduction $t \xrightarrow{nd} t_1$, where the other alternative is $t \xrightarrow{nd} t_2$, we require that the set of possible results of t is the same as the union of the possible results of t_1, t_2.

3.1 Confluence of $\xrightarrow{let}$

Definition 3.1 *A relation $\longrightarrow$ is* locally confluent, *iff whenever $a \longrightarrow b$ and $a \longrightarrow c$ for some closed expressions a, there is some d, such that $b \xrightarrow{*} d$, and $c \xrightarrow{*} d$.*

Lemma 3.2 *The reduction relations $\xrightarrow{let}$ and $\xrightarrow{set,let,nd}$ are terminating.*

Proof. The following measure $\varphi = (\varphi_1, \varphi_2)$ is used for expressions, where pairs are compared lexicographically. The first component φ_1 is the number of occurrences of choice, and the second is defined as follows:

$$\varphi_2(s) := \begin{cases} 1 & \text{if } s \text{ is a variable or choice} \\ 2 * (\varphi_2(t) + \varphi_2(r)) & \text{if } s \equiv (t\ r) \\ \varphi_2(t) & \text{if } s \equiv (\lambda\ x\ .\ t) \\ 2 * \varphi_2(r) + \varphi_2(t) & \text{if } s \equiv (\texttt{let } x = r \texttt{ in } t) \end{cases}$$

It is an easy task to check that all the reductions $\xrightarrow{\rho}$ for $\rho \in \{llet, lapp, lbeta, nd\}$ strictly reduce the measure. It is easy to extend this to sets using well-foundedness of the corresponding multiset ordering. □

Lemma 3.3 *The reduction relations $\xrightarrow{let}$ and $\xrightarrow{set,let,nd}$ are locally confluent.*

Proof. For local confluence of $\xrightarrow{let}$, we have to check 5 nontrivial overlappings. We show only for the overlap of the llet-rule with itself that there is a common reduct:
$(\texttt{let } x = (\texttt{let } y = (\texttt{let } z = t_z \texttt{ in } t_y) \texttt{ in } t_x) \texttt{ in } s)$ reduces either to $(\texttt{let } y = (\texttt{let } z = t_z \texttt{ in } t_y) \texttt{ in } (\texttt{let } x = t_x \texttt{ in } s))$ or to $(\texttt{let } x = (\texttt{let } z = t_z \texttt{ in } (\texttt{let } y = t_y \texttt{ in } t_x)) \texttt{ in } s)$. The first expression reduces in one further let-reduction to $(\texttt{let } z = t_z \texttt{ in } (\texttt{let } y = t_y \texttt{ in } (\texttt{let } x = t_x \texttt{ in } s)))$. The second expression reduces as follows: $(\texttt{let } z = t_z \texttt{ in } (\texttt{let } x = (\texttt{let } y = t_y \texttt{ in } t_x) \texttt{ in } s))$, which further reduces to $(\texttt{let } z = t_z \texttt{ in } (\texttt{let } y = t_y \texttt{ in } (\texttt{let } x = t_x \texttt{ in } s)))$. The other computations are similar.
To check that $\xrightarrow{set,let,nd}$ is locally confluent, there are no extra nontrivial overlappings. The only nontrivial argument is that let-reductions cannot move a choice out of a reduction context. □

Proposition 3.4 *The relations $\xrightarrow{let}$ and $\xrightarrow{set,let,nd}$ are confluent.*

Proof. The Newman-lemma [New42] (see also [Hue80]) shows confluence of the relation $\xrightarrow{let}$ and $\xrightarrow{set,let,nd}$ using lemmas 3.3 and 3.2. □

3.2 Confluence of $\xrightarrow{cp}$

The same technique as in subsection 3.1 is used to show confluence of $\xrightarrow{cp}$.

Definition 3.5 *The measure ψ is defined using an environment parameter. For closed expressions let $\psi(s) := [\![s]\!]_\psi\ \emptyset$.*

$$\begin{array}{lcl} [\![x]\!]_\psi \rho & := & \rho(x) \\ [\![\texttt{choice}]\!]_\psi \rho & := & 1 \\ [\![(\lambda\ x\ .\ t)]\!]_\psi \rho & := & [\![t]\!]_\psi \rho[x \mapsto 0] \\ [\![s\ t]\!]_\psi \rho & := & [\![s]\!]_\psi \rho + [\![t]\!]_\psi \rho \\ [\![(\texttt{let } x = s \texttt{ in } t)]\!]_\psi \rho & := & [\![s]\!]_\psi \rho + [\![t]\!]_\psi \rho' \\ & & \text{where } \rho' := \rho[x \mapsto [\![s]\!]_\psi \rho + 1] \end{array}$$

Lemma 3.6 *The relations $\xrightarrow{cp}$ and $\xrightarrow{set,cp}$ are terminating.*

Proof. The measure is strictly monotone in the measure of subexpressions. $\xrightarrow{cp}$ strictly reduces the measure ψ. Since the measure is well-founded, $\xrightarrow{cp}$ terminates. A consequence is that $\xrightarrow{set,cp}$ terminates. □

Proposition 3.7 *The relation $\xrightarrow{set,cp}$ is locally confluent.*

3.3 Confluence

Theorem 3.8 *The relation $\xrightarrow{set}_{\lambda,nd}$ is confluent*

Proof. We use Lemma 3.3.6 in [Bar84] which states the following: If for all a, b, c: if $a \to_1 b, a \to_2 c$ there exists some d, such that $b \xrightarrow{*}_2 d, c \xrightarrow{\leq 1}_1 d$, then $\xrightarrow{*}_1$ and $\xrightarrow{*}_2$ commute. We use this lemma with $\to_1 := \xrightarrow{set,cp}, \to_2 := \xrightarrow{set,let,nd}$. We have to check the nontrivial overlappings and to show that the forking reductions can be joined according to the restrictions. I.e., we have to check the situations $t_1 \to t_2, t_1 \to t_3$ using the syntactical structure of expressions.

- Case: one reduction is an nd-reduction. Then there is no problem in joining, since a cp-reduction cannot duplicate the nd-redex.
- Case: One reduction is a let-reduction. Then one cp-reduction and 1 or 2 let-reductions are sufficient to join the reduction. The latter case is possible if the let-reduction is within an abstraction.

The relations $\xrightarrow{set,let,nd,*}$ and $\xrightarrow{set,cp,*}$ commute. The propositions 3.4 and 3.7 and the commutation property now show that the relation $\xrightarrow{set,*}_{\lambda,nd}$ is confluent. □

Example 3.9 *The reduction $\longrightarrow_{\lambda,nd}$ would become non-set-confluent, if we would permit the* choice*-reduction in the body of abstractions: Consider the expression* (let $x = (\lambda\ y\ .$ choice 1 2) in $(x\ 0)+(x\ 0)$), *which results in* $\{2,3,4\}$ *after the correct reductions. The wrong choice-reduction would give* $\{$(let $x = (\lambda\ y\ .\ 1)$ in $(x\ 0) + (x\ 0)$), (let $x = (\lambda\ y\ .\ 2)$ in $(x\ 0) + (x\ 0)$)$\}$, *which results in* $\{2,4\}$.

4 Normal order reduction sequences

In this section we define normal order reduction and show that normal order reduction is sufficient to reduce expressions to HNF. This definition models the normal order redex as an outermost redex that is demanded. This is a slightly more lazy variant of the normal order definition in [MOW98].

