

# A Lambda-Calculus with letrec, case, constructors and non-determinism

Manfred Schmidt-Schauß and Michael Huber

Fachbereich Informatik  
Johann Wolfgang Goethe-Universität  
Postfach 11 19 32  
D-60054 Frankfurt, Germany  
E-mail: [schauss@ki.informatik.uni-frankfurt.de](mailto:schauss@ki.informatik.uni-frankfurt.de)

**Abstract.** A non-deterministic call-by-need lambda-calculus  $\lambda_{ndlr}$  with case, constructors, letrec and a (non-deterministic) erratic choice, based on rewriting rules is investigated. A standard reduction is defined as a variant of left-most outermost reduction. The semantics is defined by contextual equivalence of expressions instead of using  $\alpha\beta(\eta)$ -equivalence. It is shown that several program transformations are correct, for example all (deterministic) rules of the calculus, and in addition the rules for garbage collection, removing indirections and unique copy. This shows that the combination of a context lemma and a meta-rewriting on reductions using complete sets of commuting (forking, resp.) diagrams is a useful and successful method for providing a semantics of a functional programming language and proving correctness of program transformations.

## 1 Introduction

Functional programming languages are based on extended lambda calculi and the corresponding rewrite semantics. There are several methods of giving these languages a semantics and proving the correctness of program transformations:

- A *denotational semantics* uses a mathematical domain and a mapping from expressions to their denotation. This defines an equivalence of expressions, which can be used to define a notion of correctness of program transformations. This area is well-developed, but reaches its limits if non-deterministic operations are possible in the language.
- An *operational semantics* defining the evaluation of expressions (the execution, resp.). Sometimes this is used with a kind of syntactic equality (e.g.  $\alpha\beta(\eta)$ -equality in the lambda-calculus). It could also be complemented by a behavioral equivalence, which can be used to define a notion of correctness of program transformation.
- A *contextual semantics* is a kind of operational semantics as above enhanced with an approximation relation based on a contextual preordering (see e.g. [Smi92,MST96,Pit97]). An expression  $s$  has less information than an expression  $t$ , iff in all contexts  $C[\ ]$ , if  $C[s]$  gives some information (e.g. terminates),

then  $C[t]$  also gives some information (i.e. terminates). This notion is directly adapted to define a notion of correctness of program transformation. Often it gives the intuitive correct notion of program equivalence, and hence also of correct program transformations.

The advantage of the contextual semantics is that the number of equality relations is maximal and that the derived properties are independent of a specific domain. The properties of the contextual preorder are comparable to the orderings in domains; for example it is possible to use fixed-point constructions for recursion. The contextual semantics is superior to the more syntax-oriented  $\alpha\beta(\eta)$ -equivalence, since contextual semantics permits considerably more program transformations.

An advantage of contextual semantics over the denotational approach becomes obvious if non-determinism is on board and also sharing in the form of a (non-recursive or recursive) **let**. It appears to be very hard to construct a useful domain for denotational semantics in the presence of non-determinism and higher-order functions, whereas it is possible to use the contextual equivalence for defining an intuitive correct semantics. This can then be used to prove correctness of program transformations sometimes exploiting rewriting techniques. A slight disadvantage of the contextual semantics (w.r.t. economy of proofs) is that it depends on the available syntactic constructs, hence on the set of contexts, and the defined standard reduction.

The prominent syntactic property of the lambda-calculus is confluence of reduction [Bar84]. In the framework of a contextual semantics for the lambda-calculus (see e.g. [Abr90]), confluence is not thus important and is replaced by the correctness of program transformations. The really interesting propositions are:

- Every beta-reduction transforms a program  $P$  into an equivalent one  $P'$ , meaning that  $P$  and  $P'$  are contextually equivalent. This is the required modification of confluence.
- (standardization) Whenever there is a reduction of an expression  $t$  to an abstraction, then the standard reduction terminates, i.e. reduces  $t$  to an abstraction.

These properties can be generalized to extended lambda-calculi, where confluence may be false (see e.g. [AK94]), but contextual equivalence can be easily adapted.

Another approach is Rewriting Logic (see e.g. [Mes00]), which is a step in the direction of providing a semantics for programming languages based on rewriting rules. This appears to work for deterministic languages based on rewriting rules. However, the contextual semantics is our method of choice for the non-deterministic case.

In this paper we present the calculus  $\lambda_{ndlr}$  that is rather close to a non-strict functional core language. Reduction is like lazy call-by-need evaluation in functional programming languages.  $\lambda_{ndlr}$  can be seen as a generalization of the calculus in [KSS98,Kut00] and thus of the calculi in [AFM<sup>+</sup>95,AF97,MOW98], which

treat sharing in the lambda calculus. It also is a generalisation of [MSC99] insofar as the language of expressions is not restricted to have only variables as arguments in applications. The calculus  $\lambda_{ndlr}$  is related to the calculus in [SS00], where a similar language is investigated, but with the emphasis on an IO-interface.

Another method for treating sharing are explicit substitutions [ACCL91], which optimize resource usage of reductions by exploiting sharing, however, it is i) based on  $\alpha\beta(\eta)$ -equivalence and ii) the reduction rules are in general not compatible with non-determinism, i.e. not with  $\lambda_{ndlr}$  nor with the calculus in [Kut00]; in particular, the let-over-lambda-rules are incompatible with non-determinism. Specific ingredients of  $\lambda_{ndlr}$  are

- sharing by using **letrec**, which moreover allows recursive definitions.
- a non-deterministic (erratic) **choice**, which allows to choose between two expressions.
- a modified beta-reduction that prevents an unwanted duplication of non-deterministic expressions.

The motivation to investigate non-determinism is to model interfaces of lazy functional languages to the outside world, i.e. to model input/output. This is done by a simulation of an IO-action by a nested **choice**-expression that represents the set of possible input values of the IO.

The paper proposes to investigate extended lambda-calculi by using operational methods and a contextual semantics. The contextual semantics includes a measure for the number of non-deterministic steps. As a method for proving program transformation to be correct we propose to use complete sets of reduction diagrams in combination with an appropriate context lemma.

The results are that for  $\lambda_{ndlr}$  a rather large set of basic program transformations is proved to be correct. The paper also demonstrates the power of the method, since the reduction rules of  $\lambda_{ndlr}$  are numerous and complex.

As a check-program for complete sets, a program “Jonah“ was implemented to automatically test the complete reduction diagrams using a generate-and-test scheme; Jonah can also be used to compute proposals for complete sets.

