International School on Rewriting (ISR 2015)

Rewriting Techniques for Correctness of Program Transformations

David Sabel and Manfred Schmidt-Schauß

Computer Science Institute
Goethe-University Frankfurt am Main
{sabel,schauss}@ki.informatik.uni-frankfurt.de

last update of these notes: 6th August 2015

Abstract

The course gives an introduction to program transformations, their correctness and their relation to rewriting. The course focuses on small-step operational semantics of functional programming languages which can be seen as higher-order rewriting with a fixed strategy. On top of the operational semantics contextual equivalence is used as the main notion of program semantics and correctness of program transformations. Several proof techniques are presented show correctness of program transformations. Among them are context lemmas, a diagram-based proof technique, and induction schemes for proving transformations in the context of list-processing functions. Also approaches for the automation of the techniques using termination provers for term rewrite systems are discussed. The course also briefly treats non-deterministic and concurrent extensions of functional programming languages and adaptions of the contextual semantics.

Contents

1 Introduction

1.1 Motivation

Verification and validation of properties of programs, as well as optimizing, translating and compiling programs can benefit from the application of rewriting techniques.

Source-level program transformations are used in compilation to simplify and optimize programs, in code refactoring to improve the design of programs; and in software verification and code validation, program transformations are used to translate and simplify programs into forms suitable for specific verification purposes or tests. In all these settings correctness of the program transformation is an indispensable requirement. Here, correctness of program transformations is understood as the preservation of the meaning of the programs.

The correctness of program transformations for pure, sequential, and basic models of functional programming languages are well understood and there are a lot of sophisticated proof techniques like e.g. using denotational models, operational semantics and type-based reasoning. The situation changes when models of more realistic functional programming languages including features like side-effecting computations, lazy evaluation with sharing, and non-determinism and concurrency are taken into account. On the one hand notions of correctness of program transformation have to adapted and on the other hand new techniques to prove the correctness are required.

A very natural and general notion of semantic equality of programs is contextual equivalence. Two programs are contextually equivalent if exchanging the programs in any surrounding program (the context) is not observable. Depending on the setting (deterministic or non-deterministic) the observed behavior has to be adapted.

The objective of the course is to introduce methods and techniques to prove correctness of program transformations for functional programming languages extended by constructs of realistic languages like e.g. provided by Concurrent Haskell, the extension of Haskell by concurrency. Starting from a formal description of core models of such languages by defining their syntax and their operational small-step semantics we will introduce the notion of contextual equivalence and illustrate its generality and usefulness for reasoning about the correctness of program transformations.

Several program transformations will be taken into account including partial evaluation and laws on list processing functions. Also correctness of transformations in impure, concurrent, and non-deterministic languages will be briefly discussed.

The main proof techniques presented in the course are context lemmata, inductive methods and the so called diagram based method, which is a syntactic method for proving correctness of program transformations similar (but with important differences) to computing critical pairs in term rewriting.

To stimulate more research on program transformations using rewriting techniques the course intends to provide the basic techniques on proving correctness of program transformation and to give an overview of the state of art of this research topic.

1.2 Outline

In Section 2 we recall the lazy lambda calculus and discuss several extensions of it to derive the core language *LNAME* which is a basic model for the core language of purely function programming languages. After introducing the syntax and the operational semantics (in form of a call-by-name small-step reduction relation) of *LNAME*, several techniques for proving correctness of program transformations are presented. Among them are a context lemma, a diagram-based proof technique and its automation using termination provers for term rewrite systems, and induction schemes for proving correctness of program transformations for list processing expressions.

In Section 3 the calculus LR is introduced which uses a call-by-need evaluation which implements lazy evaluation with sharing. The calculus LR has so-called `letrec`-expressions to represent sharing and recursive functions. Again the correctness of program transformations and the diagram-based proof technique are analyzed for LR.

Finally, in Section 4 a basic extension of LR by non-determinism is taken into account. The section focuses on the definition of contextual equivalence in the non-deterministic setting and discusses necessary adaptions of the introduced proof techniques.

At several places, if appropriate, the use of rewriting techniques is mentioned and commented on explicitly. Also the relation between program evaluation and rewriting will be discussed.

2 A Functional Call-by-Name Core Language

In this section we will introduce a core language that models the functional part of lazy functional programming languages like e.g. Haskell [13]. We introduce the language step-wise by starting with the lazy lambda calculus (see [1]) and then add constructs which are required or make reasoning in the language more comfortable. After introducing the syntax and the operational semantics, we will consider program equivalences and analyze the correctness of program transformations which can be used as optimizations in the compilation of such functional languages or also for equational reasoning in automated deduction systems. A special focus will be proving equational laws for list processing functions. Our notion of program equivalence is contextual equivalence (see e.g. [17, 20]) which is a very general notion and can be used for various programming languages.

2.1 The Lazy Lambda Calculus

Let us first briefly consider the pure lazy lambda calculus to introduce some notions and notations.

Let *Var* be a countable infinite set of *variables*. We denote variables with lower-case letters from the end of the alphabet, i.e. w, x, y, z, sometimes indexed with some natural number.

The syntax of expressions $e \in E_\lambda$ of the pure lambda calculus is defined by the grammar

$$e, e_i \in E_\lambda ::= x \mid \lambda x.e \mid (e_1\ e_2)$$

where $x \in Var$. The construct $\lambda x.e$ is called an *abstraction* and it represents an anonymous function, where the formal parameter x is bound with scope e. For instance, $\lambda x.x$ is the identity function and $\lambda x.\lambda y.x$ is the function that receives two arguments and maps them to the first argument. As an abbreviation we will often write $\lambda x_1, \ldots, x_n.e$ instead of $(\lambda x_1.\lambda x_2. \ldots \lambda x_n.e)$ The construct $(e_1\ e_2)$ is called an *application*, i.e. expression e_1 is applied to the expression e_2. For instance, $((\lambda x.x)\ (\lambda y.y))$ applies the identity function to the identity function, and $(((\lambda x.\lambda y.x)\ (\lambda w_1.w_1))\ (\lambda w_2.w_2))$ is a nested application which applies $(\lambda x.\lambda y.x)$ to two instances of the identity function. To ease notation, we use the following precedence rules: The body e of an abstraction $\lambda x.e$ is as large as possible, e.g. $\lambda x.x\ x$ denotes $\lambda x.(x\ x)$ (instead of $((\lambda x.x)\ x)$)). We assume that application is left-associative, i.e. $(e_1\ e_2\ e_3)$ abbreviates $((e_1\ e_2)\ e_3)$ (instead of $(e_1\ (e_2\ e_3))$).

A *context* C is an expression with a (single) hole $[\cdot]$, i.e. the contexts C_λ of the pure lambda calculus can be defined by the grammar

$$C \in C_\lambda ::= [\cdot] \mid \lambda x.C \mid (C\ e) \mid (e\ C)$$

With $C[e]$ we denote the expression which evolves from replacing the hole $[\cdot]$ in C by expression e. For instance, let $C = \lambda x.\lambda y.[\cdot]$ and $e = \lambda w.y$, then $C[e]$ denotes the expression $\lambda x.\lambda y.\lambda w.y$.

Since variables become bound by abstractions, there is a notion for *free* and *bound* variables: For an expression $e \in E_\lambda$, its *free variables* $FV(e)$ and its *bound variables* $BV(e)$ are inductively defined by

$$\begin{array}{lcllcl} FV(x) & := & \{x\},\ \text{if } x \in Var & BV(x) & := & \emptyset,\ \text{if } x \in Var \\ FV((e_1\ e_2)) & := & FV(e_1) \cup FV(e_2) & BV((e_1\ e_2)) & := & BV(e_1) \cup BV(e_2) \\ FV(\lambda x.e) & := & FV(e) \setminus \{x\} & BV(\lambda x.e) & := & BV(e) \cup \{x\} \end{array}$$

▶ **Example 2.1.** For the expression $e = (\lambda x.\lambda y.\lambda w.(x\ y\ z))\ x$, the free variables are $FV(e) = \{x, z\}$ and the bound variables are $BV(e) = \{x, y, w\}$.

We say that an expression e is *closed* if $FV(e) = \emptyset$. An occurrence of a variable x in an expression e (i.e. $e = C[x]$ for some context C) is called *free*, if the occurrence of x is not in the body of an abstraction $\lambda x.e'$, and otherwise the occurrence is called *bound* (i.e. $C = C_1[\lambda x.C_2[\cdot]]$ for some contexts C_1, C_2).

▶ **Example 2.2.** In the expression $(x\ \lambda x.x)$ there are two occurrences of x (the first one in the hole of the context $([\cdot]\ \lambda x.x)$ and the second one in the hole of the context $(x\ \lambda x.[\cdot])$ The first one is a free occurrence, while the second one is a bound occurrence.

As usual it is allowed to consistently rename the bound variables of an expression, since we do not want to distinguish expressions up to a renaming of bound variables. A single renaming step can be performed by α-renaming,

$$\lambda x.e \xrightarrow{\alpha} \lambda x'.e[x'/x],\ \text{if } x' \notin FV(e) \text{ and } x' \notin BV(e)$$

and where α-equivalence $=_\alpha$ is the smallest congruence obtained from $\xrightarrow{\alpha}$, i.e. it is inductively defined as

$$\begin{array}{rcl} e_1 & =_\alpha & e_2,\ \text{if } e_1 \xrightarrow{\alpha} e_2 \\ e & =_\alpha & e \\ e_1 & =_\alpha & e_2,\ \text{if } e_2 =_\alpha e_1 \\ e_1 & =_\alpha & e_3,\ \text{if } e_1 =_\alpha e_2 \wedge e_2 =_\alpha e_3 \\ C[e_1] & =_\alpha & C[e_2],\ \text{if } e_1 =_\alpha e_2 \end{array}$$

In the following we do not distinguish between α-equivalent expressions. We use the *distinct variable convention*, which assumes that in any expression the free variables are different from the bound variables, and that all variables at binders are pairwise different. Note that the distinct variable convention can always be fulfilled by α-renamings.

Using this convention, we can define a more general notion of substitution in an easy way: With $e_1[e_2/x]$ we denote the *substitution* of all free occurrences of variable x in e_1 by the expression e_2, i.e. substitution can be inductively defined as

$$\begin{array}{lcl} x[e/x] & := & e \\ y[e/x] & := & y,\ \text{if } x \neq y \\ (\lambda x.e_1)[e/x] & := & \lambda x.e_1 \\ (\lambda y.e_1)[e/x] & := & \lambda y.(e_1[e/x]), \text{if } x \neq y \\ (e_1\ e_2)[e/x] & := & (e_1[e/x]\ e_2[e/x]) \end{array}$$

▶ **Example 2.3.** The expressions $\lambda x.\lambda y.(x\ y)$ and $\lambda x'.\lambda y'.(x'\ y')$ are α-equivalent. The expression $\lambda x.(x\ \lambda x.(x\ y))$ does not fulfill the distinct variable convention, since x is bound twice. An α-equivalent expression that fulfills the distinct variable convention is the expression $\lambda z.(z\ \lambda x.(x\ y))$. The expression $(x\ \lambda x.x)$ also does not fulfill the distinct variable convention, since $FV(x\ \lambda x.x) \cap BV(x\ \lambda x.x) = \{x\}$. An α-equivalent expression that fulfills the distinct variable convention is $(x\ \lambda y.y)$. Note that $(y\ \lambda x.x)$ is *not* an α-equivalent expression, since it is not allowed to rename free (occurrences) of variables.

▶ **Exercise 2.4.** Let e be the expression $(\lambda y.(y\ x))\ (\lambda x.(x\ y))\ (\lambda z.(z\ x\ y))$. Calculate the sets $FV(e)$ and $BV(e)$. Which occurrences of the variables x, y, z are free, which are bound? Rename the expression e into an α-equivalent expression which fulfills the distinct variable convention.

The operational semantics of a language tells us how to evaluate programs, i.e. how to obtain the result of a program. In functional languages (and thus also in the lambda calculus) the result of the evaluation is a single expression, i.e. some kind of normal form. The operational semantics can be defined by (higher-order) rewrite rules where usually an additional strategy is required to make evaluation unique and deterministic. The main rewrite rule of the lambda calculus is β-reduction, which defines how to evaluate the application of an abstraction to an argument:

$$(\lambda x.e_1)\ e_2 \xrightarrow{\beta} e_1[e_2/x]$$

For example, $((\lambda x.\lambda y.x)\ (\lambda w_1.w_1)) \xrightarrow{\beta} (\lambda y.x)[(\lambda w_1.w_1)/x] = (\lambda y.\lambda w_1.w_1)$. We will write $\xrightarrow{C,\beta}$ for the relation that allows to apply β-reductions in any context, i.e. whenever $e_1 \xrightarrow{\beta} e_2$ then for every context $C \in C_\lambda$: $C[e_1] \xrightarrow{C,\beta} C[e_2]$. Using $\xrightarrow{C,\beta}$ as operational semantics has the drawback that it is not deterministic, e.g. in $(\lambda x.x)((\lambda y.y)(\lambda z.z))$ there are two possibilities for a $\xrightarrow{C,\beta}$-reduction.

Hence, to fix the (lazy) evaluation strategy (called *normal order reduction*) we first define *reduction contexts* R_λ, which are those contexts, where the hole is not in the body of an abstraction and only in the function position of an application:

$$R \in R_\lambda ::= [\cdot] \mid (R\ e)$$

A *normal order reduction step* $\xrightarrow{no}$ is any β-reduction which is performed inside a reduction context, i.e.

$$\text{If } e_1 \xrightarrow{\beta} e_2, \text{ then for every reduction context } R \in R_\lambda\text{: } R[e_1] \xrightarrow{no} R[e_2].$$

Given a normal order reduction step $R[(\lambda x.e_1)\ e_2] \xrightarrow{no} R[e_1[e_2/x]]$, the expression $(\lambda x.e_1)\ e_2$ together with its position in $R[(\lambda x.e_1)\ e_2]$ is called the *normal order redex*. It is not hard to verify that normal order reduction is unique, i.e. for any expression e either there is exactly one expression e' (up-to α-equivalence), s.t. $e \xrightarrow{no} e'$ or e is irreducible w.r.t. normal order reduction.