Definition 4.1 *A* normal order redex (n-o-redex) *of a closed expression t is defined using rules for shifting a label E (for evaluation) up and down in the expression to a final position, thereby leaving as trace a label e and also a compound label describing the n-o-reduction. We start with t^E, where t is unlabelled.*

i) $C[s^E]$, *and s is an abstraction or the constant* choice. *Stop, the expression t is a HNF.*

ii) $C[(r\ s)^E]$ *and r is an abstraction. Mark the expression in the brackets as (lbeta) and return.*

iii) $C[(\texttt{choice}\ s)^E]$. *Mark the expression in the bracket as (nd) and return.*

iv) $C[(r\ s)^E]$, *and r is a variable or an application. Proceed with* $C[(r^E\ s)^e]$.

v) $C[(r\ s)^E]$, *and r is a let-expression. Mark the expression in the bracket as (lapp) and return.*

vi) $C[(\texttt{let}\ x = s\ \texttt{in}\ t)^E]$. *Proceed with* $C[(\texttt{let}\ x = s\ \texttt{in}\ t^E)^e]$.

vii) $C[(\texttt{let}\ x = (\texttt{let}\ y = t_y\ \texttt{in}\ t_x)\ \texttt{in}\ D[x^E])]$. *Mark the expression in the bracket as (llet) and return.*

viii) $C[(\texttt{let}\ x = r\ \texttt{in}\ D[x^E])]$ *and r is an abstraction or the constant* choice. *Mark the expression in the* [.]*-bracket as (cp) and also with the context $D[.]$, and return.*

ix) $C[(\texttt{let}\ x = r\ \texttt{in}\ D[x^E])]$ *and r is an application or a variable. Proceed labeling with* $C[(\texttt{let}\ x = r^E\ \texttt{in}\ D[x^e])]$

The subexpression that is finally labelled E is called the n-o-redex *if it is not a lambda-abstraction or* choice, *otherwise the whole expression t is a* head normal form (HNF).
The normal order reduction has to be performed such that the rule corresponding to the label is executed. The rule (cp) has to be performed such that the variable labelled E is replaced at the position indicated by the context $D[.]$.
A reduction sequence that reduces only n-o-redexes is called a normal order reduction.

Lemma 4.2 *The following holds for the labeling algorithm.*

- *The labeling terminates and either marks a unique n-o-redex or marks the whole expression as a HNF.*
- *Every superexpression of an n-o-redex is marked* e *in the labeling.*
- *An n-o-redex may only be subexpression of another redex of type cp or llet, but not in the expression to be copied by the cp-rule. It is also not a subexpression of another redex of type lapp, lbeta, or nd.*
- *Any n-o-redex and also the variable to be replaced is in the context W.*
- *An n-o-redex of type (cp) or (llet) is only possible in a let-context $L[\cdot]$.*
- *A HNF is of the form $L[(\lambda\ y\ .\ t)]$ or L*[choice] .

Lemma 4.3 *Let t be a HNF. If $t \longrightarrow t'$ then t' is a HNF.*

Lemma 4.4 *Let t be a closed expression with an n-o-redex. If $t \longrightarrow t'$, by a non-n-o-reduction, then t' is not a HNF.*

Proof. If the redex in $t \longrightarrow t'$ is not labelled e, this is obvious. Otherwise, this redex is marked e. It is an easy exercise to check the cases where the reduction is of type llet or cp. □

Corollary 4.5 *The last reduction before reaching a HNF is a normal order reduction of type (cp) or (lbeta).*

In the following we show that an arbitrary reduction to a HNF can be turned into an n-o-reduction by commuting the reductions. A non-n-o-reduction is also denoted as i(nternal) reduction. In order to ease notation, we denote a sequence of reductions as words: $\xrightarrow{a,\rho}$ is denoted as (a,ρ) for all types of rules and for $a \in \{no, i\}$. For example $\xrightarrow{no,nd} \circ \xrightarrow{i,cp}$ is denoted as $(no,nd) \circ (i,cp)$. We will use meta-reductions on sequences of reductions.

Definition 4.6 *Let t be a closed expression. The reduction $t \xrightarrow{cppar} t'$ is defined as follows: For $x, y \in V_{let}(t)$ let $x <\cdot y$, iff* (let $y = t_y$ in s) *is a subexpression of t and x occurs in t_y, and let $<$ be the transitive closure of $<\cdot$.*
Select an antichain $W \subseteq V_{let}(t)$, i.e. the variables in W are not related by $<$. Then define the relation $\xrightarrow{cppar}_W$ for subexpressions s of t as follows:

- $x \xrightarrow{cppar}_W x$
- $x \xrightarrow{cppar}_W t'_x$ *if* $x \in W$ *and* x *is let-bound in* t *to the abstraction* t_x, *and* t'_x *is a renamed version of* t_x.
- $\texttt{choice} \xrightarrow{cppar}_W \texttt{choice}$
- *If* $(s_1\ s_2)$ *is a subexpression of* t, *and* $s_1 \xrightarrow{cppar}_W s'_1$, $s_2 \xrightarrow{cppar}_W s'_2$, *then* $s_1\ s_2 \xrightarrow{cppar}_W s'_1\ s'_2$.
- *If* $(\lambda\ z\ .\ s_1)$ *is a subexpression of* t, *and* $s_1 \xrightarrow{cppar}_W s'_1$, *then* $(\lambda\ z\ .\ s_1) \xrightarrow{cppar}_W (\lambda\ z\ .\ s'_1)$.
- *Let* $(\texttt{let}\ z = t_z\ \texttt{in}\ s_z)$ *be a subexpression of* t. *If* $s_z \xrightarrow{cppar}_W s'_z$, $t_z \xrightarrow{cppar}_W t'_z$, *then* $(\texttt{let}\ z = t_z\ \texttt{in}\ s_z) \xrightarrow{cppar}_W (\texttt{let}\ z = t'_z\ \texttt{in}\ s'_z)$. *Note that if* $z \in W$ *then* $s_z = s'_z$.

Let $t \xrightarrow{cppar} t'$ *iff* $t \xrightarrow{cppar}_W t'$ *for some antichain* W.

Lemma 4.7 *If* $t \xrightarrow{cppar} t'$, *then* $t \xrightarrow{cp,*} t'$, *where the reductions can be performed in any order. Moreover, there is at most one n-o-reduction among them. This n-o-reduction can be shifted to the left.*

We give an explanation of the notation in the following lemma: $(i,a) \circ (no,b) \rightsquigarrow (no,b) \circ (i,a)$ means $\forall t_1,t_2,t_3 : t_1 \xrightarrow{i,a} t_2 \xrightarrow{no,b} t_3 \Rightarrow \exists t_4 : t_1 \xrightarrow{no,b} t_4 \xrightarrow{i,a} t_3$.

Lemma 4.8 *Every i-reduction followed by normal order reductions can be shifted to the right according to one of the following rules:*

- $(i,a) \circ (no,b) \rightsquigarrow (no,b) \circ (i,a)$.
- $(i,a) \circ (no,b) \rightsquigarrow (no,b) \circ (no,a)$ *for* $a \neq cppar$.
- $(i,a) \circ (no,cp) \rightsquigarrow (no,cp) \circ (no,a)^? \circ (i,a)^*$, *for* $a \neq nd, cppar$.
- $(i,cppar) \circ (no,a) \rightsquigarrow (no,a) \circ (no,cp)^* \circ (i,cppar)^?$.
- $(i,llet) \circ (no,lapp)^w \circ (no,llet)^? \circ (no,lapp)^w \circ (no,llet)^? \rightsquigarrow (no,lapp)^w \circ (no,llet)^? \circ (a,llet)$ *where* $(a \in \{i,no\})$ *and* $w \geq 0$.

Theorem 4.9 *Let* t *be a closed expression. If* $t \xrightarrow{*} t'$ *where* t' *is a HNF, then there is a HNF* t'', *such that* $t \xrightarrow{no,*} t''$ *and* $t'' \xrightarrow{*} t'$.

Proof. We use the previous lemmas. Let $t \xrightarrow{*} t'$ be a reduction. We use as meta-reduction strategy to shift the rightmost i-reduction to the right dropping the i-reductions after reaching a HNF. The following well-founded ordering shows termination of this meta-reduction, since it is strictly decreased in every meta-reduction.
It is a lexicographic ordering of four components. The first component is the number of $(i,cppar)$-reductions. The second is a multiset of the following numbers: For every i-reduction: the number of (no,cp) reductions that are right of it. The third component is the total number of internal reductions. The fourth component is the number of n-o-reductions to HNF right of the rightmost i-reduction.
It is easy to see that this measure is well-founded and that every meta-reduction on the reductions to HNF strictly decreases this size. Hence the meta-reduction will end with a reduction that is a sequence of n-o-reductions followed by a sequence of i-reductions to a HNF. The i-reductions retain the property of being a HNF, hence the theorem holds. □

Corollary 4.10 *If an expression can be reduced to HNF, then an n-o-reduction sequence has the smallest number of nd-reduction steps.*

Corollary 4.11 *Every infinite n-o-reduction contains an infinite number of cp-reductions and an infinite number of beta-reductions*

Proof. The only rule that may increase the measure ψ is the rule lbeta. On the other hand, the only rule that can increase the measure φ is cp. □

5 Contextual Equivalence

In order to prove correctness of optimized lambda-lifting and to clarify the connection with the deterministic lambda calculus, the criterion of contextual equivalence [Abr90, How89] is required. We will use a rather strong criterion including termination as well as non-termination.