As a remaining open problem the paper can be seen as a recommendation to start an investigation into adapting the Knuth-Bendix method to automatically computing the reduction diagrams. However, the reduction diagrams for (llet) for example show that it would be necessary to integrate a kind of meta-description like the Kleene- $*$ .

In this paper we do not present all proofs, but give enough hints and evidence of how the claims can be verified and that they are valid.

## 2 The calculus $\lambda_{ndlr}$

The syntax of the language is as follows:

There is a set of type-names. For every type there are constructors  $c$  coming with an arity  $ar(c)$ . This partitions the set of all constructors into the constructors

belonging to different types. For a type  $A$ ,  $|A|$  is defined to be the number of constructors belonging to  $A$ . The constructors belonging to type  $A$  are indexed, and  $c_{A,i}$  denotes the  $i^{th}$  constructor of type  $A$ .

$$\begin{aligned}
E &::= V \mid C \mid (\text{choice } s \ t) \mid (\text{case}_A E \text{ Alt}_1 \dots \text{Alt}_{|A|}) \mid (E_1 \ E_2) \\
&\quad \mid (\lambda V. E) \mid (\text{letrec } V_1 = E_1, \dots, V_n = E_n \text{ in } E) \\
\text{Alt} &::= (Pat \rightarrow E) \\
Pat &::= c \ V_1 \ \dots \ V_{ar(c)}
\end{aligned}$$

where  $E, E_i$  are expressions,  $A$  is a type,  $V, V_i$  are variables, and  $C$  is a constructor. The variables in a pattern  $Pat$  must be different, and also new ones. Moreover, in a  $\text{case}_A$ -expression, there is exactly one alternative with a pattern of the form  $(c_{A,i} \ y_1 \dots y_n)$  for every constructor  $c_{A,i}$ . The constants  $\text{case}_A$  and  $\text{choice}$  can only occur in a special syntactic construction. Thus expressions where  $\text{choice}$  or  $\text{case}_A$  is applied to a wrong number of arguments are not allowed.

The structure  $\text{letrec}$  obeys the following conditions: The variables in the bindings are all distinct. We also assume that the bindings in  $\text{letrec}$  are commutative, i.e. can be commuted without syntactic change.  $\text{letrec}$  is recursive: I.e. the scope of  $x_i$  in  $(\text{letrec } x_i = E_i \text{ in } E)$  is  $E_i, E$ . This allows to define closed, open expressions and  $\alpha$ -renamings. For simplicity we use the disjoint variable convention. I.e., all bound variables in expressions are assumed to be disjoint. The reduction rules are such that the bound variables in the result are also made distinct by  $\alpha$ -renaming. We also use the convention to omit parenthesis in denoting expression:  $(s_1 \dots s_n)$  denotes  $(\dots (s_1 \ s_2) \dots s_n)$ .

We say that an expression of the form  $(c \ t_1 \dots t_n)$  is a *constructor application*, if  $n \leq ar(c)$ . A constructor application of the form  $(c \ x_1 \dots x_n)$  is called a *pure constructor application*. An expression of the form  $(c \ t_1 \dots t_{ar(c)})$  is called a *saturated constructor application*.

**Definition 2.1.** Let  $R, R^-$ , be context classes defined as follows:

$$\begin{aligned}
R^- &::= [] \mid R^- \ E \mid (\text{case}_A R^- \ \text{alts}) \\
R &::= R^- \mid (\text{letrec } x_1 = E_1, \dots, x_n = E_n \text{ in } R^-) \\
&\quad \mid (\text{letrec } x_1 = R_1^-[\cdot], x_2 = R_2^-[x_1], \dots, x_j = R_j^-[x_{j-1}], \dots \text{ in } R^-[x_j]) \\
&\quad \text{where } R_j^- \text{ is a context of class } R^-
\end{aligned}$$

$R$  is called a *reduction context* and  $R^-$  is called a *weak reduction context*. For a term  $t$  with  $t = R^-[t_0]$ , we say  $R^-$  is *maximal*, iff there is no larger weak reduction context with this property. For a term  $t$  with  $t = R[t_0]$ , we say the reduction context  $R$  is *maximal*, iff it is either a maximal weak reduction context, or of the form  $(\text{letrec } x_1 = R_1^-[\cdot], x_2 = R_2^-[x_1], \dots, x_j = R_j^-[x_{j-1}], \dots \text{ in } R^-[x_j])$ ,

where  $t = (\text{letrec } x_1 = t_1, \dots \text{ in } R^-[x_j])$ ,  $R_1^-[.]$  is maximal for  $t_1$ , and the number  $j$  is maximal.

For example the maximal reduction context of  $(\text{letrec } x_2 = \lambda x.x, x_1 = x_2 \ x_1 \text{ in } x_1)$  is  $(\text{letrec } x_2 = [], x_1 = x_2 \ x_1 \text{ in } x_1)$ , in contrast to the non-maximal reduction context  $(\text{letrec } x_2 = \lambda x.x, x_1 = x_2 \ x_1 \text{ in } [])$ .

The (call-by-need) reduction rules defined in 2.2 follow the principle of minimizing copying at the cost of perhaps following several indirections. This holds for the copy rule (cpn) as well as (case). The technical reason is that this principle assures well-behaved reduction diagrams.

**Definition 2.2.** *The reduction rules are defined below in figure 2. If the context is important, then we denote it as a label of the reduction or state it explicitly. Note that for (case), the typical example is written down, where the position of the **case** is left open. There are two variants, one where the **case** is in the in-expression, and one where the **case**-expression is in the right hand side of a binding. An exceptional case, where perhaps a **letrec**-expression has to be omitted, is the case of a constructor with zero arguments like (**case** Nil ...).*

The union of  $ndl, ndr$  is called  $(nd)$ . Reductions are denoted using an arrow with super and/or subscripts: e.g.  $\xrightarrow{llet}$ . Transitive closure is denoted by a  $+$ , reflexive transitive closure by a  $*$ . E.g.  $\xrightarrow{*}$  is the reflexive, transitive closure of  $\rightarrow$ .

As a short comment of the reduction rules:

- (lbeta) is a sharing version of beta-reduction
- (cpn) is a lazy version of the replacement done by usual beta-reduction, where the copy may jump over several indirections.
- (case) is the generalized **if** for case analysis of values. To find the value to be analyzed, it has to be virtually assembled by following the bindings.
- (llet), (lapp), (lcase) are used to adjust the let-environments
- (nd) is the non-deterministic (erratic) choice.