▶ **Exercise 2.5.** Let $e_1 = (\lambda w.w)\ (\lambda x.x)\ ((\lambda y.((\lambda u.y)\ y))\ (\lambda z.z))$. Write down all reduction contexts $R \in R_\lambda$ and expressions e_2 s.t. $R[e_2] = e_1$. Apply a normal order reduction step to e_1.

For a relation $\rightarrow$, we write $\xrightarrow{+}$ for the transitive closure of $\rightarrow$, $\xrightarrow{*}$ for the reflexive-transitive closure, and $\xrightarrow{i}$ for exactly i $\rightarrow$-steps. This will also be use for normal order reduction $\xrightarrow{no}$, written as $\xrightarrow{no,*}$, $\xrightarrow{no,+}$, and $\xrightarrow{no,i}$, respectively.

For closed expressions, the normal forms w.r.t. normal order reduction are so-called *weak head normal forms* (WHNFs) which are exactly all abstractions. This is the usual notion for a successful program in lazy functional programming languages. Reasons not to evaluate in the body of abstractions are that it would forbid an efficient implementation using graph reduction (see e.g. [19]), and that there is no real information gain due to such reductions from a user's view.

For open expressions, also expressions of the form $R[x]$, where x is free in R and R is a reduction context, are irreducible w.r.t. normal order reduction. However, we will treat such programs not as successful, but as unsuccessful stuck expressions.

An equivalent definition of normal order reduction can be given by using a *labeling algorithm* to find the normal order redex: Let $\star$ be a label. Then the label shifting rule

$$(e_1\ e_2)^\star \to (e_1^\star\ e_2)$$

is sufficient to find the normal order redex, that is, for an expression e, start with $e^\star$ and apply the shifting rule as long as possible. Then there are three cases:

- The labeling ends with $R[((\lambda x.e_1)^\star\ e_2)]$. Then the normal order reduction is $R[(\lambda x.e_1)\ e_2] \xrightarrow{no} R[e_1[e_2/x]]$.
- The labeling ends with $(\lambda x.e_1)^\star$. Then the expression e is a WHNF.
- The labeling ends with $R[x^\star]$. Then e is not a WHNF, but no normal order reduction is applicable to e.

▶ **Example 2.6.** We consider the expression $((\lambda x.\lambda y.x)((\lambda w.w)(\lambda z.z)))$. The labeling algorithm proceeds as follows:

$$((\lambda x.\lambda y.x)((\lambda w.w)(\lambda z.z)))^\star \to ((\lambda x.\lambda y.x)^\star((\lambda w.w)(\lambda z.z)))$$

Now a normal order reduction is applicable, since the subexpression marked with $\star$ is an abstraction which is applied to an argument, i.e.

$$((\lambda x.\lambda y.x)((\lambda w.w)(\lambda z.z))) \xrightarrow{no} \lambda y.((\lambda w.w)(\lambda z.z))$$

Applying the labeling algorithm again, results in $(\lambda y.((\lambda w.w)(\lambda z.z)))^\star$, which shows that the expression is a WHNF and no more normal order reduction steps are applicable.

▶ **Exercise 2.7.** Evaluate the following expression in normal order, i.e. apply normal order reduction steps until no more normal order reductions are applicable.

$$(\lambda y.(\lambda x.y\ (x\ x)))\ (\lambda.y_1.(\lambda x_1.y_1\ (x_1\ x_1)))(\lambda w.\lambda z.w)$$

Using normal order reduction we define the notion that an expression converges:

▶ **Definition 2.8.** An expression $e \in E_\lambda$ *converges* (or *successfully terminates*) (written as $e{\Downarrow}$) iff there exists a sequence of normal order reductions starting with e and ending in a WHNF, i.e. $e{\Downarrow}$ iff $e \xrightarrow{no,*} e'$ where e' is a WHNF. We sometimes also write $e{\Downarrow}e'$ in this case. If $e{\Downarrow}$ does not hold, then we say that e *diverges* and write $e{\Uparrow}$.

Note that there are (open) expressions which do not have an infinite normal order reduction sequence, but they diverge. For instance, the expression $(\lambda x.(y\ x))\ (\lambda w.w)$ reduces to $y\ (\lambda w.w)$ with one normal order reduction, and the contractum $y\ (\lambda w.w)$ is irreducible w.r.t. normal order reduction, but it is not a WHNF and thus $(\lambda x.(y\ x))\ (\lambda w.w){\Uparrow}$.

▶ **Example 2.9.** The expression $((\lambda x.\lambda y.x)\ (\lambda z.z)\ (\lambda w.w))$ converges, since it reduces with two normal order reduction steps to a WHNF:

$$((\lambda x.\lambda y.x)\ (\lambda z.z)\ (\lambda w.w)) \xrightarrow{no} (\lambda y.\lambda z.z)\ (\lambda w.w) \xrightarrow{no} \lambda z.z.$$

The expression $(\lambda x.(x\ x))\ (\lambda y.(y\ y))$ diverges, since it has an infinite normal order reduction $(\lambda x.(x\ x))\ (\lambda y.(y\ y)) \xrightarrow{no} (\lambda y_1.(y_1\ y_1))\ (\lambda y_2.(y_2\ y_2)) \xrightarrow{no} \ldots$.

The following *standardization property* holds for normal order reduction, which shows that (restricted to the notion of convergence) normal order reduction is an optimal strategy:

▶ **Proposition 2.10.** *Let e be an expression, s.t. $e \xrightarrow{C,\beta,*} e'$ where e' is a WHNF. Then $e{\Downarrow}$.*

Finally, we introduce a notion of program semantics. We use the notion of *contextual equivalence* which considers expressions as equal if they cannot be distinguished when they are used as subexpressions in any other surrounding program. It is defined as the symmetrization of the *contextual preorder*:

▶ **Definition 2.11.** Let $e_1, e_2 \in E_\lambda$. Then e_2 *contextually approximates* e_1, written as $e_1 \leq_c e_2$ (or alternatively $e_2 \geq_c e_1$), iff for all contexts $C \in C_\lambda$: $C[e_1]{\Downarrow} \implies C[e_2]{\Downarrow}$.

The relation $\leq_c$ is called the *contextual approximation* or alternatively the *contextual preorder*. The expressions e_1 and e_2 are *contextually equivalent*, written as $e_1 \sim_c e_2$, iff $e_1 \leq_c e_2$ and $e_2 \leq_c e_1$ hold. The relation $\sim_c$ is called *contextual equivalence*.

Note that the definition of contextual approximation and contextual equivalence is very general, since it can be applied to any program calculus which is equipped with a notion of expressions, contexts, and convergence. Note also that it is usually sufficient to consider convergence as the observation in the definition of contextual equivalence, since due to the universal quantification over all contexts, different expressions or values can easily be distinguished[1]. For example, the expressions $\lambda x.\lambda y.x$ and $\lambda x.\lambda y.y$ are not contextually equivalent, since for the context $C := ([\cdot])\ (\lambda z.z)\ ((\lambda u_1.u_1\ u_1)\ (\lambda u_2.u_2\ u_2))$ we have $C[\lambda x.\lambda y.x]{\Downarrow}$ while $C[\lambda x.\lambda y.y]{\Uparrow}$.

Contextual preorder is a precongruence (i.e. a preorder that is stable w.r.t. substitution into contexts), and contextual equivalence is a congruence (i.e. an equivalence relation that is stable w.r.t. substitution into contexts)[2]. Every usable notion of program equivalence (and semantics) should be a congruence, since transforming contextually equivalent subprograms guarantees an unchanged semantics of the whole program, which for instance is important during equational reasoning and for local program optimizations applied in compilers.

We end our overview of the lazy lambda calculus by introducing some specific expressions:

▶ **Example 2.12.** Some prominent expressions of the lambda calculus are:

$$\begin{array}{lcl} I & := & \lambda x.x \\ K & := & \lambda x.\lambda y.x \\ K_2 & := & \lambda x.\lambda y.y \\ \Omega & := & (\lambda x_1.(x_1\ x_1))\ (\lambda x_2.(x_2\ x_2)) \\ Y & := & \lambda y_1.(\lambda x_1.(y_1\ (x_1\ x_1)))\ (\lambda y_2.(y_2\ (x_2\ x_2))) \end{array}$$

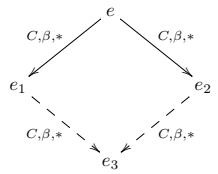
[1] Exceptions are non-deterministic languages (see Section 4), or languages where every expression converges (e.g. the simply-typed lambda calculus).

[2] As a side note, we remark that the definition of a (pre-)congruence in mathematics is sometimes different: A preorder $\approx \subseteq (D \times D)$ (equivalence relation, resp.) is a precongruence (congruence, resp.) if $\approx$ is stable w.r.t. application of all functions (i.e. for all functions $f :: D \to D : t_1 \approx t_2 \implies f(t_1) \approx f(t_2)$). Our definition is similar, but only considers contexts as functions and thus only syntactically expressible "functions" are taken into account.

The I-combinator is the identity function. The K-combinator receives to arguments and maps them to the first argument, while the K_2-combinator maps them to the second argument. Ω is a diverging expression, which has an infinite normal order reduction. The Y-combinator is a *fixpoint-combinator*, it has the property that $Y\ e \sim_c e\ (Y\ e)$ holds. It can be used to express recursion.

2.1.1 Remarks on Connections to Rewriting and Confluence

It is well-known [4] that the reduction $\xrightarrow{C,\beta,*}$ in the lambda-calculus is confluent. Confluence is the following property of a (reduction) relation: For all expressions e, e_1, e_2: if $e \xrightarrow{C,\beta,*} e_1$ and $e \xrightarrow{C,\beta,*} e_2$, then there is some e_3, such that $e_1 \xrightarrow{C,\beta,*} e_3$ as well as $e_2 \xrightarrow{C,\beta,*} e_3$ (modulo α-equivalence). In diagram notation:



This confluence result is helpful for proving that β-reduction does not change contextual equivalence (i.e. β-reduction is a correct program transformation):

▶ **Theorem 2.13.** *$\xrightarrow{\beta}$ is correct w.r.t. $\sim_c$, i.e. if $e_1 \xrightarrow{C,\beta} e_2$ then $e_1 \sim_c e_2$.*

Proof. Let $e_1 \xrightarrow{\beta} e_2$ and let C be a context such that $C[e_2]{\Downarrow}e_3$. Then $C[e_1] \xrightarrow{C,\beta,*} e_3$. Then the standardization (Proposition 2.10) shows that $C[e_1]{\Downarrow}$.

If $C[e_1]{\Downarrow}e_3$, then confluence shows that $e_3, C[e_2]$ have a common successor e_4, which must be an abstraction, since e_4 is a reduction result of e_3. Then the standardization (Proposition 2.10) shows that $C[e_2]{\Downarrow}$. ◀

With $\xleftrightarrow{C,\beta,*}$ we denote the *conversion relation* that consists of a finite sequence of beta-reduction in both directions. This relation is compatible with contexts.

Without proof we state the following lemma (it can later be proved using the context lemma).

▶ **Lemma 2.14.** *Let e_1, e_2 be two closed expressions with $e_1{\Uparrow}$ and $e_2{\Uparrow}$. Then $e_1 \sim_c e_2$.*

The conversion relation is too weak as an equivalence in the lazy lambda calculus:

▶ **Proposition 2.15.** *$\xleftrightarrow{C,\beta,*} \subseteq \sim_c$, but $\xleftrightarrow{C,\beta,*} \neq \sim_c$.*

Proof. $\xleftrightarrow{C,\beta,*} \subseteq \sim_c$ immediately follows from Proposition 2.13. The inequality follows since for example $(\lambda x.(x\ x\ x)\ (\lambda x.(x\ x\ x)){\Uparrow}$, and hence by Lemma 2.14, $\Omega \sim_c (\lambda x.(x\ x\ x))\ (\lambda x.(x\ x\ x))$. But, there is no common reduction successor of Ω and the expression $(\lambda x.(x\ x\ x))\ (\lambda x.(x\ x\ x))$. ◀

The η-transformation is defined as $e \xrightarrow{\eta} \lambda x.e\ x$ if x is not free in e. The η-transformation is not correct (i.e. it does not always preserve contextual equivalence), since e.g.$\Omega \xrightarrow{\eta} \lambda x.\Omega\ x$, but $\Omega{\Uparrow}$, whereas $\lambda x.\Omega\ x{\Downarrow}$. This property is typical for the lazy lambda calculus.

2.2 Extending the Lazy Lambda Calculus – *LNAME*

Although we have introduced the lazy lambda calculus, it is not a good model for lazy functional programming languages like Haskell. The main reason is that the syntactic possibilities of the lambda calculus are very restricted. One obvious reason is that it is hard to program in the lambda calculus. More formal reasons will be given in the subsequent subsections. Hence, we will introduce primitives for missing programming constructs. We will explain each new construct (a little bit informally) and its corresponding reduction rule. Finally, we combine all of the constructs in the definition of the language *LNAME*.

2.2.1 Data Constructors and Case-Expressions

Clearly, in programming languages one needs constructs to express data, like numbers, Booleans, lists, etc., and also corresponding functions to compare and distinguish different data values. In the pure lazy lambda calculus, we cannot easily express data. It is possible to encode data values and selector functions by using the so-called Church-encoding, but this approach is very tedious and there is also a formal argument which shows that adding data values is not only syntactic sugar, which we will give at the end of this section in Remark 2.21.