Definition 5.1 *Let* s,t *be (open) expressions. Then* $s \sim_c t$, *iff the following holds*

i) For all contexts $C[\cdot]$, *such that* $C[s], C[t]$ *is closed:* $C[s]$ *has a reduction to HNF iff* $C[t]$ *has a reduction to HNF.*

ii) For all contexts $C[\cdot]$, *such that* $C[s], C[t]$ *is closed:* $C[s]$ *has an infinite n-o-reduction, iff* $C[t]$ *has an infinite n-o-reduction.*

Note that $\sim_c$ is a congruence, which can be proved straightforwardly.
To justify the second requirement, consider for example the two expressions 0 and $Y(\lambda\ x\ .\ \texttt{choice}\ x\ 0)$, where Y is the usual fixpoint combinator. Clearly, the two expressions behave differently, and are also different using our definition, but would be equivalent without the requirement ii).
We define a relation similar to a parallel reduction in [Bar84].

Definition 5.2 *Let* t *be a closed expression. The relation* $\xrightarrow{par}$ *is defined as follows: First select a set* $W \subseteq V_{let}(t)$, *which is an antichain w.r.t. to the ordering defined in definition 4.6. Then consider the subexpressions of* t *and define the relation* $\xrightarrow{par}_W$.

- $\texttt{choice} \xrightarrow{par}_W \texttt{choice}$
- $x \xrightarrow{par}_W x$
- *(app) If* $s \xrightarrow{par}_W s'$, $r \xrightarrow{par}_W r'$, *then* $(s\ r) \xrightarrow{par}_W (s'\ r')$.
- *(lam) If* $s \xrightarrow{par}_W s'$, *then* $(\lambda\ x\ .\ s) \xrightarrow{par}_W (\lambda\ x\ .\ s')$.
- *(let) If* $r \xrightarrow{par}_W r'$, $s \xrightarrow{par}_W s'$, *then* $(\texttt{let}\ x = r_x\ \texttt{in}\ s_x) \xrightarrow{par}_W (\texttt{let}\ x = r'_x\ \texttt{in}\ s'_x)$. *If* $x \in W$, *then* $r_x \equiv r'_x$.

- *(cp)* $x \xrightarrow{par}_W t'_x$ *if* $x \in W$ *and* x *is let-bound to the abstraction* t_x *in* t *and* t'_x *is a renamed version of* t_x.
- *(lapp) if* $t_x \xrightarrow{par}_W t'_x$, $s_x \xrightarrow{par}_W s'_x$, $r \xrightarrow{par}_W r'$, *then* $((\texttt{let } x = t_x \texttt{ in } s_x)\ r) \xrightarrow{par}_W (\texttt{let } x = t'_x \texttt{ in } s'_x\ r')$. *If* $x \in W$, *then* $t_x \equiv t'_x$.
- *(lbeta) If* $t_x \xrightarrow{par}_W t'_x$, $s \xrightarrow{par}_W s'$, *then* $((\lambda\ x\ .\ t_x)s) \xrightarrow{par}_W ((\texttt{let } x = s' \texttt{ in } t'_x)$
- *(llet) If* $t_x \xrightarrow{par}_W t'_x$, $t_y \xrightarrow{par}_W t'_y$, *and* $s \xrightarrow{par}_W s'$, *then* $(\texttt{let } x = (\texttt{let } y = t_y \texttt{ in } t_x) \texttt{ in } s) \xrightarrow{par}_W (\texttt{let } y = t'_y \texttt{ in } (\texttt{let } x = t'_x \texttt{ in } s'))$. *If* $y \in W$, *then* $t_y \equiv t'_y$.

Let $t \xrightarrow{par} t'$ *iff* $t \xrightarrow{par}_W t'$ *for some antichain.*

Note that an (nd)-reduction is not permitted in the $\xrightarrow{par}$-relation.

Lemma 5.3 *If* $t \xrightarrow{par} t'$, *then there is a sequence of reductions (using the basic calculus) from* $t \to t'$, *such that the reductions can be performed in any order. It is possible to arrange the sequence, such that all normal order reductions come first.*

Proof. That the reductions can be performed in any order follows from the definition. The normal-order reductions can be shifted to the left using Lemma 4.8. □

In the following we use the wording *complete set of forking (commutation) diagrams* for a relation $\xrightarrow{\rho}$. A complete forking (commutation) diagrams means a set of meta-reduction rules, such that for every maximal n-o-reduction sequence red, the prefix of every reduction sequence $\xleftarrow{\rho} ored$ ($\xrightarrow{\rho} ored$) can be meta-reduced. Note that these complete sets are not unique.

Lemma 5.4 *In the following we consider only par-reductions consisting of internal reductions.*
A complete set of forking diagrams for par is:

i) $\xleftarrow{no,a} \circ \xrightarrow{i,par} \rightsquigarrow \xrightarrow{i,par} \circ \xleftarrow{no,a}$

ii) $\xleftarrow{no,a,*} \circ \xleftarrow{no,b} \circ \xrightarrow{i,par} \rightsquigarrow \xrightarrow{i,par,?} \circ \xleftarrow{no,b}$ *where* $a \neq nd$.

iii) $\xleftarrow{no,llet,+} \circ \xrightarrow{i,par} \rightsquigarrow \xrightarrow{i,par,?} \circ \xleftarrow{no,llet,+}$.

iv) $\xleftarrow{no,loa,*} \circ \xleftarrow{no,lapp} \circ \xrightarrow{i,par} \rightsquigarrow \xrightarrow{i,par,?} \circ \xleftarrow{no,loa,*} \circ \xleftarrow{no,lapp}$.

Proposition 5.5 *Let* t, t' *be closed expressions such that* $t \xrightarrow{par} t'$. *Then* $t \sim_c t'$:

Proof. Note that it is not necessary to use the context $C[.]$, since the par-reduction does not contain an nd-reduction, and hence has no restrictions of applications in an expression.
If t' has a HNF, then t prime has an n-o-reduction to HNF, which follows from Theorem 4.9. If t' has an infinite n-o-reduction, then the diagrams in Lemma 4.8 and corollary 4.11 show that there is also an infinite n-o-reduction for t.
Now assume that there is an n-o-reduction for t ending with a HNF. The diagrams in Lemma 5.4 show that by induction on the length of an n-o-reduction of t, we get also an n-o-reduction for t'. If the n-o-reduction for t is infinite, then the diagrams show that we can shift the (i,par) down the n-o-reduction for t and that we can produce an infinite n-o-reduction for t'. □

Corollary 5.6 *Let* t, t' *be closed expressions such that* $t \to t'$ *by a non-nd reduction. Then* $t \sim_c t'$

Proof. Every one step non-nd-reduction is also a par-reduction. □

Note that this is not true for nd-reductions, since $(\texttt{choice } 0\ \bot) \xrightarrow{nd} \bot$, and $(\texttt{choice } 0\ \bot) \xrightarrow{nd} 0$, hence $(\texttt{choice } 0\ \bot)$ has a finite as well as an infinite n-o-reduction, but $\bot$ has only an infinite one. Thus $(\texttt{choice } 0\ \bot) \not\sim_c \bot$.

6 Behavioral Equivalence of the Rules ldel and lcv

In the following we give forking and commutation diagrams for the reductions in Table 2 in the same way and under the same conditions as in Lemmas 5.4 and 4.8. The rule (ldel) is the garbage collection rule, whereas (lcv) corresponds to an elimination of indirections.