The next definition is intended to formalize the standard reduction. The idea is to find the reduction that is outermost, in a reduction context and also necessary for making progress in the evaluation.

**Definition 2.3.** *Let  $t$  be an expression. Let  $R$  be the maximal reduction context such that  $t \equiv R[t']$  for some  $t'$ . The standard redex and the corresponding standard reduction  $\xrightarrow{st}$  is defined by one of the following cases:*

- $t'$  is a choice-expression: then use  $(ndr)$  or  $(ndl)$ .
- $R = (\text{letrec } x_1 = t_1, \dots, x_n = t_n \text{ in } [])$ , and  $t'$  is a **letrec**-expression. Then apply (llet) to  $R[t']$ .
- $R = R_0[(\cdot \ t'')]$  where  $R_0$  is a reduction context. If  $t'$  is a **letrec**-expression, then use (lapp) in context  $R_0$ ; If  $t'$  is an abstraction, then use (lbeta) in context  $R_0$ .

- $R = R_0[\text{case}_A \cdot \text{alts}]$ .  
 If  $t'$  is a **letrec**-expression, then use *(lcase)* in context  $R_0$ ;  
 If  $t'$  is a saturated constructor application, then use *(case)* in context  $R_0$ , if it is applicable.
- $R = (\text{letrec } x_1 = [], x_2 = x_1, \dots, x_j = x_{j-1}, \dots \text{ in } R_1^-[x_j])$  where  $R_1^-$  is a weak reduction context.  
 If  $t'$  is an abstraction, then use *(cpn)* as follows:  $(\text{letrec } x_1 = t', x_2 = x_1, \dots, x_j = x_{j-1}, \dots \text{ in } R_1^-[x_j]) \rightarrow (\text{letrec } x_1 = t', x_2 = x_1, \dots, x_j = x_{j-1}, \dots \text{ in } R_1^-[t'])$ .  
 If  $t'$  is a **letrec**-expression, then use *(llet)*, *(lcase)*, or *(lapp)* to flatten the **letrec**-expression  $t'$  into its superexpression.  
 If  $t'$  is a constructor application, and *(case)* is applicable to a case-expression in a reduction context, then apply this *(case)*-reduction.
- $R = (\text{letrec } x_1 = [], x_2 = x_1, \dots, x_j = x_{j-1}, x_{j+1} = R_{j+1}^-[x_j], \dots \text{ in } R_\infty^-[x_k])$  where  $R_i^-, R_\infty^-$  are weak reduction contexts.  
 If  $t'$  is an abstraction, then use *(cpn)* such that the result is:  $R = (\text{letrec } x_1 = t', x_2 = x_1, \dots, x_j = x_{j-1}, x_{j+1} = R_{j+1}^-[t'], \dots \text{ in } R_\infty^-[x_k])$ .  
 If  $t'$  is a **letrec**-expression, then use *(llet)*, *(lcase)*, or *(lapp)* to flatten the **letrec**-expression  $t'$  into its superexpression.  
 If  $t'$  is a constructor application, and *(case)* is applicable to a case-expression in a reduction context, then apply this *(case)*-reduction.

**Lemma 2.4.** *For every term  $t$ : if  $t$  has a standard redex, then this redex is unique. If the standard reduction is not an (nd), then the standard reduction is also unique.*

**Definition 2.5.** *A standard reduction  $s \xrightarrow{st} s_1 \xrightarrow{st} s_2 \dots s_n \xrightarrow{st} t$  has nd-count  $D$ , iff  $D$  is the number of (nd)-reductions in it.*

Note that we use the notion standard reduction also for non-maximal reductions.

**Definition 2.6.** *For a term  $t$  and an nd-count  $D$ ,  $t \Downarrow_D$  holds if there is some standard-reduction starting with  $t$ , and the reduction has nd-count  $D$ .*

Note that a standard reduction for an nd-count  $D$  is in general not unique. Note also that there may be expressions without a standard redex.

**Definition 2.7.** *(contextual preorder and equivalence) Let  $s, t$  be terms. We define:*

$$\begin{aligned}
 s \leq_c t & \text{ iff } \forall C[], \forall D : C[s] \Downarrow_D \Rightarrow (\exists B. D \leq B \wedge C[t] \Downarrow_B) \\
 s \sim_c t & \text{ iff } s \leq_c t \wedge t \leq_c s
 \end{aligned}$$

Note that we permit contexts such that  $C[s]$  is an open term.

**Proposition 2.8.**  $\leq_c$  is a preordering and  $\sim_c$  is an equivalence relation.  
 $s \leq_c t$  implies that  $C[s] \leq_c C[t]$  for all contexts  $C[\cdot]$ . I.e.,  $\leq_c$  is a precongruence on the set of expressions.  
 $s \sim_c t$  implies that  $C[s] \sim_c C[t]$  for all contexts  $C[\cdot]$ . I.e.,  $\sim_c$  is a congruence on the set of expressions.

Note that there are terms  $t$  without a standard redex, i.e. the standard reduction stops. The reasons could be classified as i) type-error like  $(\text{case}_A(\lambda x.x) \dots)$ , ii) a kind of non-termination like  $(\text{letrec } x = x \text{ in } x)$ , iii) as a value or a kind of normal form like  $(\text{Cons True Nil})$  or  $\lambda x.x$ .

The following lemma shows that it is sufficient to use reductions contexts for checking contextual approximation.

**Lemma 2.9.** (Context Lemma) Let  $s, t$  be terms. If for all reduction contexts  $R$  and all nd-counts  $D$ :  $R[s] \Downarrow_D \Rightarrow (\exists B. D \leq B \wedge R[t] \Downarrow_B)$ , then  $s \leq_c t$ .

*Proof.* We prove the more general claim:

if for all  $i$ :  $s_i, t_i$  satisfy the conditions of the lemma for reduction contexts,  
then for all multicontexts  $C[\cdot_1, \dots, \cdot_m]$ :  $C[s_1, \dots, s_n] \Downarrow_D \Rightarrow (\exists B. D \leq B \wedge C[t_1, \dots, t_n] \Downarrow_B)$ .

Assume this is false. Then there is a multicontext  $C$ , an nd-count  $D$ , such that  $C[s_1, \dots, s_n] \Downarrow_D$ , and for all  $B$  with  $D \leq B$ :  $C[t_1, \dots, t_n] \not\Downarrow_B$ .