So let us assume that there is a finite nonempty set of *type constructors* $\mathcal{T}$, where for every $T \in \mathcal{T}$ there are pairwise disjoint finite nonempty sets of data constructors $D_T = \{c_{T,1}, \ldots c_{T,|T|}\}$. Every constructor has a fixed *arity* (a non-negative integer) denoted by $ar(T)$ or $ar(c_{T,j})$, resp. Data constructors are used to construct data, i.e. a data constructor $c_{T,i}$ of arity n is applied to n expressions to form a *constructor application* $(c_{T,i}\ e_1 \ldots e_{ar(c_{T,i})})$.

▶ **Example 2.16.** Examples are a type constructor *Bool* (of arity 0) with data constructors `True` and `False` (both of arity 0), as well as lists with a type constructor *List* (of arity 1) and data constructors `Nil` (of arity 0) and `Cons` (of arity 2). For example, a list consisting of three Boolean values can be written by nested constructor applications as (`Cons True` (`Cons False` (`Cons True Nil`))).

▶ **Exercise 2.17.** Define a type constructor together with its data constructors to represent pairs. Write down a list of three pairs of Boolean values in our notation.

Note that constructor applications are allowed only to occur fully saturated[3]. Constructor applications are treated as values (or more precisely as *constructor weak head normal forms* (CWHNFs)) and thus constructor applications are not (deeply) evaluated unless their evaluation is forced. Thus, we need a further construct to evaluate constructor applications.

For deconstructing constructor applications and to distinguish between the different constructors of the same type we introduce `case`-*expressions* where there is a $\texttt{case}_T$-construct for every type constructor $T \in \mathcal{T}$. The syntax of a `case`-expression is

$$\texttt{case}_T\ e\ \texttt{of}\ \{(c_{T,1}\ x_{1,1}\ \ldots\ x_{1,ar(c_{T,1})}) \to e_1; \ldots; (c_{T,|T|}\ x_{|T|,1}\ \ldots\ x_{|T|,ar(c_{T,|T|})}) \to e_{|T|}\}$$

The construct $(c_{T,i}\ x_{i,1}\ \ldots\ x_{i,ar(c_{T,i})}) \to e_i$ is called a `case`-*alternative*, which consists of a *pattern* $(c_{T,i}\ x_{i,1}\ \ldots\ x_{i,ar(c_{T,i})})$ and a right hand side e_i which is an arbitrary expression.

[3] Partial applications $(c_{T,i}\ e_1\ \ldots\ e_m)$ where $m < ar(c_{T,i})$ can be represented as abstractions $\lambda x_{m+1}, \ldots, x_{ar(c_{T,i})}.(c_{T,i}\ e_1\ \ldots\ e_m\ x_{m+1}\ \ldots\ x_{ar(c_{T,i})})$

The variables $x_{i,1},\ldots,x_{i,ar(c_{T,i})}$ in the case-pattern become bound by the pattern where the binding scope is the expression e_i.

Note that only variables are allowed as arguments in patterns, and note also that there is exactly one `case`-alternative for every data-constructor $c_{T,i} \in D_T$.

▶ **Example 2.18.** Using `case`-expressions we can express a conditional expression

$$\texttt{case}_{Bool}\ e_1\ \texttt{of}\ \{\texttt{True} \to e_2; \texttt{False} \to e_3\}.$$

It has the same meaning as `if` e_1 `then` e_2 `else` e_3 and thus there is no need to add `if-then-else`-expressions to our core language.

As a further example we can define an abstraction which computes the first element of a list (and diverges for the empty list):

$$\lambda xs.\texttt{case}_{List}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \Omega; (\texttt{Cons}\ y\ ys) \to y\}.$$

In our meta-notation we sometimes write $\texttt{case}_T\ e\ \texttt{of}\ alts$ if the alternatives of the `case`-expression do not matter. The reduction rule to evaluate a `case`-expression is the (case)-reduction, defined as:

$$(\texttt{case}_T\ (c_{T,i}\ e'_1\ \ldots\ e'_{ar(c_{T,i})})\ \texttt{of}\ \{\ldots; (c_{T,i}\ x_{i,1}\ \ldots\ x_{i,ar(c_{T,i})}) \to e_i; \ldots\}) \xrightarrow{(case)} e_i[e'_1/x_{i,1},\ldots,e'_{ar(c_{T,i})}/x_{i,ar(c_{T,i})}]$$

The rule says that if the first argument of a `case`-expression is a constructor application of the same type as the case-expression, then use the matching `case`-alternative for the same constructor, and replace the `case`-expression by the right hand side of the alternative, where the pattern variables are replaced by the arguments of the constructor application. Here $e[e_1/x_1,\ldots,e_n/x_n]$ means the *parallel* substitution of the variables x_i occurring in e by the expressions e_i.

▶ **Example 2.19.** Applying a (case)-reduction to the expression

$$\texttt{case}_{List}\ (\texttt{Cons True Nil})\ \texttt{of}\ \{\texttt{Nil} \to \Omega; (\texttt{Cons}\ y\ ys) \to y\}$$

results in $y[\texttt{True}/y, \texttt{Nil}/ys] = \texttt{True}$.

▶ **Exercise 2.20.** Write down an abstraction that computes the second element of a list. Hint: Use two nested `case`-expressions.

Note that a (case)-reduction requires that the first argument of `case` is evaluated to a constructor application. Hence, the reduction strategy must enforce the evaluation of the first argument of `case`. Thus, the reduction contexts for defining the normal order reduction will also include those contexts where the hole is in the first argument of `case`.

Note also that we consider an untyped calculus (except for the very weak typing of the data constructors belonging to their type constructors). As a consequence there are stuck expressions which are like a dynamic type error. I.e., expressions $(\texttt{case}_T\ \lambda x.e\ \texttt{of}\ alts)$, $(\texttt{case}_T\ (c_{T',i}\ e_1\ \ldots\ e_{ar(c_{T',i})})\ \texttt{of}\ alts)$ and $((c_{T,i}\ e_1\ \ldots\ e_{ar(c_{T,i})})\ e')$, and all expressions where one of these three expressions occurs at a reduction position (i.e. in a reduction context) are such expressions. These expressions will be treated like divergent expressions, since they have no normal order reduction to a WHNF.

▶ **Remark 2.21.** A formal argument to include data constructors and `case`-expressions as primitives to the core language is the following example (see [31]).

Let $e_1 := \lambda x,y.x\ (y\ (y\ I)))$ and $e_2 := \lambda x,y.x\ (y\ \lambda z.y\ (x\ I)\ z)$ (where I abbreviates the identity function). Then $e_1 \sim_c e_2$ in the lazy lambda calculus, but in the calculus extended by `case`-expressions and data constructors $e_1 \sim_c e_2$ does *not* hold: since the context

$$C := [\cdot]\ (\lambda u.u\ \texttt{True})\ (\lambda u.\texttt{case}_{Bool}\ u\ \texttt{of}\ \{\texttt{True} \to \texttt{False}; \texttt{False} \to I\})$$

distinguishes e_1 and e_2, since $C[e_1]\Downarrow$, but $C[e_2]\Uparrow$. Thus the extension by `case`-expressions and data constructors is not conservative (since it breaks existing equations) and thus these constructs should be added to reason about a core language of Haskell.

2.2.2 Strict Evaluation

The normal order reduction in the lazy lambda calculus is non-strict, which means that arguments are substituted without evaluating them to values before the substitution (as it is common in call-by-value calculi, see e.g. [20]). Thus, there is no possibility to enforce the evaluation of expressions except for using them in function position in an application, or for data constructors, as subexpression in a case-expression in the extended language. Sometimes strict evaluation is more efficient, e.g. in the expression $(\lambda x.(x\ x\ x))((\lambda y.y)\ (\lambda z.z))$ the argument $((\lambda y.y)\ (\lambda z.z))$ is evaluated three times, so it would be an optimization to first evaluate the argument and then apply a (β)-reduction. For these reasons, Haskell has the `seq`-operator to enforce the evaluation of expressions, i.e. in $\texttt{seq}\ e_1\ e_2$ first e_1 is evaluated to a value and then e_2 is returned as the result of the `seq`-expression. We cannot express such a function in the lazy lambda calculus, and thus we will add it as a primitive to our core language.

The operational semantics has to be extended by the following reduction rule

$$\texttt{seq}\ v\ e \xrightarrow{seq} e,\ \text{if } v \text{ is a value}$$

where a value is an abstraction or a constructor application.

Again there is a more formal argument why `seq` should be included (see [31] for details).

▶ **Remark 2.22.** Let $e_1 := \lambda x.x\ (\lambda y.x\ (Y\ K)\ \Omega\ y)\ (Y\ K)$ and $e_2 := \lambda x.x\ (x\ (Y\ K)\ \Omega)\ (Y\ K)$ where Ω, Y, and K are defined as in Example 2.12. Then $e_1 \sim_c e_2$ in the lazy lambda calculus, but the expressions e_1 and e_2 are not contextually equivalent in the lazy lambda calculus extended by `seq`: The context $C := ([\cdot]\ \lambda z.\texttt{seq}\ z\ I)$ distinguishes e_1, e_2 since $C[e_1]\Downarrow$ while $C[e_2]\Uparrow$.

There is also a counter-example which shows that in the calculus with `case`-expressions and constructors there are contextually equivalent expressions which are distinguishable if additionally the `seq`-operator is added to the language:

The expressions $e_3 := \lambda x.\texttt{case}_{Bool}\ (x\ \Omega)\ \texttt{of}\ \{\texttt{True} \to \texttt{True}; \texttt{False} \to \Omega\}$ and $e_4 := \lambda x.\texttt{case}_{Bool}\ (x\ \lambda y.\Omega)\ \texttt{of}\ \{\texttt{True} \to \texttt{True}; \texttt{False} \to \Omega\}$ are contextually equivalent in the lazy lambda calculus extended by `case`-expressions and data constructors, but if `seq` is added then they can be distinguished by the context $C' := [\cdot]\ \lambda u.\texttt{seq}\ u\ \texttt{True}$, since $C'[e_4]\Downarrow$ but $C'[e_3]\Uparrow$.

2.2.3 Named Functions and Recursion

It is possible to express recursive function in the lazy lambda calculus by using a fixpoint combinator like Y (defined in Example 2.12). For instance, the recursive *map*-function which

$$
\begin{array}{lcl@{\qquad}lcl}
FV(x) & := & \{x\} & FV(c_{T,i}\ e_1\ \ldots\ e_{ar(c_{T,i})}) & := & \bigcup_{i=1}^{ar(c_{T,i})} FV(e_i) \\
FV(\lambda x.e) & := & FV(e) \setminus \{x\} & FV(\texttt{seq}\ e_1\ e_2) & := & FV(e_1) \cup FV(e_2) \\
FV((e_1\ e_2)) & := & FV(e_1) \cup FV(e_2) & FV(f) & := & \emptyset
\end{array}
$$

$$FV(\texttt{case}_T\ e\ \texttt{of}\ \{pat_1 \to e_1; \ldots; pat_{|T|} \to e_{|T|}\}) := FV(e) \cup \bigcup_{i=1}^{|T|} (FV(e_i) \setminus FV(pat_i))$$

$$
\begin{array}{lcl@{\qquad}lcl}
BV(x) & := & \emptyset & BV(c_{T,i}\ e_1\ \ldots\ e_{ar(c_{T,i})}) & := & \bigcup_{i=1}^{ar(c_{T,i})} BV(e_i) \\
BV(\lambda x.e) & := & BV(e) \cup \{x\} & BV(\texttt{seq}\ e_1\ e_2) & := & BV(e_1) \cup BV(e_2) \\
BV((e_1\ e_2)) & := & BV(e_1) \cup BV(e_2) & BV(f) & := & \emptyset
\end{array}
$$

$$BV(\texttt{case}_T\ e\ \texttt{of}\ \{pat_1 \to e_1; \ldots; pat_{|T|} \to e_{|T|}\} := BV(e) \cup \bigcup_{i=1}^{|T|} BV(e_i) \cup FV(pat_i)$$

Figure 1 Free and bound variables in *LNAME*

maps a function to all elements of a list could be expressed as

$$(Y\ \lambda m, f, xs.\texttt{case}_{List}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \texttt{Nil}; (\texttt{Cons}\ y\ ys)\ \to \texttt{Cons}\ (f\ y)\ (m\ f\ ys)\})$$

Also nested recursion can be expressed by using multi-fixpoint combinators (see e.g. [10]). However, it is more convenient to express recursion directly. For this reason (and in this case not for semantic reasons), we add defined functions – so-called supercombinators – to our language. Thus, we assume that there is a set $\mathcal{F}$ of function symbols and that for every supercombinator $f \in \mathcal{F}$ there exists a *definition* of f of the form

$$f\ x_1\ \ldots\ x_n = e$$

where x_i are variables and e is an expression s.t. $FV(e) \subseteq \{x_1, \ldots, x_n\}$.

Thus supercombinators can be used to model named and recursive functions. For instance, the above *map*-function can be expressed as a supercombinator $\texttt{map} \in \mathcal{F}$ with the definition:

$$\texttt{map}\ f\ xs = \texttt{case}_{List}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \texttt{Nil}; (\texttt{Cons}\ y\ ys)\ \to \texttt{Cons}\ (f\ y)\ (\texttt{map}\ f\ ys\ )\}$$

As an example for a non-recursive supercombinator, consider the supercombinator $\texttt{head} \in \mathcal{F}$ which computes the first element of a list:

$$\texttt{head}\ xs = \texttt{case}_{List}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \Omega; (\texttt{Cons}\ y\ ys) \to y\}$$

The number n in a definition $f\ x_1\ \ldots\ x_n = e$ is called the arity of f and written as $ar(f)$.