Lemma 6.1 *A complete set of commutation diagrams for (ldel) is:*

- $ldel \circ (no, a) \rightsquigarrow (no, a) \circ ldel$.
- $ldel \circ (no, cp) \rightsquigarrow (no, cp) \circ ldel \circ ldel$.
- $ldel \circ (no, lapp)^{w+1} \circ (no, llet)^k \rightsquigarrow (no, lapp)^{w+1} \circ (no, llet)^k \circ (no, lapp)^{w+1} \circ (no, llet)^k \circ ldel$, *where* $w \geq 0$ *and* $k \in \{0, 1\}$.
- $ldel \circ (no, llet) \rightsquigarrow (no, llet) \circ (no, llet) \circ ldel$
- $ldel \circ (no, \rho) \rightsquigarrow (no, lapp) \circ (no, \rho) \circ ldel$, *where* $\rho \in \{nd, lbeta\}$

For (ldel) the forking diagrams are a sufficient set of meta-reductions that meta-reduce every reduction sequence of the form $t_1 \xrightarrow{ldel} t_2$ together with an n-o-reduction starting from t_1 into another sequence.

Lemma 6.2 *A complete set of forking diagrams for (ldel) is:*

- $\xleftarrow{no,a} \circ \xrightarrow{ldel} \rightsquigarrow \xrightarrow{ldel} \circ \xleftarrow{no,a}$
- $\xleftarrow{no,cp} \circ \xrightarrow{ldel} \rightsquigarrow \xrightarrow{ldel} \circ \xrightarrow{ldel} \circ \xleftarrow{no,cp}$.
- $\xleftarrow{no,lapp} \circ \xrightarrow{ldel} \rightsquigarrow \xrightarrow{ldel}$.
- $\xleftarrow{no,llet} \circ \xrightarrow{ldel} \rightsquigarrow \xrightarrow{ldel}$.

Theorem 6.3 *Let* t, t' *be closed expressions and* $t \xrightarrow{ldel} t'$. *Then* $t \sim_c t'$.

(ldel) $C[(\texttt{let } x = s \texttt{ in } t)] \xrightarrow{ldel} C[t]$
if t has no occurrences of x

(lcv) $C[(\texttt{let } x = y \texttt{ in } D[x])] \xrightarrow{lcv} C[(\texttt{let } x = y \texttt{ in } D[y])]$
where y is a variable

Table 2: Extended rules of λ^{+}_{nd}

Proof. Let $t \xrightarrow{ldel} t'$ and assume there is an n-o-reduction for t', finite or infinite. If the n-o-reduction is a finite one to a HNF, then use as a measure for meta-reductions the following lexicographical ordering of three components: i) the multiset of: for every ldel-reduction, the number of (no,cp) that are right of it. ii) The total number of ldel-reductions, iii) the number of n-o-reductions to HNF right of the rightmost ldel reduction. It is easy to see using the commutation diagrams for ldel, that this measure is strictly decreased if the rightmost ldel is shifted and an ldel for a HNF are eliminated.
If the reduction for t' is infinite, then the number of its (no,cp)-reduction steps is infinite. We show by induction on the number of (no,cp)-reductions in a normal order reduction sequence of t', that shifting ldel's to the right does not change this number. By using the strategy to shift all ldels over the first (no,cp), then all ldel's over the second and so forth, we get an infinite normal order reduction for t.
Now assume that there is an n-o-reduction for t. Now we have to use the forking diagrams in lemma 6.2. Using as main measure the number of (no,cp)-reductions in a sequence, it is easy to see that for a finite as well as for an infinite normal order reduction, we can construct a finite (infinite) normal order reduction for t'. □

Lemma 6.4 *A forking of an n-o-reduction and an internal cppar-reduction can be meta-reduced according to one the following rules.*

- $\overset{no,a}{\longleftarrow} \circ \overset{i,cppar}{\longrightarrow} \rightsquigarrow \overset{i,cppar}{\longrightarrow} \circ \overset{no,a}{\longleftarrow}$
- $\overset{no,cp,*}{\longleftarrow} \circ \overset{no,b}{\longleftarrow} \circ \overset{i,cppar}{\longrightarrow} \rightsquigarrow \overset{i,cppar,?}{\longrightarrow} \circ \overset{no,b}{\longleftarrow}$.

Proof. The main arguments are: an internal cp cannot copy an n-o-redex nor the into-position of an (no,cp)-redex. The second rule covers the case that there may be an normal-order reduction in the (i,cppar) after another n-o-reduction. □

Lemma 6.5 *A complete set of commutation diagrams for lcv is:*

- $lcv \circ (no, a) \rightsquigarrow (no, a) \circ lcv$.
- $lcv \circ (no, cp) \rightsquigarrow (no, cp) \circ lcv \circ lcv$
- $lcv \circ (no, cp) \rightsquigarrow (no, cp) \circ (a, cp) \circ \overset{i,cp}{\longleftarrow}$ *where* $a \in \{i, no\}$

Lemma 6.6 *A complete set of forking diagrams for lcv is:*

- $\overset{no,a}{\longleftarrow} \circ \overset{lcv}{\longrightarrow} \rightsquigarrow \overset{lcv}{\longrightarrow} \circ \overset{no,a}{\longleftarrow}$.
- $\overset{no,cp}{\longleftarrow} \circ \overset{lcv}{\longrightarrow} \rightsquigarrow \overset{lcv}{\longrightarrow} \circ \overset{lcv}{\longrightarrow} \circ \overset{no,cp}{\longleftarrow}$.
- $\overset{no,cp}{\longleftarrow} \circ \overset{no,cp}{\longleftarrow} \circ \overset{lcv}{\longrightarrow} \rightsquigarrow \overset{i,cp}{\longleftarrow} \circ \overset{no,cp}{\longleftarrow}$.
- $\overset{no,cp}{\longleftarrow} \circ \overset{lcv}{\longrightarrow} \rightsquigarrow \overset{i,cp}{\longrightarrow} \circ \overset{i,cp}{\longleftarrow} \circ \overset{no,cp}{\longleftarrow}$.

We require a special measure for lcv.

Definition 6.7 *The measure ξ is defined using an environment parameter. For closed expressions we define $\xi(s) := [\![s]\!]_\xi\, \emptyset$.*

$$
\begin{array}{lcl}
[\![x]\!]_\xi \rho & := & \rho(x) \\
[\![\texttt{choice}]\!]_\xi \rho & := & 1 \\
[\![(\lambda\ x\ .\ t)]\!]_\xi \rho & := & [\![t]\!]_\xi \rho[x \mapsto 0] \\
[\![s\ t]\!]_\xi \rho & := & 2 * [\![s]\!]_\xi \rho + [\![t]\!]_\xi \rho \\
[\![(\texttt{let } x = s \texttt{ in } t)]\!]_\xi \rho & := & [\![s]\!]_\xi \rho + [\![t]\!]_\xi \rho' \\
 & & \textit{where } \rho' := \rho[x \mapsto [\![s]\!]_\xi \rho + a] \\
 & & \textit{where } a = 0 \textit{ if } s \textit{ is a variable.} \\
 & & \textit{Otherwise: } a = 1.
\end{array}
$$

Lemma 6.8 *If $t \xrightarrow{\gamma} t'$ for $\gamma \in \{llet, lapp, cp, nd\}$, then $\xi(t) > \xi(t')$. If $t \xrightarrow{lcv} t'$, then $\xi(t) = \xi(t')$*

Proof. Evaluate the expressions before and after application of the rules and compare the measures. □

Theorem 6.9 *Let t, t' be closed expressions and $t \xrightarrow{lcv} t'$. Then $t \sim_c t'$.*

Proof. First let a normal order reduction for t' be given. We use induction on the number of lbeta-reductions and the size $\xi(t')$. First assume that the reduction is finite. We make induction on triples (t, t', red), where red is a normal order reduction from t' to HNF. The induction ordering is: $(t_1, t_1', red_1) < (t_2, t_2', red_2)$, iff $(\#(lbeta, red_1), \xi(t_1)) < (\#(lbeta, red_2), \xi(t_2))$ in the lexicographical ordering. The induction hypothesis is that we can find a normal order reduction for t with the same number of lbeta-reductions. We go through the three possibilities in Lemma 6.5. In the first diagram, induction is easy. In the second diagram, let $t \xrightarrow{no,cp} t_1 \xrightarrow{lcv} t_2 \xrightarrow{lcv} t_3 \xleftarrow{no,cp} t'$. Lemma 6.8 shows that we can apply the induction hypothesis first to the triple (t_2, t_3, red_3), where $red = t \xrightarrow{no,cp} red_3$. Then we can use the hypothesis for the triple (t_1, t_1, red_2), where red_2 is the existing n-o-reduction for t_2. In the last diagram, we use Lemma 6.4 and 4.8, which shows an (i,cppar) reduction in front of a normal order reduction leaves the number of lbeta reductions unchanged. Since lcv keeps the property of being a HNF, we have shown that there is a finite n-o-reduction to a HNF starting from t.
If the reduction starting from t' is infinite, then we use the same arguments for the claim: "the number of lbeta reductions of a normal order reduction of t' is greater than n". This permits to construct an infinite n-o-reduction for t.