We select a multicontext,  $C$ , terms  $s_i, t_i$ , and an nd-count  $D$ , and a corresponding reduction, such that the counterexample is minimal w.r.t. the following lexicographic ordering: i) the number of reduction steps of  $C[s_1, \dots, s_n]$ , ii) the number of holes of  $C[\dots]$ .

The search for a standard redex is performed top-down. There are two cases:

1. The search for the reduction context inspects the term in a hole. Then we can assume wlog that the first hole is inspected first. Hence  $C[\cdot, t_2, \dots, t_n]$  is a reduction context. Let  $C' := C[s_1, \cdot_2, \dots, \cdot_n]$ . Since  $C'[s_2, \dots, s_n] \equiv C[s_1, \dots, s_n]$ , we can select the the same standard reduction for nd-count  $D$ . Since the number of holes of  $C'$  is smaller than the number of holes in  $C$ , we obtain some  $B \geq D$  with  $C'[t_2, \dots, t_n] \Downarrow_B$ , which means  $C[s_1, t_2, \dots, t_n] \Downarrow_B$ . Since  $C[\cdot, t_2, \dots, t_n]$  is a reduction context, the preconditions of the lemma imply that there is some  $B' \geq B$  with  $C[t_1, t_2, \dots, t_n] \Downarrow_{B'}$ , a contradiction.
2. The search for the reduction context does not inspect any hole of  $C$ . Then  $C[s_1, \dots, s_n]$  as well as  $C[t_1, \dots, t_n]$  can be reduced using the same standard reduction, since the search for a standard redex takes place only in the outer context  $C[\dots]$ . There are two cases for a reduction:

If the reduction  $C[s_1, \dots, s_n] \xrightarrow{st} s'$  is not (nd), then this may result in  $C'[\dots]$  with more holes, and the holes are filled with copies of  $s_i, t_i$ . Then we get a smaller counterexample since the number of reductions steps is smaller, and since non-(nd) standard reductions are unique.

If the reduction  $C[s_1, \dots, s_n] \xrightarrow{st} s'$  is an (ndl) (or (ndr), respectively), then the reduction of  $s'$  has nd-count  $D' = D - 1$ . We make the corresponding reduction:  $C[t_1, \dots, t_n] \xrightarrow{st, ndl} t'$ . This is a smaller counterexample; hence we get a contradiction also in this case.

**Definition 2.10.** A program transformation is a relation  $T$  between programs (expressions). A program transformation  $T$  is called correct, iff for all expressions  $P, P'$ :  $P TP'$  implies  $P \sim_c P'$ .

The reductions rules in definition 2.2 define corresponding program transformations if they are allowed in arbitrary contexts.

**Definition 2.11.** Let an internal reduction be a non-standard reduction that takes place within a reduction context. Usually, this is denoted by the label  $i$  on the reduction arrow.

We define complete sets of commuting and forking diagrams adapted from [Kut99, Kut00]. In the following definition we use a notation for rewrite rules on reduction sequences. For example  $\xrightarrow{(i, llet)} \cdot \xrightarrow{(st, a)} \sim \xrightarrow{(st, a)} \cdot \xrightarrow{(i, llet)}$ , where  $a$  is a reduction type. The  $\cdot$  on the left hand side is like a joker, and the  $\cdot$  on the right hand side can be seen as an existentially quantified term.

**Definition 2.12.** Assume given a reduction type ( $red$ ) and a set of (complementary) reduction types  $T$ , where the base calculus reduction types are contained in  $T$ , as well as ( $red$ ).

A complete set of commuting diagrams for a reduction ( $red$ ) is a set of rewrite rules on reduction sequences of the form

$$\xrightarrow{i, red} \cdot \xrightarrow{st, a_1} \dots \xrightarrow{st, a_k} \sim \xrightarrow{st, b_1} \dots \xrightarrow{st, b_m} \cdot \xrightarrow{i, c_1} \dots \xrightarrow{i, c_h},$$

where  $c_i \in T$ , such that for every reduction sequence  $s \xrightarrow{i, red} \cdot \xrightarrow{st, *} t$ : Either it can be transformed using one of the meta-reductions into another reduction sequence from  $s$  to  $t$ , such that at least  $\xrightarrow{i, red}$  can be replaced. Or  $\cdot \xrightarrow{st, *} t$  can be prolonged into a longer standard reduction sequence  $\dots \xrightarrow{st, *} t \xrightarrow{st, +} t'$ , such that it can be replaced as above.

A complete set of forking diagrams for a reduction ( $red$ ) is a set of rewrite rules on reduction sequences of the form

$$\xleftarrow{st, a_1} \dots \xleftarrow{st, a_k} \cdot \xrightarrow{i, red} \sim \xrightarrow{i, c_1} \dots \xrightarrow{i, c_h} \cdot \xleftarrow{st, b_1} \dots \xleftarrow{st, b_m},$$

where  $c_i \in T$ , such that: Either every reduction sequence  $s \xleftarrow{st, *} \cdot \xrightarrow{i, red} t$  can be transformed into another reduction sequence between  $s$  and  $t$ , such that at least  $\xrightarrow{i, red}$  is replaced. Or  $s \xleftarrow{st, *} \cdot$  can be prolonged into a reduction sequence  $s' \xleftarrow{st, +} s \xleftarrow{st, *} \cdot$ , such that it can be replaced as above.

We also use reductions not in the base calculus as internal reductions.



It is intended that the corresponding meta-rewriting on reduction sequences terminates, which has to be proved for every such complete set. The complete sets of commuting (forking) diagrams are not unique.

Note that in many cases, the forking diagrams can be derived from the commuting diagrams.

**Lemma 2.13.** *For every reduction that is not a (llet) or (cp)-reduction, i.e.,  $a \in \{(nd), (l\beta), (lapp), (lcase), (case)\}$ , there are no internal reductions. This means, every internal  $a$ -reduction with  $a \in \{(nd), (l\beta), (lapp), (lcase), (case)\}$  is a standard reduction.*

*Proof.* By inspecting all the finitely many cases.

**Proposition 2.14.** *If  $s \xrightarrow{a} t$ , where  $a \in \{(l\beta), (lapp), (lcase), (case)\}$ , then  $s \sim_c t$ .*

*I.e., all the program transformations defined by one of the reductions  $\{(l\beta), (lapp), (lcase), (case)\}$  are correct.*

*Proof.* Let  $s' \xrightarrow{a, []} t'$  by a (a)-reduction on the surface with  $a \in \{(l\beta), (lapp), (lcase), (case)\}$ .