In expressions, supercombinators occur as constants. The evaluation can only evaluate a supercombinator (by unfolding the function definition) if enough arguments are present, i.e. partial applications like $(\texttt{map}\ (\lambda x.x))$ are treated like abstractions, since they are semantically equivalent to $\lambda xs.\texttt{map}\ (\lambda x.x)\ xs$. For fully saturated applications of a supercombinator to arguments, the (SCβ)-reduction is used:

$$(f\ e_1\ \ldots\ e_n) \xrightarrow{SC\beta} e[e_1/x_1, \ldots, e_n/x_n],\ \text{if } f\ x_1\ \ldots\ x_n = e \text{ is the definition of } f$$

2.2.4 The Calculus *LNAME*

Since we have introduced all language constructs already, we can now briefly define the syntax and the semantics of the core language *LNAME*.

▶ **Definition 2.23** (Syntax of *LNAME*-expressions)**.** We assume a set of constants $f \in F$ called *function symbols*, *functions*, or *supercombinators*. The syntax of the language *LNAME* is defined by the following grammar, where $x, x_i \in Var$, $f \in F$, and $c_{T,i} \in K_T$:

$$
\begin{array}{lcl}
e, e_i \in E_{LNAME} & ::= & x \mid \lambda x.e \mid f \mid (e_1\ e_2) \mid (c_{T,i}\ e_1\ \ldots\ e_{ar(c_{T,i})}) \\
 & & \mid \texttt{case}_T\ e\ \texttt{of}\ \{pat_{T,1} \to e_1; \ldots; pat_{T,|T|} \to e_{|T|}\} \mid \texttt{seq}\ e_1\ e_2 \\
pat_{T,i} & ::= & (c_{T,i}\ x_{i,1}\ \ldots\ x_{i,ar(c_{T,i})})
\end{array}
$$

The sets of free and bound variables of an expression are defined in Figure 1. For every $f \in F$ there is a *definition* of the form $f\ x_1 \ldots\ x_n = e$ where x_i are variables and $FV(e) \subseteq \{x_1, \ldots, x_n\}$. We say that the arity of f is n in this case (written as $ar(f) = n$).

We do not repeat the definitions of α-renaming and α-equivalence for *LNAME*. They can be straightforwardly derived from the definitions for the lazy lambda calculus by respecting the new binders in case-patterns. We will also use the distinct variable convention for *LNAME*.

Besides abstractions also constructor applications will be treated as successfully evaluated programs:

▶ **Definition 2.24.** A *functional weak head normal form* (FWHNF) in *LNAME* is any abstraction and any expression of the form $(f\ e_1\ \ldots\ e_m)$ where $f \in F$ and $ar(f) > m$. A *constructor weak head normal form* (CWHNF) is any expression of the form $(c_{T,i}\ e_1\ \ldots\ e_{ar(c_{T,i})})$. A *weak head normal form* (WHNF) is any FWHNF or CWHNF.

A *context* $C \in C_{LNAME}$ of *LNAME* is any expression with a hole $[\cdot]$ at some expression position:

▶ **Definition 2.25.** The syntax of *contexts* $C \in C_{LNAME}$ is defined by the following grammar:

$$
\begin{array}{l}
C \in C_{LNAME} ::= [\cdot] \mid (\lambda x.C) \mid (C\ e) \mid (e\ C) \mid (c_{T,i}\ e_1 \ldots e_{j-1}\ C\ e_{j+1} \ldots e_{ar(c_{T,i})}) \\
\quad \mid (\texttt{seq}\ C\ e) \mid (\texttt{seq}\ e\ C) \mid (\texttt{case}_T\ C\ alts) \mid (\texttt{case}_T\ e\ \{\ldots; pat_{T,j} \to C; \ldots\})
\end{array}
$$

We will also use *multicontexts* $C[\cdot_1, \ldots, \cdot_n]$ which are expressions with several, i.e. zero or more, different holes.

There are four reduction rules in the calculus *LNAME*. Besides the β-reduction, there is the (case)-reduction to evaluate a case-expression, the (seq)-reduction to evaluate a seq-expression, and the (SC-β)-reduction to unfold a function definition:

$$
\begin{array}{ll}
\beta & (\lambda x.e_1)\ e_2 \to e_1[e_2/x] \\
\text{(case)} & (\texttt{case}_T\ (c_{T,i}\ e'_1\ \ldots\ e'_{ar(c_{T,i})})\ \texttt{of}\ \{\ldots; (c_{T,i}\ x_{i,1}\ \ldots\ x_{i,ar(c_{T,i})}) \to e_i; \ldots\}) \\
 & \to e_i[e'_1/x_{i,1}, \ldots, e'_{ar(c_{T,i})}/x_{i,ar(c_{T,i})}] \\
\text{(seq)} & \texttt{seq}\ v\ e \to e,\ \text{if } v \text{ is a WHNF.} \\
\text{(SC}\beta\text{)} & f\ e_1\ \ldots\ e_n \to e[e_1/x_1, \ldots, e_n/x_n],\ \text{ if } f\ x_1 \ldots\ x_n = e \text{ is the definition of } f.
\end{array}
$$

To define the normal order reduction, we proceed as in the lambda calculus and define reduction contexts:

▶ **Definition 2.26.** The *reduction contexts* R_{LNAME} of $LNAME$ are defined by the following grammar:

$$R \in R_{LNAME} ::= [\cdot] \mid (R\ e) \mid (\mathtt{case}_T\ R\ \mathtt{of}\ alts) \mid (\mathtt{seq}\ R\ e)$$

A *normal order reduction* $\xrightarrow{no}$ is any reduction in a reduction context, i.e. if $e_1 \xrightarrow{a} e_2$ where $a \in \{(\beta), (case), (seq), (SC\beta)\}$ and $R \in R_{LNAME}$, then $R[e_1] \xrightarrow{no} R[e_2]$.

▶ **Definition 2.27.** Let $e \in E_{LNAME}$. We say e *converges* (written $e{\Downarrow}$) iff there exists a normal order reduction of e to a WHNF, i.e. $\exists e' : e \xrightarrow{no,*} e'$ and e' is a WHNF. We sometimes also write $e{\Downarrow}e'$ in this case. If $e{\Downarrow}$ does not hold, then we write $e{\Uparrow}$ and say e *diverges*.

As in the lambda calculus we give an alternative description of normal order reduction by a labeling algorithm. Let $e \in E_{LNAME}$. Then the labeling starts with $e^\star$ and exhaustively applies the following shifting rules:

$$\begin{array}{lcl}
(e_1\ e_2)^\star & \to & (e_1^\star\ e_2) \\
(\mathtt{seq}\ e_1\ e_2)^\star & \to & (\mathtt{seq}\ e_1^\star\ e_2) \\
(\mathtt{case}_T\ e\ \mathtt{of}\ alts)^\star & \to & (\mathtt{case}_T\ e^\star\ alts)
\end{array}$$

Then a single normal order reduction step is defined by the following cases:

- If the labeling ends with $R[\mathtt{case}_T\ (c_{T,i}\ e_1 \dots e_{ar(c_{T,i})})^\star\ alts]$, then apply a (case)-reduction to the expression inside the hole of R.
- If the labeling ends with $R[(\lambda x.e_1)^\star\ e_2]$, then apply a (β)-reduction to the expression inside the hole of R.
- If the labeling ends with $R[f^\star\ e_1\ \dots\ e_n]$ and $ar(f) = n$, then apply a (SCβ)-reduction to the expression inside the hole of R.
- If the labeling ends with $R[\mathtt{seq}\ (c_{T,i}\ e_1 \dots e_{ar(c_{T,i})})^\star\ e']$, then apply a (seq)-reduction to the expression inside the hole of R.
- If the labeling ends with $R[\mathtt{seq}\ (\lambda x.e)^\star\ e']$, then apply a (seq)-reduction to the expression inside the hole of R.
- If the labeling ends with $R[\mathtt{seq}\ (f^\star\ e_1\ \dots\ e_m)\ e']$ where $ar(f) > m$, then apply a (seq)-reduction to the expression inside the hole of R.

▶ **Example 2.28.** Let $\mathsf{map}, \mathsf{not}, \mathsf{repTrue} \in \mathcal{F}$ where the definitions are

$$\begin{array}{ll}
\mathsf{map}\ f\ xs & = \mathtt{case}_{List}\ xs\ \mathtt{of}\ \{\mathtt{Nil} \to \mathtt{Nil}; (\mathtt{Cons}\ y\ ys) \to \mathtt{Cons}\ (f\ y)\ (\mathsf{map}\ f\ ys\ )\} \\
\mathsf{not}\ x & = \mathtt{case}_{Bool}\ x\ \mathtt{of}\ \{\mathtt{True} \to \mathtt{False}; \mathtt{False} \to \mathtt{True}\} \\
\mathsf{repTrue} & = \mathtt{Cons}\ \mathtt{True}\ \mathsf{repTrue}
\end{array}$$

We evaluate the expression $(\mathsf{map}\ \mathsf{not}\ \mathsf{repTrue})$ in normal order, where the expressions are labeled after exhaustively applying the shifting rules:

$$\begin{array}{ll}
 & (\mathsf{map}^\star\ \mathsf{not}\ \mathsf{repTrue}) \\
\xrightarrow{no,SC\beta} & \mathtt{case}_{List}\ \mathsf{repTrue}^\star\ \mathtt{of}\ \{ \\
 & \quad \mathtt{Nil} \to \mathtt{Nil}; \\
 & \quad (\mathtt{Cons}\ y\ ys) \to \mathtt{Cons}\ (\mathsf{not}\ y)\ (\mathsf{map}\ \mathsf{not}\ ys\ )\} \\
\xrightarrow{no,SC\beta} & \mathtt{case}_{List}\ (\mathtt{Cons}\ \mathtt{True}\ \mathsf{repTrue})^\star\ \mathtt{of}\ \{ \\
 & \quad \mathtt{Nil} \to \mathtt{Nil}; \\
 & \quad (\mathtt{Cons}\ y\ ys) \to \mathtt{Cons}\ (\mathsf{not}\ y)\ (\mathsf{map}\ \mathsf{not}\ ys\} \\
\xrightarrow{no,case} & (\mathtt{Cons}\ (\mathsf{not}\ \mathtt{True})\ (\mathsf{map}\ f\ \mathsf{repTrue}))^\star
\end{array}$$

The last expression is a WHNF, and thus $(\mathsf{map}\ \mathsf{not}\ \mathsf{repTrue}){\Downarrow}$.

▶ **Exercise 2.29.** Let $\mathsf{last} \in \mathcal{F}$ be defined as

$$\begin{array}{l}
\mathsf{last}\ xs = \mathtt{case}_{List}\ xs\ \mathtt{of}\ \{ \\
\qquad \mathtt{Nil} \to \Omega; \\
\qquad (\mathtt{Cons}\ y\ ys) \to \mathtt{case}_{List}\ ys\ \mathtt{of}\ \{\mathtt{Nil} \to y; (\mathtt{Cons}\ u\ us) \to \mathsf{last}\ ys\}\}
\end{array}$$

Evaluate the expression $((\lambda xs.\mathsf{last}\ xs)\ (\mathtt{Cons}\ ((\lambda x.x)\ \mathtt{True})\ \mathtt{Nil}))$ in normal order until a WHNF is reached.

2.3 Contextual Equivalence and Program Transformations

2.3.1 Contextual Equivalence

Contextual preorder and contextual equivalence can now be defined in the standard way:

▶ **Definition 2.30.** Let $e_1, e_2 \in E_{LNAME}$. Then e_2 *contextually approximates* e_1, written as $e_1 \leq_c e_2$ (or alternatively $e_2 \geq_c e_1$), iff for all contexts $C \in C_{LNAME}$: $C[e_1]{\Downarrow} \implies C[e_2]{\Downarrow}$. The relation $\leq_c$ is called the *contextual approximation* or alternatively the *contextual preorder*. The expressions e_1 and e_2 are *contextually equivalent*, written as $e_1 \sim_c e_2$, iff $e_1 \leq_c e_2$ and $e_2 \leq_c e_1$. The relation $\sim_c$ is called *contextual equivalence*.

▶ **Proposition 2.31.** *Contextual preorder $\leq_c$ is a precongruence, and $\sim_c$ is a congruence.*

Proof. It is sufficient to show that $\leq_c$ is precongruence, i.e. that it is a preorder which is compatible w.r.t. substitution into contexts. Clearly, $\leq_c$ is reflexive ($e \leq_c e$ for all $e \in E_{LNAME}$). Transitivity of $\leq_c$ also holds: Suppose that $e_1 \leq_c e_2$ and $e_2 \leq_c e_3$ and that $C[e_1]{\Downarrow}$ holds for some context C. Then the definition of $\leq_c$ shows $C[e_2]{\Downarrow}$ and again the definition of $\leq_c$ shows $C[e_3]{\Downarrow}$. Finally let $e_1 \leq_c e_2$ and let C be a context. We have to show $C'[C[e_1]]{\Downarrow} \implies C'[C[e_2]]{\Downarrow}$ for all C'. This holds, since $C'[C[\cdot]]$ is again a context. ◀

▶ **Exercise 2.32.** Assume that $\mathsf{head}, \mathsf{tail} \in \mathcal{F}$, where the definitions of the supercombinators head and tail are as follows:

$$\begin{array}{ll}
\mathsf{head}\ xs & = \mathtt{case}_{List}\ xs\ \mathtt{of}\ \{\mathtt{Nil} \to \Omega; (\mathtt{Cons}\ y\ ys) \to y\} \\
\mathsf{tail}\ xs & = \mathtt{case}_{List}\ xs\ \mathtt{of}\ \{\mathtt{Nil} \to \Omega; (\mathtt{Cons}\ y\ ys) \to ys\}
\end{array}$$

Prove the following in-equations $e_1 \not\sim_c e_2$ by providing a counter-example in form of a context C, s.t. either $C[e_1]{\Downarrow} \wedge C[e_2]{\Uparrow}$ or $C[e_1]{\Uparrow} \wedge C[e_2]{\Downarrow}$.