Let t have a finite or infinite n-o-reduction. Similar as above, we base the proof on the number of lbeta-reductions in a n-o-reduction of t and t'.

□

7 Deterministic subexpressions

In order to provide optimized lambda-lifting and to clarify the relation to the deterministic lambda calculus, we identify certain subexpressions that can be copied as they can be in the deterministic lambda calculus. This is not intended as an operational rule for an implementation, but only for proving correctness of lambda-lifting with deterministic maximal free expressions, and also of correctness of the usual lambda-calculi rules in the absence of `choice`.

Definition 7.1 *Let t be a closed expression. Then a subexpression s of t is* deterministic *iff s is an expression without occurrences of* `choice`*, which is either closed, or in which all free variables are let-bound variables and moreover, all the let-bound variables are bound to deterministic subexpressions of t.*

Definition 7.2 *A subexpression of t is* reproducible, *iff it is either a lambda-abstraction, a variable, the constant* `choice` *or deterministic.*

The following rule (pdcp) is a parallel copy rule that is directly related to the lambda calculus. The rule (pdld) defined below is used for technical purposes.

Definition 7.3 *Let t be a closed expression. The reduction $t \xrightarrow{pdcp} t'$ is defined as follows: Use the same ordering $<$ as in Lemma 4.6*
Select an antichain $W \subseteq V_{let}(t)$, such that the variables in W are introduced by lets, and all terms let-bound to a variable in W are deterministic. Then define the relation $\xrightarrow{pdcp}_W$ for subexpressions s of t as follows:

- $x \xrightarrow{pdcp}_W x$
- $x \xrightarrow{pdcp}_W t'_x$ *if x is let-bound in t to the (deterministic) expression t_x, and t'_x is a renamed version of t_x.*
- $\texttt{choice} \xrightarrow{pdcp}_W \texttt{choice}$
- *If $(s_1\ s_2)$ is a subexpression of t, and $s_1 \xrightarrow{pdcp}_W s'_1$, $s_2 \xrightarrow{pdcp}_W s'_2$, then $s_1\ s_2 \xrightarrow{pdcp}_W s'_1\ s'_2$.*
- *If $(\lambda\ z\ .\ s_1)$ is a subexpression of t, and $s_1 \xrightarrow{pdcp}_W s'_1$, then $(\lambda\ z\ .\ s_1) \xrightarrow{pdcp}_W (\lambda\ z\ .\ s'_1)$.*
- *Let $(\texttt{let}\ z = t_z\ \texttt{in}\ s_z)$ be a subexpression of t. If $s_z \xrightarrow{pdcp}_W s'_z$, $t_z \xrightarrow{pdcp}_W t'_z$, then $(\texttt{let}\ z = t_z\ \texttt{in}\ s_z) \xrightarrow{pdcp}_W (\texttt{let}\ z = t'_z\ \texttt{in}\ s'_z)$. Note that if $z \in W$ then $t_z = t'_z$.*

Let $t \xrightarrow{pdcp} t'$ iff $t \xrightarrow{pdcp}_W t'$ for some antichain W.

Note that pdcp may also copy (deterministic) variables.

Definition 7.4 *Let t be a closed expression. The reduction $t \xrightarrow{pdld} t'$ is defined as follows.*
Select a set $W \subseteq V_{let}(t)$, such that the variables in W are deterministic. Then define the relation $\xrightarrow{pdld}_W$ for subexpressions s of t as follows:

- $x \xrightarrow{pdld}_W x$
- $\texttt{choice} \xrightarrow{pdld}_W \texttt{choice}$
- *If $(s_1\ s_2)$ is a subexpression of t, and $s_1 \xrightarrow{pdld}_W s'_1$, $s_2 \xrightarrow{pdld}_W s'_2$, then $s_1\ s_2 \xrightarrow{pdld}_W s'_1\ s'_2$.*
- *If $(\lambda\ z\ .\ s_1)$ is a subexpression of t, and $s_1 \xrightarrow{pdld}_W s'_1$, then $(\lambda\ z\ .\ s_1) \xrightarrow{pdld}_W (\lambda\ z\ .\ s'_1)$.*
- *Let $(\texttt{let}\ z = t_z\ \texttt{in}\ s_z)$ be a subexpression of t. Let $s_z \xrightarrow{pdld}_W s'_z$. If $z \in W$, then let $t_z \equiv t'_z$. Otherwise, let $t_z \xrightarrow{pdld}_W t'_z$ and $(\texttt{let}\ z = t_z\ \texttt{in}\ s_z) \xrightarrow{pdld}_W (\texttt{let}\ z = t'_z\ \texttt{in}\ s'_z)$.*
- *Let $s_1 \xrightarrow{pdld}_W s_2$. If $x \in W$, let the subexpression of t be $(\texttt{let}\ x = t_x\ \texttt{in}\ s_x)$, such that s_1 is a subexpression of s_x. Then $s_1 \xrightarrow{pdld}_W (\texttt{let}\ x' = t'_x\ \texttt{in}\ s'_2)$, where t'_x is a renamed version of t_x and s'_2 is a version of s_2, where any occurrences of x are renamed by x'.*

Let $t \xrightarrow{pdld} t'$ iff $t \xrightarrow{pdld}_W t'$ for some W.

Lemma 7.5 *All reductions in λ_{nd} , λ^+_{nd} and the reductions pdcp, pdld preserve the property that a subexpression is deterministic.*

Proof. An easy analysis of the cases. □

We give the commutation and forking diagrams for pdcp and pdld
We assume that pdcp is internal, i.e., has no normal order component.

Lemma 7.6 *A complete set of commutation diagrams for (pdcp) is:*

- $pdcp \circ (no, a) \rightsquigarrow (no, a) \circ (no, cp)^? \circ\ pdcp^?$, *where a means a reduction in the base calculus.*
- $pdcp \circ (no, cp) \rightsquigarrow (no, llet)^* \circ (no, cp) \circ (no, cp)^? \circ pdcp^? \circ pdld^? \circ \xleftarrow{i,llet,*} \circ \xleftarrow{i,cp}$.
- $pdcp \circ (no, cp) \rightsquigarrow (no, cp) \circ (no, cp)^? \circ pdcp^? \circ \xleftarrow{i,cppar}$
- $pdcp \circ (no, a) \rightsquigarrow (no, a) \circ (no, cp)^? \circ pdcp^? \circ \xleftarrow{i,a,*}$ *for $a \in \{lbeta, lapp\}$.*
- $pdcp \circ (no, lapp) \rightsquigarrow (no, llet) \circ (no, cp)^? \circ pdcp^? \circ pdld \circ \xleftarrow{i,llet,*}$.
- $pdcp \circ (no, llet) \rightsquigarrow (no, llet) \circ (no, cp)^? \circ pdcp^? \circ pdld \circ (i, llet)^? \circ \xleftarrow{i,llet,*}$.

Lemma 7.7 *A complete set of commutation diagrams for (pdld) is:*

- $pdld \circ (no, nd) \rightsquigarrow (no, nd) \circ pdld$.
- $pdld \circ (no, a) \rightsquigarrow (no, a) \circ pdld \circ \xleftarrow{i,a,*}$ *for all* $a \in \{cp, llet, lapp, lbeta\}$.
- $pdld \circ (no, a) \rightsquigarrow pdld$ *for* $a \in \{lapp, llet\}$.