We show  $s' \leq_c t'$  exploiting the context lemma. Let  $R$  be a reduction context. Then  $R[s'] \xrightarrow{i, a} R[t']$  is not possible by Lemma 2.13. Then  $R[s'] \xrightarrow{st, a} R[t']$  by a unique standard reduction, hence if there a reduction for  $R[s']$  with nd-count  $D$ , there is also one for  $R[t']$  with nd-count  $D$ . The context lemma now shows that  $s' \leq_c t'$ .

To show  $t' \leq_c s'$  using the context lemma is similar: If there a standard reduction for  $R[t']$  with nd-count  $D$ , there is also one for  $R[s']$  with nd-count  $D$ . The context lemma now shows that  $t' \leq_c s'$ .

Hence we have shown  $s' \sim_c t'$ . Since  $\sim_c$  is a congruence, we have also that  $C[s'] \sim_c C[t']$  for an arbitrary context  $C$ . Hence the proposition holds.

### 3 Correctness of the reduction (llet)

The union of the reductions (llet), (lapp), (lcase) is denoted as (lll). The reduction  $lll^+$  means a reduction sequence consisting only of (lll)-reductions of length at least 1. Accordingly  $lll^*$  is defined as any number of (lll)-reductions.  $(i, llet)^{0 \vee 1}$  means no reduction or 1 reduction  $(i, llet)$ . In the following two lemmas,  $a$  stands for an arbitrary reduction  $\xrightarrow{a}$ .

**Lemma 3.1.** *A complete set of commuting diagrams for (llet) is:*

$$\begin{array}{c}
- \xrightarrow{(i, llet)} \cdot \xrightarrow{(st, a)} \rightsquigarrow \xrightarrow{(st, a)} \cdot \xrightarrow{(i, llet)} \\
- \xrightarrow{(i, llet)} \cdot \xrightarrow{(st, a)} \rightsquigarrow \xrightarrow{(st, a)} \cdot \xrightarrow{(st, llet)} \\
- \xrightarrow{(i, llet)} \cdot \xrightarrow{(st, lll^+)} \rightsquigarrow \xrightarrow{(st, lll^+)} \cdot \xrightarrow{(i, llet)^{0 \vee 1}}
\end{array}$$

**Lemma 3.2.** *A complete set of forking diagrams for (llet) is:*

$$\begin{array}{c}
- \xrightarrow{(st,a)} \cdot \xrightarrow{(i,llet)} \rightsquigarrow \xrightarrow{(i,llet)} \cdot \xleftarrow{(st,a)} \\
- \xleftarrow{(st,llet)} \cdot \xleftarrow{(st,a)} \cdot \xrightarrow{(i,llet)} \rightsquigarrow \xleftarrow{(st,a)} \\
- \xleftarrow{(st,lll^+)} \cdot \xrightarrow{(i,llet)} \rightsquigarrow \xrightarrow{(i,llet)} \cdot \xleftarrow{(st,lll^+)} \\
- \xleftarrow{(st,lll^+)} \cdot \xrightarrow{(i,llet)} \rightsquigarrow \xleftarrow{(st,lll^+)}
\end{array}$$

**Proposition 3.3.** *If  $s \xrightarrow{(i,llet)} t$ , then  $s \sim_c t$ .*

*I.e. (llet) is a correct program transformation in any context.*

*Proof.* First we assume that the reduction is on top level.

To use the context lemma, we have to show what happens in a reduction context.

I.e. assume that  $s \equiv R[s']$  and  $s'$  is the llet-redex.

Using the forking diagrams, it is possible to construct from a standard reduction of  $s$  a standard reduction of  $t$  with the same nd-count. The context lemma then shows that  $s \leq_c t$ .

Using the commuting diagrams, it is possible to construct from a standard reduction of  $t$  a standard reduction of  $s$  with the same nd-count by shifting the  $(i,llet)$   $\longrightarrow$  to the right. The context lemma then shows that  $t \leq_c s$ .

Together, this means  $s \sim_c t$ .

Finally, the congruence property of  $\sim_c$  implies that a (llet) can be applied everywhere in a term.  $\square$

## 4 Garbage Collection: ldel

Garbage collection in the calculus has two forms, a non-cyclic one, and the other one that also collects cyclic references:

The noncyclic reduction (ldel) is defined as :

$$\begin{array}{l}
(ldel) \quad (\mathbf{letrec} \ x = s \ \mathbf{in} \ t) \rightarrow t \text{ if } x \text{ does not occur in } t \\
(ldel) \quad (\mathbf{letrec} \ x = s, E \ \mathbf{in} \ t) \rightarrow (\mathbf{letrec} \ E \ \mathbf{in} \ t) \text{ if } x \text{ does not occur in } t, E
\end{array}$$

The cyclic reduction (ldelcyc) consisting of (ldelcyc1), (ldelcyc2) is defined as :

$$\begin{array}{l}
(ldelcyc1) \quad (\mathbf{letrec} \ x_1 = s_1, \dots, x_m = s_m \ \mathbf{in} \ t) \rightarrow (\mathbf{letrec} \ x_j = s_j, \dots, x_m = s_m \ \mathbf{in} \ t) \\
\quad \text{if } x_i \text{ for } 1 \leq i \leq j-1 \text{ does not occur in } s_j, \dots, s_m, t \text{ and } m > 1 \\
(ldelcyc2) \quad (\mathbf{letrec} \ x_1 = s_1, \dots, x_m = s_m \ \mathbf{in} \ t) \rightarrow t \\
\quad \text{if } x_i \text{ for } 1 \leq i \leq m \text{ does not occur in } t
\end{array}$$

Here we show the correctness of (ldel).

**Lemma 4.1.** *A complete set of commuting diagrams for (ldel) is:*

$$- \xrightarrow{(ldel)} \cdot \xrightarrow{(st,a)} \rightsquigarrow \xrightarrow{(st,a)} \cdot \xrightarrow{(ldel)}$$

$$\begin{array}{c}
- \xrightarrow{(ldel)} \sim \xrightarrow{(st, lll^+)} \cdot \xrightarrow{ldel} \\
- \xrightarrow{(ldel)} \cdot \xrightarrow{(st, lll^+)} \sim \xrightarrow{(st, lll^*)} \cdot \xrightarrow{(ldel)}
\end{array}$$

As an example for computing commuting diagrams, we show one case: We write  $\backslash$  instead of  $\lambda$ .