- $\mathtt{True} \not\sim_c \mathtt{False}$
- $x \not\sim_c y$
- $\lambda x.\lambda y.y \not\sim_c \lambda x.\lambda y.\mathtt{seq}\ x\ y$
- $\mathsf{head} \not\sim_c \mathsf{tail}$
- $\lambda xs.\mathtt{Cons}\ (\mathsf{head}\ xs)\ (\mathsf{tail}\ \mathsf{xs}) \not\sim_c \lambda xs.xs$

2.3.2 Program Transformations and Proof Techniques

A program transformation P is a binary relation on expressions. For example, the reduction rules (β), $(case)$, (seq) and $(SC\beta)$ are program transformations. Another well-known program transformation in the lambda calculus is the η-expansion defined as

$$e \xrightarrow{\eta} \lambda x.(e\ x) \text{ if } x \notin FV(e)$$

▶ **Definition 2.33.** A program transformation P is called *correct* iff it preserves contextual equivalence, i.e. $P \subseteq \sim_c$.

Note that η-expansion is not a correct program transformation in *LNAME*, since e.g. $\texttt{True} \xrightarrow{\eta} \lambda x.(\texttt{True}\ x)$ but the context $C := \texttt{case}_{Bool}\ [\cdot]\ \texttt{of}\ \{\texttt{True} \to \texttt{True}; \texttt{False} \to \texttt{False}\}$ shows that $\texttt{True} \not\sim_c \lambda x.(\texttt{True}\ x)$, since $C[\texttt{True}]\Downarrow$, but $C[\lambda x.(\texttt{True}\ x)]\Uparrow$.

The context lemma (in the presented form sometimes also called CIU-Lemma[4]) is a helpful tool to prove contextual equivalences and correctness of program transformations. It shows that it is sufficient to consider closing substitutions and reductions contexts instead of all contexts. We omit its proof, it can be found e.g. in [36, 34].

▶ **Theorem 2.34** (Context lemma). *Let $e_1, e_2 \in E_{LNAME}$ be expressions. If for all (closed) reductions contexts R and substitutions σ, s.t. $\sigma(e_1)$ and $\sigma(e_2)$ are closed the implication $R[\sigma(e_1)]\Downarrow \implies R[\sigma(e_2)]\Downarrow$ holds, then also $e_1 \sim_c e_2$ holds.*

▶ **Theorem 2.35.** *The reductions (β), (case), (seq), and $(SC\beta)$ are correct program transformations.*

Proof. Let $e_1 \xrightarrow{a} e_2$ where $a \in \{(\beta), (case), (seq), (SC\beta)\}$. Let σ be a substitution such that $\sigma(e_1), \sigma(e_2)$ are closed. Then $\sigma(e_1) \xrightarrow{a} \sigma(e_2)$ by the same reduction rule. Moreover, for every reduction context R: $R[\sigma(e_1)] \xrightarrow{no,a} R[\sigma(e_2)]$. Now assume $R[\sigma(e_1)]\Downarrow$. Since normal order reduction is unique, we also have $R[\sigma(e_2)]\Downarrow$. Thus the context lemma shows $e_1 \leq_c e_2$. For the other direction, assume $R[\sigma(e_2)]\Downarrow$. Then clearly $R[\sigma(e_1)]\Downarrow$ and the context lemma shows $e_2 \leq_c e_1$. ◀

▶ **Lemma 2.36.** *The restricted variant of η-expansion:*

$$\lambda x.e \to \lambda y.((\lambda x.e)\ y) \text{ if } y \notin FV(e)$$

is a correct program transformation.

Proof. This holds, since $\lambda y.((\lambda x.e)\ y) \xrightarrow{C,\beta} \lambda y.e[y/x] =_\alpha \lambda x.e$ and since (β) is a correct program transformation. ◀

▶ **Proposition 2.37.** *Let e be a closed, diverging expression of LNAME. Then for every expression e' the inequation $e \leq_c e'$ holds.*

Proof. Again we use the context lemma: Let σ be a substitution s.t. $\sigma(e')$ (and thus also $\sigma(e)$) is closed. Then $R[\sigma(e)]\Uparrow$, since $\sigma(e) = e$ and $e\Uparrow$. Thus for all reduction contexts $R[\sigma(e)]\Downarrow \implies R[\sigma(e')]\Downarrow$ and the context lemma shows $e \leq_c e'$. ◀

▶ **Corollary 2.38.** *Let e_1, e_2 be closed diverging expressions. Then $e_1 \sim_c e_2$.*

Since all closed diverging expressions are in the same equivalence class of contextual equivalence, it makes sense to represent them by an abstract symbol:

▶ **Definition 2.39.** We use the symbol $\bot$ to represent a closed diverging expression of *LNAME*. E.g., Ω is such an expression.

Some helpful program transformations are the following $\bot$-reductions. They allow us to simplify expressions that contain a $\bot$ at a reduction position:

[4] CIU is an abbreviation for closed instances of use.

▶ **Definition 2.40.** The following transformations are called the $\bot$-*reductions*:

$$\begin{array}{ll} (app\bot) & (\bot\ e) \to \bot \\ (case\bot) & \texttt{case}_T\ \bot\ \texttt{of}\ alts \to \bot \\ (seq\bot) & \texttt{seq}\ \bot\ e \to \bot \end{array}$$

The proof of the following proposition is left as an exercise:

▶ **Proposition 2.41.** *The $\bot$-reductions are correct program transformations.*

▶ **Proposition 2.42.** *There is no largest element w.r.t. $\leq_c$.*

Proof. Assume there is a largest element e. Then clearly $e\Downarrow$. Let e' be the corresponding WHNF.

If e' is a CWHNF, then $\lambda x, y.y \not\leq_c e$, which is shown using the context $C = [\cdot]\ (\lambda z.z)$: $C[e]\Uparrow$, but $C[\lambda x, y.y]\Downarrow$.

If e' is an FWHNF, then let c be a constructor of type T with arity n. Then $(c \bot\ \ldots\ \bot) \not\leq_c e$, which is shown using the context $C = \texttt{case}_T\ [\cdot]\ \texttt{of}\ \{\ldots; (c\ x_1\ \ldots\ x_n) \to \lambda z.z; \ldots\}$: $C[e]\Uparrow$, but $C[c \bot\ \ldots\ \bot]\Downarrow$. ◀

Note that the lazy lambda calculus has $(Y\ K)$ as largest element w.r.t. $\leq_c$.

2.3.3 A Diagram-Based Proof Technique

We demonstrate a proof technique to show correctness of program transformations where other (direct methods) often fail. We first explain the so-called diagram-based technique in general and then demonstrate the technique for some exemplary program transformations.

As already introduced, a program transformation $\xrightarrow{P}$ is a binary relation on expressions. To prove correctness of the transformation $\xrightarrow{P}$, we have to show that for all e_1, e_2 with $e_1 \xrightarrow{P} e_2$ and all contexts C the equivalence $C[e_1]\Downarrow \iff C[e_2]\Downarrow$ holds. Using the context lemma it suffices to show for all e_1, e_2 with $e_1 \xrightarrow{P} e_2$: $R[\sigma(e_1)]\Downarrow \iff R[\sigma(e_2)]\Downarrow$ for all reduction contexts R and closing substitutions σ. To simplify the notation, we consider the closure $\xrightarrow{R,\sigma,P}$ of $\xrightarrow{P}$ by reduction contexts and closing substitutions:

$$\xrightarrow{R,\sigma,P} := \left\{ (R[\sigma(e_1)], R[\sigma(e_2)]) \;\middle|\; \begin{array}{l} e_1 \xrightarrow{P} e_2, R \in R_{LNAME}, \sigma \text{ a substitution}, \\ FV(\sigma(e_1)) \cup FV(\sigma(e_2)) = \emptyset \end{array} \right\}$$

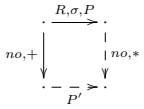
Thus to show correctness of $\xrightarrow{P}$, it suffices to show that the equivalence $e_1\Downarrow \iff e_2\Downarrow$ holds for all $e_1 \xrightarrow{R,\sigma,P} e_2$.

2.3.3.1 Proving $\xrightarrow{P} \subseteq \leq_c$

Let us consider the implication $e_1\Downarrow \implies e_2\Downarrow$ for all $e_1 \xrightarrow{R,\sigma,P} e_2$, and thus the proof of $\xrightarrow{P} \subseteq \leq_c$. Based on the normal order reduction $e_1 \xrightarrow{no,k} e_1'$ where e_1' is a WHNF and $k \geq 0$ our proof method (inductively) constructs a sequence of normal order reductions $e_2 \xrightarrow{no,k'} e_2'$ where e_2' is a WHNF, which shows $e_2\Downarrow$.

For the construction of the reduction sequence of e_2, we have to compute all overlappings of a normal order reduction step and a transformation step, i.e. all *forks* $e' \xleftarrow{no} e \xrightarrow{R,\sigma,P} e''$ for all expressions e, e', e''. This computation is done by inspecting the definition of the normal order reduction, the program transformation and by unifying the corresponding

meta-expressions in the left hand sides of the reduction and transformation rules. The computation is related to the computation of critical pairs for term rewrite systems (see e.g. [3]). The differences are that not all term positions are inspected (only the top position), that contexts are given explicitly (in form of a meta-symbol for the reduction context) and that higher-order meta-terms (i.e. terms with binders) are overlapped. After computing the overlappings, the forks are closed by showing that program transformations and normal order reductions exist to join the "critical pair" (e, e''). Such a situation is expressed by a *forking diagram* for transformation $\xrightarrow{R,\sigma,P}$ which is of the form:



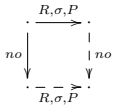
In such a diagram the solid arrows represent the given reductions and transformations, and the dashed arrows represent the existentially-quantified reductions to join the forking situation. The diagram abstracts away from the concrete expressions (and thus represents them by the dot-symbol $\cdot$). Note that for the given fork we allow more than one normal order reduction (expressed by the transitive closure $\xrightarrow{no,+}$). Since the calculus is deterministic, this is sometimes helpful (since we can search for the "right" successor w.r.t. normal order reduction). The arrow $\xrightarrow{P'}$ represents arbitrary transformation steps, not necessarily of the transformation $\xrightarrow{R,\sigma,P}$. However, if other transformations than $\xrightarrow{R,\sigma,P}$ occur in such a diagram, then also diagrams for these transformations are required, or correctness of these transformations must be known.

Often more information about the normal order reduction is attached to the arrows in a diagram, i.e. the name of the used reduction rule and the number of normal order reductions. The semantics of a forking diagram is that it describes how an overlapping may be closed, but it does not necessarily cover every such case. I.e., there may be several forking diagrams describing the same fork, but other possibilities to close the fork. We say that a forking diagram is *applicable* to a (concrete) fork $e' \xleftarrow{no} e \xrightarrow{R,\sigma,P} e''$, if the existentially quantified reductions and expressions of the diagram can concretely be constructed for e' and e''. To cover all cases the notion of completeness is required:

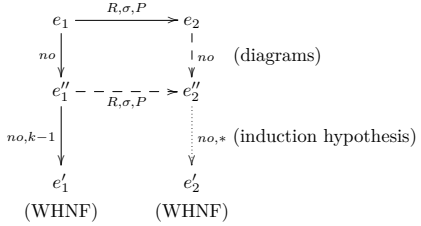
▶ **Definition 2.43.** A set of forking diagrams for transformation $\xrightarrow{R,\sigma,P}$ is *complete* iff the set contains at least one applicable diagram for every fork $e' \xleftarrow{no} e \xrightarrow{R,\sigma,P} e''$ where $e'{\Downarrow}$.

Now the proof of $e_1{\Downarrow} \implies e_2{\Downarrow}$ for all $e_1 \xrightarrow{R,\sigma,P} e_2$ is done by induction on the given reduction sequence $e_1 \xrightarrow{no,k} e_1'$ where e_1' is a WHNF and $k \geq 0$. The base case of such an inductive proof is that e_1 is a WHNF. For this case, usually a lemma is required which states that either e_2 is also a WHNF in this case, or that e_2 can be reduced (in normal order) to a WHNF. Such a lemma can often be proved by inspecting the WHNFs and the program transformation. For the induction step we assume $k > 0$ and thus $e_1 \xrightarrow{no} e_1'' \xrightarrow{no,k-1} e_1'$. Then first apply a forking diagram to $e_1'' \xleftarrow{no} e_1 \xrightarrow{R,\sigma,P} e_2$ and then use the induction hypothesis for $e_1'' \xrightarrow{P'} e$. In general, the induction measure depends on the form of the diagrams, in easy cases the length k is sufficient. We illustrate the induction proof for the simple case

that the only forking diagram is a simple commutation, i.e. a square diagram:

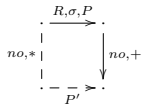


In this case we can use induction on k and the diagram shows that $e_1'' \xrightarrow{R,\sigma,P} e_2''$ for some e_2'' with $e_2 \xrightarrow{no} e_2''$. Thus we can apply the induction hypothesis to $e_1'' \xrightarrow{R,\sigma,P} e_2''$ and thus $e_2''{\Downarrow}$ which immediately implies $e_2{\Downarrow}$. This can be illustrated by the following picture:



2.3.3.2 Proving $\xrightarrow{P} \subseteq \geq_c$

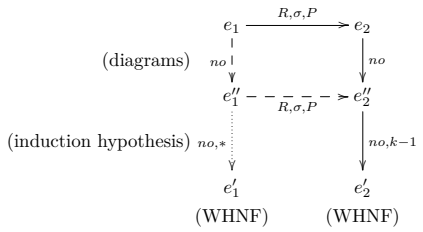
For proving the missing direction, i.e. showing that the implication $e_2{\Downarrow} \implies e_1{\Downarrow}$ holds for all $e_1 \xrightarrow{R,\sigma,P} e_2$, the proof technique is completely analogous with the difference that the given reduction and transformation steps do not form a fork but a sequence. I.e., one has to compute all overlappings of the form $e_1 \xrightarrow{R,\sigma,P} e_2 \xrightarrow{no} e_2'$ and join them accordingly. The overlappings and their joinability is expressed by so-called *commuting diagrams* for $\xrightarrow{R,\sigma,P}$, which are of the form



where again solid arrows mean the given expressions, reductions and transformations, and dashed arrows are existentially quantified.