Proof. There are less complications than in the commutation case for pdcp. We illustrate a complex case:
$(\texttt{let } y = (\texttt{let } x = t_x \texttt{ in } t_y) \texttt{ in } y)$
$\xrightarrow{pdld}$ $(\texttt{let } y = (\texttt{let } x = t_x \texttt{ in } t_y) \texttt{ in } (\texttt{let } y = (\texttt{let } x = t_x \texttt{ in } t_y) \texttt{ in } y))$
$\xrightarrow{no,llet}$ $(\texttt{let } y = (\texttt{let } x = t_x \texttt{ in } t_y) \texttt{ in } (\texttt{let } x = t_x \texttt{ in } (\texttt{let } y = t_y \texttt{ in } y)))$.
The other reduction gives: $(\texttt{let } y = (\texttt{let } x = t_x \texttt{ in } t_y) \texttt{ in } y)$
$\xrightarrow{no,llet}$ $(\texttt{let } x = t_x \texttt{ in } (\texttt{let } y = t_y \texttt{ in } y))$
$\xrightarrow{pdld}$ $(\texttt{let } x = t_x \texttt{ in } (\texttt{let } y = t_y \texttt{ in } (\texttt{let } x' = t'_x \texttt{ in } (\texttt{let } y' = t'_y \texttt{ in } y')))) \xleftarrow{i,llet}$. The pdld reduction in the second reduction sequence fulfills the condition of the definition, since the replaced positions are not within a copied body. □

In the following we use the measure of a reduction sequence μ that is a pair, compared lexicographically, where the first component μ_1 is the number of nd, lbeta, and cp-reductions, and the second component μ_2 is the measure φ defined in subsection 3.1.

Lemma 7.8 *Let an n-o-reduction red from t (to a HNF) be given. If* $t \xrightarrow{\rho} t'$, *with* $\rho \in \{llet, lapp, cp, lbeta\}$, *then there is an n-o-reduction red' from t' (to a HNF) with* $\mu_1(red') \leq \mu_1(red)$

Proof. Follows from lemma 5.4 □

Lemma 7.9 *Let an n-o-reduction red' from t' (to a HNF) be given. If* $t \xrightarrow{llet} t'$, *then there is an n-o-reduction red from t (to a HNF) with* $\mu_1(red) = \mu_1(red')$

Proof. Follows from lemma 4.8 □

Lemma 7.10 *Let* $t \xrightarrow{pdld} t'$. *If t' has an n-o-reduction red' to a HNF, then t has an n-o-reduction red to a HNF, where* $\mu_1(red) \leq \mu_1(red')$.

Proof. We use lemma 7.7. The second diagram in lemma 7.7 has a backward reduction, for which we need Lemma 7.8. □

Lemma 7.11 *Let* $t \xrightarrow{pdld} t'$. *If t' has an infinite n-o-reduction red', then t has an infinite n-o-reduction red.*

Proof. We use lemma 7.7. The second diagram in lemma 7.7 has a backward reduction, for which we need lemma 5.6. For the third diagram we need the argument that an infinite n-o-reduction cannot contain solely of lapp, llet-reductions. □

Lemma 7.12 *Let* $t \xrightarrow{pdcp} t'$. *If t' has a finite n-o-reduction to HNF, then t has a finite n-o-reduction to a HNF.*

Proof. By induction using the following lexicographically ordered measure: Let $t \xrightarrow{pdcp} t'$ and red' be a n-o-reduction to HNF from t'. Then the first component of the measure is $\mu(red')$, the second is $\xi(t)$.
If the first reduction from t' is of type nd, lbeta, or cp, then the diagrams in lemma 7.6 show that the first component is sufficient to use the induction, where lemmas 7.8, 7.7, and 7.10 are used.
If the first reduction in red' is a llet or lapp-reduction. Then the corners of the diagram are $t \xrightarrow{no,+} t_1$, $t \xrightarrow{pdcp} t'$, $t' \xrightarrow{no,+} t_2$. The tail of the n-o-reduction red' is red'_2. We use for the last diagram in lemma 7.6, the claim of lemma 7.9 to show that $\mu_1(red'_2) \leq \mu_1(red')$. In any case, we can apply the induction hypothesis since $\xi(t_1) < \xi(t)$.
If t' is already a HNF, then t is also a HNF. □

Lemma 7.13 *Let* $t \xrightarrow{pdcp} t'$. *If t' has an infinite n-o-reduction red, then t has an infinite n-o-reduction.*

Proof. By induction on μ.
The diagrams show that there is a reduction sequence $t \xrightarrow{no,+} t_1 \xrightarrow{pdcp} t'_1$. It is easy to see for all cases of diagrams in lemma 7.6 that if t' has an infinite n-o-reduction, then t'_1 has an infinite n-o-reduction, where lemma 7.11 and lemma 5.6 is required. Since every such step adds at least one n-o-reduction to the n-o-reduction sequence after t, we can construct an infinite n-o-reduction for t. □

Lemma 7.14 *A complete set of forking diagrams for internal pdcp is:*

- $\xleftarrow{no,cp,?} \circ \xleftarrow{no,a} \circ \xrightarrow{pdcp} \rightsquigarrow \xrightarrow{pdcp} \circ \xleftarrow{no,a}$
- $\xleftarrow{no,cp,?} \circ \xleftarrow{no,a} \circ \xrightarrow{pdcp} \rightsquigarrow \xrightarrow{pdcp} \circ \xleftarrow{i,a,*} \circ \xleftarrow{no,a}$ *for* $a \in \{lapp, lbeta, cp\}$.
- $\xleftarrow{no,cp,?} \circ \xleftarrow{no,llet} \circ \xrightarrow{pdcp} \rightsquigarrow \xrightarrow{pdcp} \circ \xrightarrow{pdld,?} \circ \xleftarrow{i,llet,*} \circ \xleftarrow{no,llet,?}$.

Proof. By checking the possible overlaps using the same techniques as before. □

Lemma 7.15 *A complete set of forking diagrams for (pdld) is:*

- $\xleftarrow{no,a} \circ \xrightarrow{pdld} \rightsquigarrow \xrightarrow{pdld} \circ \xleftarrow{no,a}$ *for all a in the base calculus.*
- $\xleftarrow{no,cp} \circ \xrightarrow{pdld} \rightsquigarrow \xrightarrow{pdld} \circ \xleftarrow{no,cp} \circ \xleftarrow{no,loa,*}$.
- $\xleftarrow{no,nd} \circ \xrightarrow{pdld} \rightsquigarrow \xrightarrow{pdld} \circ \xleftarrow{no,nd} \circ \xleftarrow{no,loa,*}$.
- $\xleftarrow{no,a} \circ \xrightarrow{pdld} \rightsquigarrow \xrightarrow{pdld} \circ \xleftarrow{i,a,*} \circ \xleftarrow{no,a} \circ \xleftarrow{no,loa,*}$ *for all* $a \in \{lapp, lbeta, llet\}$.
- $\xleftarrow{no,llet} \circ \xrightarrow{pdld} \rightsquigarrow \xrightarrow{pdld} \circ \xleftarrow{i,llet,*}$.