*Example 4.2.* We compute the overlap of an (ldel)-redex and a standard (lapp)-redex. If the overlap is trivial, then it is not hard to see that the reductions commute, including the property “standard”.

In the case of a proper overlap, the redex and the corresponding reduction is as follows:

$$\begin{array}{c}
((\text{letrec } x = c \text{ in } \backslash y.y) \text{ d}) \xrightarrow{ldel} (\backslash y.y \text{ d}) \\
\text{On the other hand, if first the (lapp) rule is applied, then:} \\
((\text{letrec } x = c \text{ in } \backslash y.y) \text{ d}) \xrightarrow{st, lapp} (\text{letrec } x = c \text{ in } (\backslash y.y \text{ d})) \\
\xrightarrow{ldel} (\backslash y.y \text{ d})
\end{array}$$

This is covered by the rule  $\xrightarrow{(ldel)} \sim \xrightarrow{(st, lll^+)} \cdot \xrightarrow{ldel}$ .

**Lemma 4.3.** *A complete set of forking diagrams for (ldel) is:*

$$\begin{array}{c}
- \xleftarrow{(st, a)} \cdot \xrightarrow{(ldel)} \sim \xrightarrow{(ldel)} \cdot \xleftarrow{(st, a)} \\
- \xleftarrow{(st, lll^+)} \cdot \xrightarrow{(ldel)} \sim \xrightarrow{(ldel)} \\
- \xleftarrow{(st, lll^+)} \cdot \xrightarrow{(ldel)} \sim \xrightarrow{(ldel)} \cdot \xleftarrow{(st, lll^*)}
\end{array}$$

**Lemma 4.4.** *There are no infinite lll-reductions*

*Proof.* This can be shown by a natural-number valuation of expressions similar as in [Kut00], which is strictly decreasing in every reduction step.  $\square$

**Proposition 4.5.** *If  $s \xrightarrow{(ldel)} t$ , then  $s \sim_c t$ .*

*Proof.* (sketch)

Follows by induction on the length of reductions from the context lemma, and since there are no infinite (lll)-reduction sequences.  $\square$

## 5 Copying variables

This section contains the reduction (lcv) which is like compressing references used in **letrecs**. It can also be described as removing indirections.

$$\begin{array}{c}
(lcv) (\text{letrec } x = y, E \text{ in } C[x]) \rightarrow (\text{letrec } x = y, E \text{ in } C[y]) \\
(lcv) (\text{letrec } x_1 = y, x_2 = C[x_1], E \text{ in } t) \\
\rightarrow (\text{letrec } x_1 = y, x_2 = C[y], E \text{ in } t)
\end{array}$$

**Lemma 5.1.** *A complete set of commuting diagrams for (lcv) is:*

$$\begin{aligned}
& - \frac{(lcv)}{\longrightarrow} \cdot \frac{(st,a)}{\longrightarrow} \rightsquigarrow \frac{(st,a)}{\longrightarrow} \cdot \frac{(lcv)}{\longrightarrow} \\
& - \frac{(lcv)}{\longrightarrow} \cdot \frac{(st,cpn)}{\longrightarrow} \rightsquigarrow \frac{(st,cpn)}{\longrightarrow} \cdot \frac{(lcv)}{\longrightarrow} \cdot \frac{(lcv)}{\longrightarrow} \\
& - \frac{(lcv)}{\longrightarrow} \cdot \frac{(st,a)}{\longrightarrow} \rightsquigarrow \frac{(st,a)}{\longrightarrow}, \text{ where } a \in \{\text{case}, \text{cpn}, \text{ndr}, \text{ndl}\}.
\end{aligned}$$

**Lemma 5.2.** *A complete set of forking diagrams for (lcv) is:*

$$\begin{aligned}
& - \frac{(st,a)}{\longleftarrow} \cdot \frac{(lcv)}{\longrightarrow} \rightsquigarrow \frac{(lcv)}{\longrightarrow} \cdot \frac{(st,a)}{\longleftarrow} \\
& - \frac{(st,cpn)}{\longleftarrow} \cdot \frac{(lcv)}{\longrightarrow} \rightsquigarrow \frac{(lcv)}{\longrightarrow} \cdot \frac{(lcv)}{\longrightarrow} \cdot \frac{(st,cpn)}{\longleftarrow} \\
& - \frac{(st,a)}{\longleftarrow} \cdot \frac{(lcv)}{\longrightarrow} \rightsquigarrow \frac{(st,a)}{\longleftarrow} \text{ for } a \in \{\text{cp}, \text{case}, \text{ndl}, \text{ndr}\}.
\end{aligned}$$

**Proposition 5.3.** *If  $s \xrightarrow{(lcv)} t$ , then  $s \sim_c t$ .  
I.e., (lcv) is a correct program transformation in any context.*

The proof uses the context lemma, and the complete set of commuting and forking diagrams to meta-reduce reduction sequences.

## 6 Contextual equivalence of copy reductions

The required diagrams and the proof of correctness of non-standard copy reductions are complex. Only the complete set of commuting diagrams are presented. For this rule we require a special class of contexts: surface contexts: Surface contexts define expressions with holes not in the body of an abstraction.

**Definition 6.1.**

$$\begin{aligned}
S ::= & \square \mid (S \ E) \mid (E \ S) \mid (\text{case}_A \ S \ \text{alts}) \mid (\text{case}_A \ E \ \dots (p \rightarrow S) \dots) \\
& \mid (\text{choice } E \ S) \mid (\text{choice } S \ E) \\
& \mid ((\text{letrec } \dots \text{ in } S) \mid ((\text{letrec } \dots, x_i = S, \dots \text{ in } E))
\end{aligned}$$

where  $E$  stands for an expression.  $S$  is called surface context.