▶ **Definition 2.44.** A set of commuting diagrams for transformation $\xrightarrow{R,\sigma,P}$ is *complete* iff the set includes at least one applicable diagram for every sequence $e \xrightarrow{R,\sigma,P} e' \xrightarrow{no} e''$ where $e''{\Downarrow}$.

The commuting diagrams together with a lemma for the base case (stating that $e_1{\Downarrow}$ whenever $e_1 \xrightarrow{R,\sigma,P} e_2$ and e_2 is a WHNF) is then used to construct a converging reduction sequence for e_1 from a converging reduction sequence for e_2. I.e., if only a square diagram exists, then the induction step in this proof can be illustrated as follows:



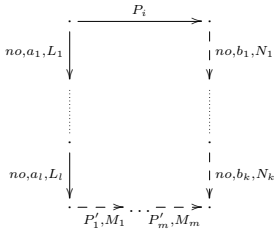
2.3.3.3 Notes on the Automation of the Diagram-Based Method

Finding the overlappings and closing the diagrams can be automated by using an extended unification algorithm with contexts and higher-order terms (e.g. using nominal unification) to find the overlappings and using matching and search algorithms to close the diagrams. This is an active research topic. In a research project[5] this is analyzed in a generic way including a lambda-calculus with `letrec`-expressions, where the unification algorithm becomes even more complicated [23, 22].

The proof of correctness of a program transformation is not finished by having computed the diagrams, since, as demonstrated before, one has to show that the induction using the diagrams is possible. Interestingly, this problem can be formulated as a termination problem for term rewrite systems (with free variables), which we will briefly explain (see [21, 26] for details). We only consider the forking diagrams here, since the commuting diagrams can be treated analogously.

A *closed* set of forking diagrams for program transformations $P_1, \ldots, P_n$ (where P_i may be the transformation $\xrightarrow{R,\sigma,P}$ whose correctness is considered) is the union of complete sets of forking diagrams for transformation P_i for $i = 1, \ldots, n$ such that all program transformations occurring in the diagrams are the transformations $P_1, \ldots, P_n$.

In such a case every diagram for the P_i can be written as



where $P'_j \in \{P_1, \ldots, P_n\}$, $a_i, b_j \in \mathcal{P}(\{\beta, SC\beta, seq, case\})$ (if a reduction is labeled by more than one reduction rule, then all of the rules may be possible in the concrete reduction) and $L_i, M_{i'}, N_{i''}$ are either empty or $+$ where $+$ is the transitive closure of the corresponding reduction or transformation.

[5] `http://www.ki.cs.uni-frankfurt.de/research/dfg-diagram/en/`

Let us ignore the transitive closure first, and thus assume that no $+$ occurs in the diagrams. The first observation is that such a diagram can be written as a rewrite rule from the given sequences to the existentially quantified:

$$\xleftarrow{P_i} \xrightarrow{no,a_1} \ldots \xrightarrow{no,a_l} \rightsquigarrow \xrightarrow{no,b_1} \ldots \xrightarrow{no,b_k} \xleftarrow{P'_m} \ldots \xleftarrow{P'_1}$$

Indeed, the induction proof mentioned before does such a rewriting by applying the rules to concrete reduction sequences until no more rule is applicable. However, on the one hand we have to add rewrite rules for the induction base: An answer diagram is a rewrite rule of the form

$$\xleftarrow{P_i} A \rightsquigarrow \xrightarrow{no,b_1,N_1} \ldots \xrightarrow{no,b_k,N_k} A$$

where A represents a WHNF and the diagram shows that whenever a transformation P_i is applied to a WHNF, then also the transformed expression converges. A set of answer diagrams is complete for transformation P_i, if for every concrete WHNF e and transformation $e \xrightarrow{P_i} e'$ an applicable diagram is in the set. So a concrete converging sequence $e \xrightarrow{no,a_1} e_1 \ldots e_{n-1} \xrightarrow{no,a_n} e_n$ together with the transformation step $e \xrightarrow{P_i} e'$ is abstractly represented as $\xleftarrow{P_i} \xrightarrow{no,a_1} \ldots \xrightarrow{no,a_n} A$ and the rewrite rules rewrite it to a normal form $\xrightarrow{no,a'_1} \ldots \xrightarrow{no,a'_m} A$ which has a concretization $e' \xrightarrow{no,a'_1} e'_1 \ldots e'_{m-1} \xrightarrow{no,a'_m} e'_m$ where e'_m is a WHNF.

All these rules can easily be expressed with a term rewriting system which has unary symbols for the reductions $\xrightarrow{no,a_o}$ and the transformations $\xrightarrow{P_i}$ and a constant A for representing the answers:

- For a forking diagram:

$$P_i(no\_a_1(\ldots(no\_a_l(X)))) \to no\_b_1(\ldots(no\_b_k(P'_m(\ldots(P'_1(X))))))$$

- For an answer diagram

$$P_i(A) \to no\_b_1(\ldots(no\_b_k(A)))$$

Proving termination of the term rewrite system (e.g. using an automated termination prover) then ensures that the induction proof in the correctness proof is possible.

Diagrams that contain transitive closures (denoted by $+$ in the reduction arrows) represent infinitely many diagrams and thus their encoding as a term rewrite system is not so obvious. For the transitive closures occurring in the left hand side of rules, the encoding is rather easy, since the rules simply have to ensure that arbitrary many reductions can be removed. We illustrate the encoding for a single transitive closure (if there are more reductions, then the encoding can be extended in a straightforward manner). Let $\xleftarrow{P_i} \xrightarrow{no,a_i,+} \rightsquigarrow r$ be a forking diagram. Then the following term rewrite rules are used to encode the diagram:

$$\begin{array}{lcl} P_i(no\_a_i(X)) & \to & P_i(remove\_no\_a_i(X)) \\ P_i(remove\_no\_a_i(no\_a_i(X))) & \to & P_i(remove\_no\_a_i(X)) \\ P_i(remove\_no\_a_i(X)) & \to & enc(r) \end{array}$$

where $remove\_no\_a_i$ is a fresh unary function symbol, $enc(r)$ is the TRS-encoding of the right hand side r.

For transitive closures in the right hand sides of the rules, an naive approach would be to add rules of the form $P_i^+(X) \to P_i(P_i^+(X))$ but these rules are useless since they lead to a nonterminating rewrite system (since they also represent infinite sequences of $\xrightarrow{P_i}$-steps which are not in the semantics of the transitive closure). Inserting all unfolded diagrams is

also not an option for automation. A solution is to add term rewrite rules which allow free variables on the right hand side which may only be instantiated with constructor terms:

$$P_i^+(X) \to genP_i(K, X)$$

here K is such a free variable on the right hand side of the rule. Finally, rules for generating $K+1$ many P_i-steps have to be added:

$$\begin{array}{lcl} genP_i(succ(K), X) & \to & P_i(genP_i(K, X)) \\ genP_i(zero, X) & \to & P_i(X) \end{array}$$

where *succ* is a unary constructor symbol and *zero* is a constant. An adapted version of AProVE supports innermost-termination of such systems and the certifier CeTA can certify these proofs. Another option are *Integer Term Rewrite Systems* [8] which are also supported by AProVE and allow to explicitly deal with numbers in the term rewrite systems.

2.3.4 Correctness of a Program Transformation (An Example)

We consider the transformation (case-app):

$$\begin{array}{ll} \text{(case-app)} & ((\texttt{case}_T\ e\ \texttt{of}\ \{pat_1 \to e_1; \ldots; pat_n \to e_n\})\ e') \\ & \to (\texttt{case}_T\ e\ \texttt{of}\ \{pat_1 \to (e_1\ e'); \ldots; pat_n \to (e_n\ e')\}) \end{array}$$

▶ **Lemma 2.45.** *A complete set of forking diagrams for (R,case-app) is:*

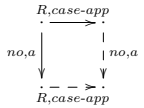
for $a \in \{(case), (seq), (SC\beta), (\beta)\}$

Proof. We compute a complete set of forking diagrams for $(R, case\text{-}app)$, i.e. we have to inspect all the overlappings of a (case-app)-transformation applied in a reduction context with a normal order reduction. Thus it is sufficient to unify the left hand side of the normal order rules (i.e. the reduction rules applied in reduction contexts) against the left hand side of (case-app) applied in a reduction context. Note that this unification is done on the level of meta-terms.

Let $R_1[((\texttt{case}_T\ e\ \texttt{of}\ \{pat_1 \to e_1; \ldots; pat_n \to e_n\})\ e')]$ be the left hand side of an $(R, case\text{-}app)$-transformation.

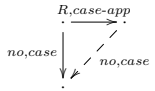
The general case is that $R_2[s] \xrightarrow{no} R_2[t]$ with redex $s \to t$. We split this case by the concrete reduction rule (instantiating s and t with more concrete expressions), if needed.

An easy case is that $R_2 = R_1[(\texttt{case}_T\ R_3\ \texttt{of}\ alts)\ e')]$, since in this case the reduction and the transformation commute, i.e. the forking diagram



for $a \in \{(case), (seq), (SC\beta), (\beta)\}$

is applicable. For $a \neq (case)$ this is the only possible case. For an (no,case)-reduction there is the additional case that $R_2 = R_1[[\cdot]\ e']$. Then the following triangle diagram holds:

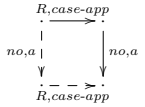


◀

▶ **Lemma 2.46.** *A complete set of commuting diagrams for (R,case-app) is:*

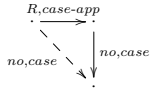
for $a \in \{(case), (seq), (SC\beta), (\beta)\}$

Proof. We compute a complete set of commuting diagrams and thus have to unify the right hand side of a (case-app)-transformation (applied in a reduction context) with normal order-reductions. Let $R_1[(\texttt{case}_T\ e\ \texttt{of}\ \{pat_1 \to (e_1\ e'); \ldots; pat_n \to (e_n\ e')\})]$ be the right hand side. The cases where the redex of the normal order reduction is inside e are similar as for the forking diagrams, since the normal order reduction and the transformation commute:



for $a \in \{(case), (seq), (SC\beta), (\beta)\}$

Again the only other case is that a (no,case)-reduction is applied where the redex of the normal order reduction is the redex of the (case-app)-transformation. Again the following triangle diagram holds:



◀

▶ **Proposition 2.47.** *The transformation (case-app) is correct.*

Proof. We use the context lemma and thus it suffices to show for all reduction contexts R, all closing substitutions σ, all e_1, e_2 with $e_1 \xrightarrow{case\text{-}app} e_2$:

1. $R[\sigma(e_1)]\Downarrow \implies R[\sigma(e_2)]\Downarrow$
2. $R[\sigma(e_2)]\Downarrow \implies R[\sigma(e_1)]\Downarrow$

Since $e_1 \xrightarrow{case\text{-}app} e_2$ implies $\sigma(e_1) \xrightarrow{case\text{-}app} \sigma(e_2)$ we can ignore the substitutions in the following.

1. Suppose that $R[e_1]\Downarrow$, i.e. $R[e_1] \xrightarrow{no,k} e_1'$ for some WHNF e_1' and $k \geq 0$. We use induction on k to show $R[e_2]\Downarrow$. For the base case ($k = 0$) it suffices to verify that $R[e_2]$ is a WHNF if $R[e_1]$ is a WHNF which holds, since neither $R[e_1]$ nor $R[e_2]$ can be a WHNF. For the induction step let $R[e_1] \xrightarrow{no} e_1'' \xrightarrow{no,k-1} e_1'$. We apply a forking diagram to the fork $e_1'' \xleftarrow{no} R[e_1] \xrightarrow{case\text{-}app} R[e_2]$. There are two cases:
   - If the first diagram is applicable, then $e_1'' \xrightarrow{R,case\text{-}app} e_2'' \xleftarrow{no} R[e_2]$ for some expression e_2''. In this case we apply the induction hypothesis to e_1'' and thus $e_2''\Downarrow$ which immediately implies $R[e_2]\Downarrow$.
   - If the second diagram is applicable, then $R[e_2] \xrightarrow{no} e_1''$ and thus $R[e_2]\Downarrow$
2. The proof is completely analogous to the first case, where instead of the forking diagrams, the commuting diagrams are applied.

◀

▶ **Remark 2.48.** The inductions in the proof can be automated by proving termination of the term rewrite system:

```
(VAR x)
(RULES
  T(no_beta(x))    -> no_beta(T(x))
  T(no_scbeta(x)) -> no_scbeta(T(x))
  T(no_case(x))    -> no_case(T(x))
  T(no_seq(x))     -> no_seq(T(x))
  T(no_case(x))    -> no_case(x)
  T(A)             -> A
)
```

where T represents $\xleftarrow{R,case\text{-}app}$ for the forking case and $\xrightarrow{R,case\text{-}app}$ for the commuting case. The termination prover AProVE [2, 9] shows termination of the TRS within a few seconds.

▶ **Exercise 2.49.** The transformation (seq-app) is defined as follows:

$$\text{(seq-app)} \quad ((\texttt{seq}\ e_1\ e_2)\ e_3) \to (\texttt{seq}\ e_1\ (e_2\ e_3))$$

Prove correctness of the transformation (seq-app) by first computing complete sets of forking and commuting diagrams, answer diagrams and then transforming all diagrams into a term rewrite system whose termination ensures correctness of the program transformation. Use a termination prover to show termination of the system.

2.3.5 Induction and List Laws

In this section we consider program transformation on expressions which process lists. Since direct proof techniques usually fail to show correctness of such program transformations and since lists are recursive data structures which may be potentially infinitely long, some formalisms are required.