Proof. By checking the possible overlaps. □

Lemma 7.16 *Let $t \xrightarrow{pdld} t'$ such that t has a finite n-o-reduction to a HNF. Then t' has a finite n-o-reduction to a HNF.*

Proof. By induction on the length of an n-o-reduction of t, the diagrams in lemma 7.15 show that there is a mixed reduction for t' to a HNF. Theorem 4.9 shows that there is also a n-o-reduction to some HNF. □

Lemma 7.17 *Let $t \xrightarrow{pdcp} t'$ such that t has a finite n-o-reduction to a HNF. Then t' has a finite n-o-reduction to a HNF.*

Proof. By induction on the length of an n-o-reduction of t, the diagrams in lemma 7.15 and lemma 7.16 show that there is a mixed reduction for t' to a HNF. Theorem 4.9 shows that there is also a n-o-reduction to some HNF. □

Lemma 7.18 *Let $t \xrightarrow{pdcp} t'$ such that t has an infinite n-o-reduction. Then t' has an infinite n-o-reduction.*

Proof. We show by induction on the number of lbeta-reduction, that if t has an n-o-reduction with more than n lbeta-reductions, then this holds also for t'.
First, if $s \xrightarrow{pdld} s'$ and s has an n-o-reduction with more than n lbeta-reductions, then we can construct a mixed reduction for s' with not less than n n-o-lbeta-reductions by lemma 7.15. The commutation lemma 4.8 then shows that there are at least n lbeta-reductions in the n-o-reduction after using the commutation rules.
We use lemma 7.14 to show that the same arguments apply to pdcp.
Finally, every infinite n-o-reduction has an infinite number of lbeta-reductions, hence the lemma holds. □

Theorem 7.19 *If $t \xrightarrow{pdcp} t'$, then $t \sim_c t'$*

Proof. Follows from lemmas 7.12, 7.13, 7.17, 7.18. □

8 Lambda-lifting using deterministic maximal free expressions

In general, lambda-lifting using expressions larger than variables (maximal free expressions in [PJ87]) is not correct for non-deterministic reduction. However, it can be used for expressions that do not use choice, i.e., for deterministic maximal free expressions (dmfe). We provide a definition that generalizes lambda-lifting:

Definition 8.1 *Let $C[D[t_1, \ldots, t_n]]$ be an expression, such that the t_i are reproducible expressions. Then the following relation generalizes dmfe-lambda-lifting:*

$$C[D[t_1, \ldots, t_n]] \xrightarrow{ll-dmfe} C[((\lambda z_1, \ldots, z_n.D[z_1, \ldots, z_n]\ t_1\ \ldots t_n)]$$

where z_i are fresh variables.

Theorem 8.2 *Let $C[D[t_1, \ldots, t_n]]$ be an expression, such that the t_i are reproducible expressions. Then $C[(\lambda z_1, \ldots, z_n.D[z_1, \ldots, z_n])\ t_1\ \ldots t_n] \sim_c C[D[t_1, \ldots, t_n]]$*

Proof. Using Theorem 7.19, 6.3 and 6.9 we show that the lambda-lifted expression is contextually equivalent to the previous one. $C[(\lambda z_1, \ldots, z_n.D[z_1, \ldots, z_n])\ t_1\ \ldots t_n] \sim_c C[(\texttt{let}\ z_1 = t_1\ \texttt{in}\ \lambda z_2, \ldots, z_n.D[z_1, \ldots, z_n])\ t_2\ \ldots t_n] \xrightarrow{pdcp\ |\ lcv} C[(\texttt{let}\ z_1 = t_1\ \texttt{in}\ \lambda z_2, \ldots, z_n.D[t_1, z_2 \ldots, z_n])\ t_2\ \ldots t_n] \xrightarrow{ldel} C[(\lambda z_2, \ldots, z_n.D[t_1, z_2 \ldots, z_n])\ t_2\ \ldots t_n]$. By induction on n and using the lemmas above, we obtain the claim that this is $\sim_c C[D[t_1, \ldots, t_n]]$. □

In general, lambda-lifting using arbitrary maximal free expressions [PJ87] is not correct for non-deterministic reduction:

Example 8.3 *The expression* $(\texttt{let}\ z = \lambda x.(\texttt{choice}\ 1\ x)\ \texttt{in}\ (z\ 0) + (z\ 0))$ *has* $\{0, 1, 2\}$ *as possible results. Using unrestricted lambda-lifting with maximal free expressions, the following expression would result:* $(\texttt{let}\ z = (\lambda y.\lambda x.(y\ x))\ (\texttt{choice}\ 1)\ \texttt{in}\ (z\ 0) + (z\ 0))$, *which has as possible results* $\{0, 2\}$.

Example 8.4 *This example demonstrates that the rule let-over-lambda is not correct in the calculus λ_{nd}. The expressions* $(\texttt{let}\ x = t_x\ \texttt{in}\ (\lambda\ y\ .\ t_y))$ *and* $(\lambda\ y\ .\ (\texttt{let}\ x = t_x\ \texttt{in}\ t_y))$ *are not behaviorally equivalent: The expression* $(\texttt{let}\ z = (\texttt{let}\ x = \texttt{choice}\ 0\ 1\ \texttt{in}\ (\lambda\ y\ .\ x))\ \texttt{in}\ (z\ 0) + (z\ 0))$ *has as possible results* $\{0, 2\}$, *whereas* $(\texttt{let}\ z = (\lambda\ y\ .\ (\texttt{let}\ x = \texttt{choice}\ 0\ 1\ \texttt{in}\ x))\ \texttt{in}\ (z\ 0) + (z\ 0))$ *has* $\{0, 1, 2\}$ *as possible results.*

9 Relation to classical lambda calculus

We consider the usual lambda calculus with β-reduction and β-equivalence, where we assume implicit α-conversions. Let τ be a translation of choice-free expressions with $\tau((\texttt{let}\ x = s\ \texttt{in}\ t)) = ((\lambda\ x\ .\ t)\ s)$. Then the following holds:

Theorem 9.1 *Let s, t be choice-free closed expression. Then*

- $\tau(s) \xrightarrow{\beta} \tau(t) \Rightarrow s \sim_c t$
- $s \xrightarrow{*}_{\lambda,nd} t \Rightarrow \tau(s) \xleftrightarrow{\beta,*} \tau(t)$.
- *Let t be an expression without lbeta-redex. Then t is in HNF iff $\tau(t)$ can be reduced to a HNF as follows: First mark all $\beta-$redexes in the expression; then reduce the expression using a normal order strategy, where only marked redexes are allowed to be reduced.*

Proof. The first implication follows from Theorem 7.19. The second implication follows straightforwardly by proving this for the reductions llet, lapp, cp, and lbeta. The third claim holds, since if t is a HNF w.r.t. $\lambda_{nd,let}$, the transformed expression can be reduced to a HNF by first marking the redexes, then reducing only the marked redexes. The other direction follows, since every redex of $\tau(t)$ corresponds to a let-expression in t. □

10 Remarks

10.1 Supercombinators

A common step in compiling lazy functional programming languages is to transform expressions into a set of definitions of supercombinators. This requires lambda-lifting or an equivalent technique, which is clarified in Theorem 8.2. Further transformations by extracting non-recursive supercombinators do not present a problem.

10.2 Recursion

The fixed point combinator $Y := \lambda f . (\lambda x . f(x\ x))\ (\lambda x . f(x\ x))$ is sufficient to express recursion. It is a fixed point combinator, since for a lambda-expression F: $Y\ F \to$ (let $f = F$ in $(\lambda x . f(x\ x))\ (\lambda x . f(x\ x))$) $\xrightarrow{ldel}$ $(\lambda x . F(x\ x))\ (\lambda x . F(x\ x)) \xrightarrow{*}$ (let $x = (\lambda y . F(y\ y))$ in $F(x\ x)$)) $\xrightarrow{*} F((\lambda y . F(y\ y))\ (\lambda y . F(y\ y)))$. Using the criterion of behavioural equivalence it appears to be possible in to use Y implemented as a cyclic fixed-point combinator if F is deterministic or a lambda abstraction of at least two arguments. However, see the discussion on recursion in [MOW98]. For recursively defined constants that are not deterministic like the constant $L :=$ `choice` $1\ (1+L)$, a noncyclic implementation of Y is required.

11 Applications of the calculus λ_{nd}

Natural EL [HNSSH97] is a lazy functional programming language that uses direct calls for I/O. The implementors based the compiler on [PJ87], but they soon detected that the transformation rules did not properly work in this framework. In particular the lambda-lifting rule using maximal free expressions once introduced an infinite loop by using the value of a previous I/O-action instead of repeating the I/O. This was remedied by lambda-lifting variables only. There are also other restrictions in Natural EL, for example Natural EL uses a non-cyclic Y-combinator. Recently, Nigel Hutchison told me that the optimzations crossed let-boundaries, but not λ-boundaries. Happily, the combination of all the methods, in particular sharing of nodes, finally produced a lazy functional programming language with an easy-to-grasp operational behavior.