We consider the following atomic copy reductions:

$$\begin{aligned}
& (\text{cp}) \ (\text{letrec } x_1 = s_1, \dots, x_n = s_n \text{ in } C[x_1]) \\
& \quad \rightarrow (\text{letrec } x_1 = s_1, \dots, x_n = s_n \text{ in } C[s_1]) \\
& \quad \text{where } s_1 \text{ is an abstraction} \\
& (\text{cp}) \ (\text{letrec } x_1 = s_1, \dots, x_n = s_n \text{ in } s) \\
& \quad \rightarrow (\text{letrec } x_1 = s_1, \dots, x_j = C[s_1], \dots, x_n = s_n \text{ in } s) \\
& \quad \text{where } s_1 \text{ is an abstraction} \\
& \quad \text{and where } s_j \equiv C[x_1]
\end{aligned}$$

We distinguish the (cp)-reduction into two subreductions: If the target occurrence of the variable is in a surface context, then (cpt), otherwise it is a (cpd). Equivalently, it is a (cpd) iff the target variable is within an abstraction. Thus  $((\text{letrec } x = s, E \text{ in } D[\lambda z. C[x]])) \rightarrow ((\text{letrec } x = s, E \text{ in } D[\lambda z. C[s]]))$  is a reduction of type (cpd).

**Lemma 6.2.** *A complete set of commuting diagrams for (cpt), (cpd) is:*

$$\begin{array}{l}
- \frac{(i, cpt)}{\longrightarrow} . \frac{(st, a)}{\longrightarrow} \rightsquigarrow \frac{(st, a)}{\longrightarrow} . \frac{(\{i, st\}, cpt)}{\longrightarrow} \\
- \frac{(i, cpt)}{\longrightarrow} . \frac{(st, a)}{\longrightarrow} \rightsquigarrow \frac{(st, a)}{\longrightarrow}, \text{ where } a \in \{case, ndr, ndl\}. \\
- \frac{(i, cpd)}{\longrightarrow} . \frac{(st, a)}{\longrightarrow} \rightsquigarrow \frac{(st, a)}{\longrightarrow} . \frac{(i, cpd)}{\longrightarrow} \\
- \frac{(i, cpd)}{\longrightarrow} . \frac{(st, cpn)}{\longrightarrow} \rightsquigarrow \frac{(st, cpn)}{\longrightarrow} . \frac{(i, cpd)}{\longrightarrow} . \frac{(i, cpd)}{\longrightarrow} \\
- \frac{(i, cpd)}{\longrightarrow} . \frac{(st, a)}{\longrightarrow} \rightsquigarrow \frac{(st, a)}{\longrightarrow}, \text{ where } a \in \{case, ndr, ndl\}. \\
- \frac{(i, cpd)}{\longrightarrow} . \frac{(st, lbeta)}{\longrightarrow} \rightsquigarrow \frac{(st, lbeta)}{\longrightarrow} . \frac{(\{i, st\}, cpt)}{\longrightarrow}
\end{array}$$

This is sufficient to show that the (cp)-reductions retain contextual equivalence by a meta-reduction on reduction sequences.

**Proposition 6.3.** *If  $s \xrightarrow{(cp)} t$ , then  $s \sim_c t$ .  
I.e., (cp) is a correct program transformation in any context.*

In summary, we can prove:

**Theorem 6.4.** *All the reductions of the base calculus with the exception of (ndr), (ndl) are correct program transformations*

It is obvious that (ndr), (ndl) are not correct as program transformations, since (choice True False) may reduce to True, but True is not equivalent to (choice True False).

## 7 Unique Copy: Inlining

If a letrec-bound variable occurs only once, then it is possible to replace this variable by the bound expression and to remove the binding:

- (ucp)  $(\text{letrec } x = s, E \text{ in } C[x]) \rightarrow (\text{letrec } E \text{ in } C[s])$ , where  $C[]$  is a surface context,  $s$  arbitrary,  $x$  has exactly one occurrence in  $C[x]$  and no occurrence in  $E$  nor in  $s$ .
- (ucp)  $(\text{letrec } x = s \text{ in } C[x]) \rightarrow C[s]$ , where  $C[]$  is a surface context,  $s$  arbitrary, and  $x$  has exactly one occurrence in  $C[x]$  and no occurrence in  $s$ .
- (ucp)  $(\text{letrec } x = s, y = C[x], E \text{ in } t) \rightarrow (\text{letrec } y = C[s], E \text{ in } t)$ , where  $C[]$  is a surface context,  $s$  arbitrary,  $x$  has exactly one occurrence in  $C[x]$  and no occurrence in  $E, s$  and  $t$ .

Note that if  $s$  is an abstraction, then the rule is a combination of (cp) and (ldel).

**Lemma 7.1.** *A complete set of commuting diagrams for (ucp) is:*

$$\begin{array}{l}
- \frac{(ucp)}{\longrightarrow} . \frac{(st, a)}{\longrightarrow} \rightsquigarrow \frac{(st, a)}{\longrightarrow} . \frac{(ucp)}{\longrightarrow} \\
- \frac{(ucp)}{\longrightarrow} . \frac{(st, a)}{\longrightarrow} \rightsquigarrow \frac{(st, a)}{\longrightarrow} . \frac{(ldel)}{\longrightarrow} \text{ for } a \in \{case, ndr, ndl, cpn\} \\
- \frac{(ucp)}{\longrightarrow} . \frac{(st, a)}{\longrightarrow} \rightsquigarrow \frac{(st, a)}{\longrightarrow} \text{ for } a \in \{case, ndr, ndl\}
\end{array}$$

$$\begin{array}{c}
- \xrightarrow{(ucp)} . \xrightarrow{(st, ill^*)} \rightsquigarrow \xrightarrow{(st, ill^*)} . \xrightarrow{(ucp)} \\
\text{where the extreme case } \xrightarrow{(ucp)} \rightsquigarrow \xrightarrow{(ucp)} \text{ is excluded.} \\
- \xrightarrow{(ucp)} \rightsquigarrow \xrightarrow{(st, cpn)} . \xrightarrow{(ldel)}
\end{array}$$

**Proposition 7.2.** If  $s \xrightarrow{(ucp)} t$ , then  $s \sim_c t$ .  
*I.e.,  $(ucp)$  is a correct program transformation in any context.*

## 8 Conclusion

The rewriting based method of computing complete sets of commuting (resp. forking) diagrams to prove program transformations to be correct is demonstrated to be successful. We are able to show that all deterministic reduction rules in the rather complex lambda calculus  $\lambda_{ndlr}$  and also some other rules are correct. A general automatic method to compute diagrams by checking all non-trivial overlaps would be a valuable tool and deserves further research efforts.