▶ **Definition 2.50.** Let $e_1 \leq_c e_2 \leq_c \ldots$ be an (infinite) ascending $\leq_c$-chain. We write $\langle e_i \rangle$ for such a chain. An expression e is an upper bound of this chain iff for all i: $e_i \leq_c e$ holds. Expression e is a least upper bound (lub) of this chain (written $e = \text{lub}(\langle e_i \rangle)$), iff e is an upper bound and for any other upper bound e' of this chain $e \leq_c e'$ holds.

Expression e is a *contextually least upper bound* (club) of the chain (written $e = \text{club}(\langle e_i \rangle)$) iff $\forall C : C[e] = \text{lub}(\langle C[e_i] \rangle)$.

We prove some helpful properties of ascending chains, lubs, and clubs:

▶ **Lemma 2.51.** *1. Let r_1, r_2 be (c)lubs of the chain $\langle e_i \rangle$. Then $r_1 \sim_c r_2$.*
2. For an ascending chain $\langle e_i \rangle$ and a multicontext C, also $C[e_1, \cdots, e_1] \leq_c C[e_2, \cdots, e_2] \leq_c \ldots$ is an ascending chain.
3. Let $r := \text{club}(\langle e_i \rangle)$ and C be a multicontext. Then $C[r, \cdots, r]$ is a club of $\langle C[e_i, \cdots, e_i] \rangle$.

Proof. For item (1), the definition of lubs implies that $r_1 \leq_c r_2$ and $r_2 \leq_c r_1$. For the remaining parts, we first show the following property:

$$\begin{array}{l}\text{For all } e, e' \in E_{LNAME} \text{ and multicontexts } C \text{ it holds:} \\ \quad e \leq_c e' \implies C[e, \cdots, e] \leq_c C[e', \ldots, e']\end{array} \qquad (\dagger)$$

We use induction on the number of holes in C. If C has no holes, then the claim holds. If C has $n > 0$ holes, then consider the context $C' := C[e, \cdot_2, \cdots, \cdot_n]$. Since C' has $n - 1$ holes, the induction hypothesis shows $C'[e, \cdots, e] \leq_c C'[e', \cdots, e']$. Finally, $e \leq_c e'$ and the congruence property of $\leq_c$ show $C[e, e', \cdots, e'] \leq_c C[e', \cdots, e']$ and thus the claim holds.

Now the implication (†) immediately shows part (2) of the lemma.

For part (3), implication (†) shows that $C[r, \cdots, r]$ is a lub of $\langle C[e_i, \ldots, e_i] \rangle$. Let C' be a context, then we have to show: $C'[C[r, \cdots, r]]$ is a lub of $\langle C'[C[e_i, \cdots, e_i]] \rangle$. Let s be an upper bound of $\langle C'[C[e_i, \cdots, e_i]] \rangle$. We have to show $C'[C[r, \cdots, r]] \leq_c s$. Assume this is false and thus there exists a context C'' s.t. $C''[C'[C[r, \cdots, r]]]\Downarrow$ and $C''[s]\Uparrow$. Since s is an upper bound, we also have $C''[C'[C[e_i, \cdots, e_i]]]\Uparrow$ for all e_i. Since $r = \text{club}(\langle e_i \rangle)$, and thus $e_i \leq_c r$ for all e_i, implication (†) shows $C''[C'[C[r, \cdots, r]]]\Uparrow$ which is a contradiction, and thus $C'[C[r, \cdots, r]] \leq_c s$ holds. ◀

▶ **Lemma 2.52.** *Let $\langle e_i \rangle$ be an ascending chain and let e be an expression. If*

1. for all i: $e_i \leq_c e$, and
2. for all contexts C: $C[e]\Downarrow \implies \exists i : C[e_i]\Downarrow$

Then $e = \text{club}(\langle e_i \rangle)$.

Proof. Let C' be a context. We have to show that $C'[e] = \text{lub}(\langle C'[e_i] \rangle)$. Since $\leq_c$ is a precongruence we have $C'[e_i] \leq_c C'[e]$, i.e. $C'[e]$ is an upper bound. Now let e' be another upper bound of the chain $\langle C'[e_i] \rangle$. We have to show $C'[e] \leq_c e'$. Let C'' be a context s.t. $C''[C'[e]]\Downarrow$. Then by condition (2) there exists an index j s.t. $C''[C'[e_j]]\Downarrow$. Since e' is an upper bound of $\langle C'[e_i] \rangle$ we have $C'[e_j] \leq_c e'$ and thus also $C''[C'[e_j]]\Downarrow \implies C''[e']\Downarrow$ which immediately shows $C''[e']\Downarrow$. Since C'' was chosen arbitrarily, we have $e \leq_c e'$ and thus $C'[e]$ is a least upper bound. ◀

▶ **Example 2.53.** Let $e_0 := \bot$, $e_i := \texttt{Cons True}\ e_{i-1}$ for $i = 1, 2, \ldots$, and $\texttt{repTrue} \in \mathcal{F}$ with definition $\texttt{repTrue} = \texttt{Cons True repTrue}$. Then $\langle e_i \rangle$ is an ascending $\leq_c$-chain and $\text{club}(\langle e_i \rangle) = \texttt{repTrue}$.

A set of expressions $\mathcal{S} \subseteq E_{LNAME}$ is $\sim_c$-*closed* iff for all $e_1, e_2 \in E_{LNAME}$:

$$e_1 \sim_c e_2 \implies (e_1 \in \mathcal{S} \iff e_2 \in \mathcal{S}).$$

▶ **Definition 2.54.** A predicate $\mathcal{P} \subseteq E_{LNAME}$ is *admissible*, if $\mathcal{P}$ is $\sim_c$-closed and if for every ascending chain $\langle e_i \rangle$: $\mathcal{P}(\text{club}(\langle e_i \rangle))$ holds, whenever $\text{club}(\langle e_i \rangle)$ exists and there is an index i_0 s.t. for all $i \geq i_0$: $\mathcal{P}(e_i)$.

The advantage of admissible predicates is that they can be used to prove that a predicate holds for an expression by using a kind of fixpoint induction:

▶ **Proposition 2.55.** *Let $\mathcal{P}$ be an admissible predicate and let $\langle e_i \rangle$ be an ascending chain. If* $\text{club}(\langle e_i \rangle) = e$ *and for all i: $\mathcal{P}(e_i)$ holds, then also $\mathcal{P}(e)$ holds.*

▶ **Proposition 2.56.** *Let C_1, C_2 be multicontexts and for $i = 1, \ldots, n$ let $\emptyset \neq \mathcal{S}_i \subseteq E_{LNAME}$ be $\sim_c$-closed sets. Then the following predicates are admissible:*

1. $\mathcal{P}(x) := C_1[x, \cdots, x] \sim_c C_2[x, \cdots, x]$.
2. $\mathcal{P}(x) := \forall z_1 \in \mathcal{S}_1, \ldots, z_n \in \mathcal{S}_n.(C_1[x, \cdots, x] \sim_c C_2[x, \cdots, x])$ *where the z_i occur free in C_1 and / or C_2.*

Proof. **1.** Let $\langle e_i \rangle$ be an ascending chain s.t. $r := \text{club}(\langle e_i \rangle)$ exists and there is an index i_0 s.t. for all $i \geq i_0$: $\mathcal{P}(e_i) = C_1[e_i, \cdots, e_i] \sim_c C_2[e_i, \cdots, e_i]$ holds.
Consider the ascending chain $C_1[e_{i_0}, \cdots, e_{i_0}] \leq_c C_1[e_{i_0+1}, \cdots, e_{i_0+1}] \leq_c \ldots$. Then the expression $C_1[r, \cdots, r]$ is a club of this chain. Since $C_1[e_i, \cdots, e_i] \sim_c C_2[e_i, \cdots, e_i]$ for all $i \geq i_0$, $C_1[r, \cdots, r]$ is also a club of the chain $C_2[e_{i_0}] \leq_c C_2[e_{i_0+1}] \leq_c \ldots$ which also has $C_2[r, \cdots, r]$ as a club. Lemma 2.51 shows $C_1[r, \cdots, r] \sim_c C_2[r, \cdots, r]$ and thus $\mathcal{P}(r)$ holds.

2. Let $\langle e_i \rangle$ be an ascending chain s.t. $r := \text{club}(\langle e_i \rangle)$ exists and there is an index i_0 s.t. for all $i \geq i_0$: $\mathcal{P}(e_i) = \forall z_1, \ldots, z_n.C_1[e_i, \cdots, e_i] \sim_c C_2[e_i, \cdots, e_i]$ holds.
It is sufficient to show that $\mathcal{P}_\sigma(r)$ holds for all substitutions σ which substitute each of the variables z_i for an expression in $\mathcal{S}_i$ and where $\mathcal{P}_\sigma(x) = \sigma(C_1[x, \cdots, x] \sim_c C_2[x, \cdots, x])$.
Let $s_1 \in \mathcal{S}_i, \ldots, s_n \in \mathcal{S}_n$ be arbitrary but fixed and let $\sigma_1 = \{z_1 \mapsto s_1, \ldots, z_n \mapsto s_n\}$. The predicate

$$\mathcal{P}'_{\sigma_1} = (\ldots(\lambda z_1, \ldots, z_n.(C_1[x, \cdots, x]))\ s_1)\ldots s_n) \sim_c (\ldots(\lambda z_1, \ldots, z_n.(C_2[x, \cdots, x]))\ s_1)\ldots s_n)$$

is admissible (by the first item of the proposition) and thus $\mathcal{P}'_{\sigma_1}(r)$ holds.
Since $(\ldots(\lambda z_1, \ldots, z_n.(C_1[r, \cdots, r]))\ s_1) \ldots\ s_n) \sim_c \sigma(C_1[r, \cdots, r])$ (since (β) is correct), this shows $\mathcal{P}'_{\sigma_1} = \mathcal{P}_{\sigma_1}$ and thus $\mathcal{P}_{\sigma_1}(r)$ holds. Since $s_1, \ldots, s_n$ were chosen arbitrarily, this also shows $\mathcal{P}_\sigma(r)$ for all σ and thus $\mathcal{P}(r)$ also holds. ◀

In abuse of notation, we will often omit the sets $\mathcal{S}_i$ in our predicates, where we usually mean all expressions if not specified otherwise.

▶ **Example 2.57.** Note that in-equations are usually not admissible.
The predicate $\mathcal{P}(x) = x \not\sim_c r$ is not admissible: Let $l = \texttt{repeat True}$ and $l_i = \underbrace{\texttt{True} : \texttt{True} : \ldots : \texttt{True}}_{i} : \bot$. Then $l_i \not\sim_c l$ for all i, but $\text{club}_i(l_i) = l$. Since $l \sim_c l$, the predicate $\mathcal{P}$ is not admissible.

Clearly, there are further admissible predicates (i.e. Boolean combinations (using $\wedge$ or $\vee$) or also predicates that consider $\leq_c$ instead of $\sim_c$). However, we will not consider them in the following.

We will define expressions which evaluate to a list. Note that it is not possible to define those expressions by requiring that they are contextual equivalent to a list, since there are expressions that "evaluate" to an infinite list, but the syntax of *LNAME* does not include infinite structures. Thus we describe them by using a greatest fixpoint.

▶ **Definition 2.58.** The set of all *list expressions* is the greatest fixpoint of the following operator ψ on expressions. Let $M \subseteq E_{LNAME}$.

- If $e \Downarrow e'$ then $e \in \psi(M)$ iff either
  - $e' = \texttt{Nil}$, or
  - $e' = \texttt{Cons}\ e_1\ e_2$, where $e_2 \in M$.

The set of all *finite list expressions* is the least fixpoint of the operator ψ

So informally a list expression is any *LNAME*-expression which diverges, can be evaluated to a finite list, to an infinite list, or to a list whose n^{th} tail is a diverging expression (which is sometimes called a *partial list*). A finite list expression is any *LNAME*-expression which can be evaluated to a finite list.

▶ **Theorem 2.59** (Induction Scheme for Lists)**.** *Let $\mathcal{P}$ be an admissible predicate on list expressions such that*

1. $\mathcal{P}(\bot)$ *holds, and*
2. $\mathcal{P}(\texttt{Nil})$ *holds, and*
3. *for any list expression xs and any expression x: $\mathcal{P}(xs) \implies \mathcal{P}(\texttt{Cons}\ x\ xs)$*

Then $\mathcal{P}(xs)$ holds for all list expressions xs.

We sketch the main arguments of the correctness of the induction scheme.

The preconditions (2) and (3) inductively show that $\mathcal{P}(xs)$ holds for all *finite* lists xs and thus also for all expressions which generate finite lists (since these are contextually equivalent to a finite list).

Let $\texttt{Cons}\ x_1\ (\texttt{Cons}\ x_2\ (\texttt{Cons}\ (x_3 \ldots$ be an "infinite list" (or more precisely an expression which can be evaluated to such an infinite list). The sequence of partial lists $e_0 := \bot$, $e_1 := x_1 : \bot$, $e_2 := x_1 : x_2 : \bot$, $\ldots$ is an ascending chain, s.t. $e_i \leq_c e_{i+1}$ for all i, and $\text{club}(\langle e_i \rangle)$ is the infinite list (more precisely the expression which can be evaluated to such an infinite list).

The preconditions (1) and (3) inductively show that $\mathcal{P}(xs)$ holds for all *partial lists*, i.e. lists where the last tail is $\bot$, i.e. $\mathcal{P}(e_j)$ for every e_j in the ascending chain $\langle e_i \rangle$ holds. Now Proposition 2.55 shows that in this case also $\mathcal{P}(e)$ holds. Clearly, explicit infinite lists do not exist in the language, but there are expressions that generate such infinite lists.