The calculus λ_{nd} can be seen as an important part of the theoretic foundation for such a language and for a compiler based on transformations into supercombinators. The I/O-calls can be modeled in λ_{nd} as a big `choice`-expression. The results on deterministic subexpressions are of practical advantage in that they provide an (easily) decidable criterion to use optimizing transformations, for example lambda-lifting with maximally free expressions.

The effect of using λ_{nd} in a lazy functional programming language can be illustrated by three applications: i) The usage of a trace functions that outputs a certain text by a direct call and then returns `True`. The trace expression is simulated by (`choice True` $\bot$) and thus cannot be optimized away. ii) Calling a random number generator using a direct call is easily possible. An infinite list of random numbers could be defined by `randlist = (rand () : randlist`. Interestingly, the logical sequence may be different from the sequence of calls to the random oracle. iii) Several threads of execution are possible by simply permitting (conservative) parallel evaluation. For example, parallel evaluation of the expression `askint () + askint ()` using an appropriate user interface would behave as follows: Two windows asking for a number are opened, the user can decide which to answer first or second. Only if both are answered, the sum is returned as a result.

12 Related work

There is a lot of work on non-deterministic computation. It is impossible to cite or discuss the literature in depth. We confine ourselves to a selection of papers that is concerned with higher-order functions and non-determinism.

There are papers that investigate combinations of lambda-calculi where a fixed (strict or call-by name or call-by value) reduction strategy is used e.g. [Man95]. Some other approaches [Ong93, DP95, San94] do not take care of sharing properties and thus suffer from the "implicit choice"-problem mentioned in the introduction.

Bottom-avoiding choice operators like `amb` or `por` as investigated for example in [HO90, HO89, HM92, Bou94] corresponds to the operational semantics of (local) speculative evaluation; i.e. it is not like a committed choice. In [Bou94], confluence of a calculus using (por) is shown, which is related to the properties of λ_{nd}, but the calculi are different, since λ_{nd} builds upon a stronger contextual equivalence (Definition 5.1) and thus justifies a different set of valid program transformations.

In [Pat91] similar ideas are developed for an application in the field of functional-logic programming, but rigorous consideration of the operational behavior are missing. [AC79] also considered sharing (of ground expressions) as an important technique for modeling non-determinism. However, their calculus is also different, since they included the let-over-lambda rule, which is not correct in our setting (see Example 8.4).

Explicit sharing in functional programming is also a base for investigations in optimal reductions [Yos93, ACCL91] and in [Lau93] for a better understanding of implementations of lazy functional programs.

Our calculus can be seen as a generalization of the calculi in [MOW98, AF97] though there are some differences. In [MOW98], the core calculus has garbage collection (ldel) as an additional rule, which seems to be not the minimal calculus. The normal order reduction in [MOW98] is defined differently: it uses "strict" evaluation of the let-rules, whereas ours is "lazy". There are only slight differences, for example lengths of normal order reductions are in some examples longer than for our calculus (there may be more llet's). We have not explored the way to base the proof on their n-o-reduction, but experience shows that the proofs based on diagrams are very sensible to minor changes in the rules. The proofs of our theorems in a setting based on the normal order reduction definition of [MOW98] would be very different and perhaps more complex.

The calculus in [AF97] models almost only the normal order reduction and thus has a narrow range of program transformations.

13 Further Research

The extension of our calculus to non-strict functional languages using constructors and a polymorphic type system has to be investigated. A more detailed analysis of the

input-output behavior of a functional programming language based on the non-determinism is required. The properties of choice as a parallel combinator have to be clarified, where associativity, commutativity and idempotency appear to hold. An investigation in a semantics for λ_{nd} is in order.

References

[Abr90] Samson Abramsky. The lazy lambda calculus. In D. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1990.

[AC79] Egidio Astesiano and Gerardo Costa. Sharing in nondeterminism. In *Proc. 6th ICALP 79*, pages 1–15, 1979.

[ACCL91] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J Lévy. Explicit substitutions. *J. functional programming*, 4(1):375–416, 1991.

[Ach96] Peter Achten. *Interactive functional programs: models, methods and implementation.* PhD thesis, Computer Science Department, University Nijmegen, 1996.

[AF97] Z.M. Ariola and M Felleisen. The call-by-need lambda calculus. *J. functional programming*, 7(3):265–301, 1997.

[AFM+95] Z.M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A call-by-need lambda calculus. In *Principles of programming languages*, pages 233–246, San Francisco, California, 1995. ACM Press.

[Bar84] H.P. Barendregt. *The Lambda Calculus. Its Syntax and Semantics.* North-Holland, Amsterdam, New York, 1984.

[Bou94] G. Boudol. Lambda-calculi for (strict) parallel functions. *Information and Computation*, 108:51–127, 1994.

[DP95] U. De'Liguoro and A. Piperno. Nondeterministic extensions of untyped λ-calculus. *Information and Computation*, 122:149–177, 1995.

[HM92] J. Hughes and A. Moran. A semantics for locally bottom-avoiding choice. In *Proc. Glasgow functional programming workshop 1992*, Workshops in Computing. Springer-Verlag, 1992.

[HNSSH97] N.W.O. Hutchison, U. Neuhaus, M. Schmidt-Schauß, and C.V Hall. Natural Expert: A commercial functional programming environment. *J. of Functional Programming*, 7(2):163–182, 1997.

[HO89] J. Hughes and J. O'Donnell. Expressing and reasoning about non-deterministic functional programs. In *Glasgow workshop on functional programming 1989*, Workshops in Computing, pages 308–328. Springer-Verlag, 1989.

[HO90] J. Hughes and J. O'Donnell. Nondeterministic functional programming with sets. In *IV Higher Order Workshop*, Workshops in Computing, pages 11–31. Springer-Verlag, 1990.

[How89] D. Howe. Equality in lazy computation systems. In *4th IEEE Symp. on Logic in Computer Science*, pages 198–203, 1989.

[Hue80] G.P. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. of the ACM*, 27:797–821, 1980.

[Lau93] J Launchbury. A natural semantics for lazy evaluation. In *Proc. 20th Principles of Programming Languages*, 1993.

[Man95] L. Mandel. *Constrained Lambda Calculus.* Verlag Shaker, Aachen, Germany, 1995.

[MOW98] John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. *J. of Functional programming*, 1998. to appear.

[New42] M.H.A. Newman. On theories with a combinatorial definition of "equivalence". *Annals of Mathematics*, 2:223–243, 1942.

[NSvP91] E. Nöcker, J. E. Smetsers, M. van Eekelen, and M. J. Plasmeijer. Concurrent Clean. In *Proc of Parallel Architecture and Languages Europe (PARLE'91)*, number 505 in LNCS, pages 202–219. Springer Verlag, 1991.

[Ong93] C.-H. L. Ong. Non-determinism in a functional setting. In *Proc. 8th IEEE Symposium on Logic in Computer Science (LICS '93)*, pages 275–286. IEEE Computer Society Press, 1993.

[Pat91] Ross Paterson. A tiny functional language with logical features. In M.Coppo et.al., editor, *Declarative Programming, Sasbachwalden*, pages 66–79, 1991.

[PHA+97] J. Peterson [ed.], K. Hammond [ed.], L. Augustsson, B. Boutel, W. Burton, J. Fasel, A. D. Gordon, J. Hughes, P. Hudak, Th. Johnsson, M. Jones, E. Meijer, S. Peyton Jones, A. Reid, and P. Wadler. Report on the programming language Haskell: A non-strict, purely functional language, Version 1.4, 1997.

[PJ87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall International, London, 1987.

[PJL91] Simon L. Peyton Jones and David R. Lester. *Implementing Functional Languages: a Tutorial.* Prentice-Hall International, London, 1991.

[San94] D. Sangiorgi. The lazy lambda calculus in a concurrency scenario. *Information and Computation*, 111:120–153, 1994.

[SS92] H. Søndergard and P. Sestoft. Non-determinism in functional languages. *The Computer Journal*, 35(5):514–523, 1992.

[Yos93] N. Yoshida. Optimal reductions in weak-λ-calculus with shared environments. In *Proc. functional programming languages and computer architecture*, pages 243–252. ACM press, 1993.