## References

- Abr90. Samson Abramsky. The lazy lambda calculus. In D. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1990.
- ACCL91. M. Abadi, L. Cardelli, P.-L. Curien, and J.-J Lévy. Explicit substitutions. *J. functional programming*, 4(1):375–416, 1991.
- AF97. Z.M. Ariola and M Felleisen. The call-by-need lambda calculus. *J. functional programming*, 7(3):265–301, 1997.
- AFM<sup>+</sup>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.
- AK94. Z.M. Ariola and Jan Willem Klop. Cyclic lambda graph rewriting. In *LICS 94*, pages 416–425. IEEE Press, 1994.
- Bar84. H.P. Barendregt. *The Lambda Calculus. Its Syntax and Semantics*. North-Holland, Amsterdam, New York, 1984.
- KSS98. Arne Kutzner and Manfred Schmidt-Schauß. A nondeterministic call-by-need lambda calculus. In *International Conference on Functional Programming 1998*, pages 324–335. ACM Press, 1998.
- Kut99. Arne Kutzner. A non-deterministic call-by-need lambda-calculus with erratic choice: operational semantics, program transformations and applications. draft of thesis, 1999.
- Kut00. Arne Kutzner. *Ein nichtdeterministischer call-by-need Lambda-Kalkül mit erratic choice: Operationale Semantik, Programmtransformationen und Anwendungen*. Dissertation, J.W.Goethe-Universität Frankfurt, 2000. in german.
- Mes00. José Meseguer. Rewriting Logic and Maude: concepts and applications. In Leo Bachmair, editor, *Proceedings RTA'2000*, pages 1–26. Springer-Verlag, 2000.
- MOW98. John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. *J. of Functional programming*, 8:275–317, 1998.

- MSC99. A.K.D. Moran, D. Sands, and M. Carlsson. Erratic fudgets: A semantic theory for an embedded coordination language. In *Coordination '99*, volume 1594 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.
- MST96. Ian Mason, Scott F. Smith, and Carolyn L. Talcott. From operational semantics to domain theory. *Information and Computation*, 128:26–47, 1996.
- Pit97. Andrew D. Pitts. Operationally-based theories of program equivalence. In *Semantics and Logics of Computation*. Cambridge University Press, 1997.
- Smi92. S.F. Smith. From operational to denotational semantics. In *7th International Conference MFPS*, LNCS 598, pages 54–76. Springer-Verlag, 1992.
- SS00. M. Schmidt-Schauß. FUNDIO: A lambda-calculus with a recursive let, case, constructors, and an IO-interface, 2000. draft.

(lbeta)	$((\lambda x.s) t) \rightarrow (\text{letrec } x = t \text{ in } s)$
(cpn)	$(\text{letrec } x_1 = s_1, x_2 = x_1, \dots, x_j = x_{j-1}, x_{j+1} = s_{j+1} \dots \text{ in } C[x_j])$ $\rightarrow (\text{letrec } x_1 = s_1, x_2 = x_1, \dots, x_j = x_{j-1}, x_{j+1} = s_{j+1} \dots \text{ in } C[s_1])$ where $s_1$ is an abstraction
(cpn)	$(\text{letrec } x_1 = s_1, x_2 = x_1, \dots, x_j = x_{j-1}, x_{j+1} = C[x_j], \dots \text{ in } s)$ $\rightarrow (\text{letrec } x_1 = s_1, x_2 = x_1, \dots, x_j = x_{j-1}, x_{j+1} = C[s_1], \dots \text{ in } s)$ where $s_1$ is an abstraction
(llet)	$(\text{letrec } x_1 = s_1, \dots, x_n = s_n \text{ in } (\text{letrec } y_1 = t_1, \dots, y_m = s_m \text{ in } r))$ $\rightarrow (\text{letrec } x_1 = s_1, \dots, x_n = s_n, y_1 = t_1, \dots, y_m = s_m \text{ in } r)$
(llet)	$(\text{letrec } x_1 = s_1, \dots, x_i =$ $\quad (\text{letrec } y_1 = t_1, \dots, y_m = t_m \text{ in } s_i), \dots, x_n = s_n \text{ in } r)$ $\rightarrow (\text{letrec } x_1 = s_1, \dots, x_n = s_n, y_1 = t_1, \dots, y_m = s_m \text{ in } r)$
(lapp)	$((\text{letrec } x_i = t_i \text{ in } t) s) \rightarrow (\text{letrec } x_i = t_i \text{ in } (t s))$
(lcase)	$(\text{case}_A (\text{letrec } E \text{ in } t) \text{ alts}) \rightarrow (\text{letrec } E \text{ in } \text{case}_A t \text{ alts})$
(case)	$(\text{case}_A (c_{A,i} t_1 \dots t_n) \dots ((c_{A,i} y_1 \dots y_n) \rightarrow t) \dots)$ $\rightarrow (\text{letrec } y_1 = t_1 \dots y_n = t_n \text{ in } t)$  $\text{letrec } x_1 = (c_{A,i} t_1 \dots t_{j_1}),$ $\quad x_2 = x_1 t_{j_1+1} \dots t_{j_2},$ $\quad \dots$ $\quad x_m = x_{m-1} t_{j_{m-1}+1} \dots t_{j_m},$ $\quad \dots$ $\quad C[\text{case}_A (x_m t_{j_m+1} \dots t_{j_{m+1}}) \dots ((c_{A,i} z_1 \dots z_n) \rightarrow t)]$ $\longrightarrow$ $\text{letrec } x_1 = (c_{A,i} y_1 \dots y_{j_1}), y_1 = t_1, \dots, y_{j_1} = t_{j_1},$ $\quad x_2 = x_1 y_{j_1+1} \dots y_{j_2}, y_{j_1+1} = t_{j_1+1}, \dots, y_{j_2} = t_{j_2},$ $\quad \dots$ $\quad x_m = x_{m-1} y_{j_{m-1}+1} \dots y_{j_m}, y_{j_{m-1}+1} = t_{j_{m-1}+1}, \dots, y_{j_m} = t_{j_m},$ $\quad \dots$ $\quad C[(\text{letrec } y_{j_m+1} = t_{j_m+1}, \dots, y_{j_{m+1}} = t_{j_{m+1}}, z_1 = y_1, \dots, z_n = y_n \text{ in } t)]$ where $n = j_m$ and the case-expression may be in a bound or in the in-expression and where $y_i$ are fresh variables
(ndl)	$(\text{choice } s t) \rightarrow s$
(ndr)	$(\text{choice } s t) \rightarrow t$

**Fig. 1.** Reduction rules of  $\lambda_{ndlr}$