▶ **Example 2.60.** Let $\mathsf{listid} \in \mathcal{F}$ with the definition

$$\mathsf{listid}\ xs = \texttt{case}_{List}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \texttt{Nil}; (\texttt{Cons}\ y\ ys) \to (\texttt{Cons}\ y\ (\mathsf{listid}\ ys))\}$$

We show that $\mathcal{P}(xs) := \mathsf{listid}\ xs \sim_c xs$ holds for all list expressions. We use Theorem 2.59 for the predicate $\mathcal{P}$.

∘ **Case $\mathcal{P}(\bot)$:**

$$\begin{array}{ll} & \mathsf{listid}\ \bot \\ \sim_c & \texttt{case}_{List}\ \bot\ \texttt{of}\ alts & \text{(since (SC}\beta\text{) is correct)} \\ \sim_c & \bot & \text{(since (case}\bot\text{) is correct)} \end{array}$$

∘ **Case $\mathcal{P}(\texttt{Nil})$:**

$$\begin{array}{ll} & \mathsf{listid}\ \texttt{Nil} \\ \sim_c & \texttt{case}_{List}\ \texttt{Nil}\ \texttt{of}\ alts & \text{(since (SC}\beta\text{) is correct)} \\ \sim_c & \texttt{Nil} & \text{(since (case) is correct)} \end{array}$$

∘ **Case $\mathcal{P}(xs) \implies \mathcal{P}(\texttt{Cons}\ x\ xs)$:** As induction hypothesis we assume that $\mathcal{P}(xs)$ holds for a fixed but arbitrary list expression xs. Let x be an arbitrary expression. We show $\mathcal{P}(\texttt{Cons}\ x\ xs)$:

$$
\begin{array}{lll}
& \mathsf{listid}\ (\texttt{Cons}\ x\ xs) & \\
\sim_c & \texttt{case}_{List}\ \texttt{Cons}\ x\ xs\ \texttt{of}\ alts & \text{(since (SC}\beta\text{) is correct)} \\
\sim_c & \texttt{Cons}\ x\ (\mathsf{listid}\ xs) & \text{(since (case) is correct)} \\
\sim_c & \texttt{Cons}\ x\ xs & \text{(induction hypothesis and since } \sim_c \text{ is a congruence)}
\end{array}
$$

We also define an induction scheme for finite lists:

▶ **Theorem 2.61** (Induction Scheme for Finite Lists). *Let $\mathcal{P}$ be a admissible predicate on list expressions such that*

1. *$\mathcal{P}(\texttt{Nil})$ holds, and*
2. *for any list expression xs and any expression x: $\mathcal{P}(xs) \implies \mathcal{P}(\texttt{Cons}\ x\ xs)$*

Then $\mathcal{P}(xs)$ holds for all finite list expressions xs.

We will use the induction scheme for list expressions, to show several equational laws for list processing functions.

Let us define several prominent supercombinators on lists.

▶ **Definition 2.62.** The supercombinators append, reverse, revacc are defined by

$$
\begin{array}{lll}
\mathsf{append}\ xs\ ys & = \texttt{case}_{List}\ xs\ \texttt{of}\ \{ & \\
& \quad \texttt{Nil} \to ys; & \\
& \quad (\texttt{Cons}\ u\ us) \to \texttt{Cons}\ u\ (\mathsf{append}\ us\ ys)\} & \\
\mathsf{reverse}\ xs & = \texttt{case}_{List}\ xs\ \texttt{of}\ \{ & \\
& \quad \texttt{Nil} \to \texttt{Nil}; & \\
& \quad (\texttt{Cons}\ u\ us) \to (\mathsf{append}\ (\mathsf{reverse}\ us)\ (\texttt{Cons}\ u\ \texttt{Nil}))\} & \\
\mathsf{revacc}\ xs\ acc & = \texttt{case}_{List}\ xs\ \texttt{of}\ \{ & \\
& \quad \texttt{Nil} \to acc; & \\
& \quad (\texttt{Cons}\ u\ us) \to \mathsf{revacc}\ us\ (\texttt{Cons}\ u\ acc)\} &
\end{array}
$$

▶ **Exercise 2.63.** Show that the following two program transformations are correct for all list expressions xs.

1. $\mathsf{append}\ \texttt{Nil}\ xs \to xs$
2. $\mathsf{append}\ xs\ \texttt{Nil} \to xs$

Note that for the first equation it is not necessary to use induction, while for the second transformation the induction scheme for list expressions can be used to show correctness.

▶ **Example 2.64.** We show that append is associative, i.e. for all list expressions xs, ys, zs the equation $\mathsf{append}\ xs\ (\mathsf{append}\ ys\ zs) \sim_c \mathsf{append}\ (\mathsf{append}\ xs\ ys)$ holds. We also use correctness of the transformations of Exercise 2.63.

Let $\mathcal{P}(xs) = \forall ys, zs.\mathsf{append}\ xs\ (\mathsf{append}\ ys\ zs) \sim_c \mathsf{append}\ (\mathsf{append}\ xs\ ys)\ zs$. We use the induction scheme from Theorem 2.59 for the predicate $\mathcal{P}$.

∘ **Case $\mathcal{P}(\bot)$:** Let ys, zs be arbitrary expressions. We transform the left hand side of the equation:

$$
\begin{array}{lll}
& \mathsf{append}\ \bot\ (\mathsf{append}\ ys\ zs) & \\
\sim_c & \texttt{case}_{List}\ \bot\ \texttt{of}\ alts & \text{(since (SC}\beta\text{) is correct)} \\
\sim_c & \bot & \text{(since (case}\bot\text{) is correct)}
\end{array}
$$

and the right hand side of the equation:

$$
\begin{array}{lll}
& \mathsf{append}\ (\mathsf{append}\ \bot\ ys)\ zs & \\
\sim_c & \texttt{case}_{List}\ (\mathsf{append}\ \bot\ ys)\ \texttt{of}\ alts & \text{(since (SC}\beta\text{) is correct)} \\
\sim_c & \texttt{case}_{List}\ (\texttt{case}_{List}\ \bot\ \texttt{of}\ alts)\ \texttt{of}\ alts & \text{(since (SC}\beta\text{) is correct)} \\
\sim_c & \texttt{case}_{List}\ \bot\ \texttt{of}\ alts & \text{(since (case}\bot\text{) is correct)} \\
\sim_c & \bot & \text{(since (case}\bot\text{) is correct)}
\end{array}
$$

Thus $\mathsf{append}\ \bot\ (\mathsf{append}\ ys\ zs) \sim_c \mathsf{append}\ (\mathsf{append}\ \bot\ ys)\ zs$ for all expressions ys, zs.

∘ **Case $\mathcal{P}(\texttt{Nil})$:** Again let ys, zs be arbitrary expressions. We transform the left hand side of the equation:

$$
\begin{array}{lll}
& \mathsf{append}\ \texttt{Nil}\ (\mathsf{append}\ ys\ zs) & \\
\sim_c & (\mathsf{append}\ ys\ zs) & \text{(Exercise 2.63)}
\end{array}
$$

and the right hand side of the equation:

$$
\begin{array}{lll}
& \mathsf{append}\ (\mathsf{append}\ \texttt{Nil}\ ys)\ zs & \\
\sim_c & \mathsf{append}\ ys\ zs & \text{(Exercise 2.63)}
\end{array}
$$

Thus $\mathsf{append}\ \texttt{Nil}\ (\mathsf{append}\ ys\ zs) \sim_c \mathsf{append}\ (\mathsf{append}\ \texttt{Nil}\ ys)\ zs$ for all expressions ys, zs.

∘ **Induction step:** As induction hypothesis we assume that for an arbitrary (but fixed) list expression xs, $\mathcal{P}(xs)$ holds, and show $\mathcal{P}(\texttt{Cons}\ x\ xs)$ where x is an arbitrary expression. Let ys, zs be arbitrary expressions. We transform the left hand side of the equation:

$$
\begin{array}{lll}
& \mathsf{append}\ (\texttt{Cons}\ x\ xs)\ (\mathsf{append}\ ys\ zs) & \\
\sim_c & \texttt{case}_{List}\ (\texttt{Cons}\ x\ xs)\ \texttt{of}\ \{ & \\
& \quad \texttt{Nil} \to (\mathsf{append}\ ys\ zs); & \\
& \quad (\texttt{Cons}\ u\ us) \to \texttt{Cons}\ u\ (\mathsf{append}\ us\ (\mathsf{append}\ ys\ zs))\} & \\
& & \text{(since (SC}\beta\text{) is correct)} \\
\sim_c & \texttt{Cons}\ x\ (\mathsf{append}\ xs\ (\mathsf{append}\ ys\ zs)) & \text{(since (case) is correct)} \\
\sim_c & \texttt{Cons}\ x\ (\mathsf{append}\ (\mathsf{append}\ xs\ ys)\ zs) & \\
& & \text{(induction hypothesis and since } \sim_c \text{ is a congruence)}
\end{array}
$$

and the right hand side of the equation:

$$\begin{array}{lr}
\mathsf{append}\ (\mathsf{append}\ (\mathtt{Cons}\ x\ xs)\ ys)\ zs & \\
\sim_c \mathsf{append}\ (\mathtt{case}_{List}\ (\mathtt{Cons}\ x\ xs)\ \mathtt{of}\ \{ & \\
\qquad\qquad \mathtt{Nil} \to ys; & \\
\qquad\qquad (\mathtt{Cons}\ u\ us) \to \mathtt{Cons}\ u\ (\mathsf{append}\ us\ ys)\})\ zs & \\
 & \text{(since (SC}\beta\text{) is correct)} \\
\sim_c \mathsf{append}\ (\mathtt{Cons}\ x\ (\mathsf{append}\ xs\ ys))\ zs & \text{(since (case) is correct)} \\
\sim_c \mathtt{case}_{List}\ (\mathtt{Cons}\ x\ (\mathsf{append}\ xs\ ys))\ \mathtt{of}\{ & \text{(since (SC}\beta\text{) is correct)} \\
\qquad \mathtt{Nil} \to zs; & \\
\qquad (\mathtt{Cons}\ u\ us) \to \mathtt{Cons}\ u\ (\mathsf{append}\ us\ zs)\} & \\
\sim_c \mathtt{Cons}\ x\ (\mathsf{append}\ (\mathsf{append}\ xs\ ys)\ zs) & \text{(since (case) is correct)}
\end{array}$$

Thus $\mathsf{append}\ (\mathtt{Cons}\ x\ xs)\ (\mathsf{append}\ ys\ zs) \sim_c \mathsf{append}\ (\mathsf{append}\ (\mathtt{Cons}\ x\ xs)\ ys)\ zs$ holds for all expressions ys, zs.

▶ **Example 2.65.** We show that the equivalence

$$\mathsf{reverse}\ (\mathsf{append}\ xs\ ys) \sim_c \mathsf{append}\ (\mathsf{reverse}\ ys)\ (\mathsf{reverse}\ xs)$$

holds for all *finite* list expressions xs and all list expressions ys. Note that the equivalence does not hold for all lists, since e.g. for $ys = \mathtt{Cons\ True\ Nil}$ and $xs = \mathsf{repTrue}$ $\mathsf{reverse}\ (\mathsf{append}\ xs\ ys)\Uparrow$ while $\mathsf{append}\ (\mathsf{reverse}\ ys)\ (\mathsf{reverse}\ xs)\Downarrow$.

We use the induction scheme for finite lists (Theorem 2.61).

Let $\mathcal{P}(xs) = \forall ys.\mathsf{reverse}\ (\mathsf{append}\ xs\ ys) \sim_c \mathsf{append}\ (\mathsf{reverse}\ ys)\ (\mathsf{reverse}\ xs)$. We use Theorem 2.61 for the predicate $\mathcal{P}$.

- **Case $\mathcal{P}(\mathtt{Nil})$:** Let ys be an arbitrary expression. We show that we can transform the left hand side into the right hand side of the equation.

$$\begin{array}{lr}
\mathsf{reverse}\ (\mathsf{append}\ \mathtt{Nil}\ ys) & \\
\sim_c \mathsf{reverse}\ ys & \text{(Exercise 2.63)} \\
\sim_c \mathsf{append}\ (\mathsf{reverse}\ ys)\ \mathtt{Nil} & \text{(Exercise 2.63)} \\
\sim_c \mathsf{append}\ (\mathsf{reverse}\ ys)\ (\mathsf{reverse}\ \mathtt{Nil}) & \text{(correctness of (SC}\beta\text{) and (case))}
\end{array}$$

- **Case $\mathcal{P}(xs) \implies \mathcal{P}(\mathtt{Cons}\ x\ xs)$:** Assume that $\mathcal{P}(xs)$ holds for some arbitrary but fixed finite list expression xs, and let x be an arbitrary expression. We show $\mathcal{P}(\mathtt{Cons}\ x\ xs)$. Let ys be an arbitrary list expression.

$$\begin{array}{lr}
\mathsf{reverse}\ (\mathsf{append}\ (\mathtt{Cons}\ x\ xs)\ ys) & \\
\sim_c \mathsf{reverse}\ (\mathtt{Cons}\ x\ (\mathsf{append}\ xs\ ys)) & \text{(correctness of (SC}\beta\text{) and (case))} \\
\sim_c \mathsf{append}\ (\mathsf{reverse}\ ((\mathsf{append}\ xs\ ys)))\ (\mathtt{Cons}\ x\ \mathtt{Nil}) & \\
 & \text{(correctness of (SC}\beta\text{) and (case))} \\
\sim_c \mathsf{append}\ (\mathsf{append}\ (\mathsf{reverse}\ ys)\ (\mathsf{reverse}\ xs))\ (\mathtt{Cons}\ x\ \mathtt{Nil}) & \text{(induction hypothesis)} \\
\sim_c \mathsf{append}\ (\mathsf{reverse}\ ys)\ (\mathsf{append}\ (\mathsf{reverse}\ xs)\ (\mathtt{Cons}\ x\ \mathtt{Nil})) & \text{(Example 2.64)} \\
\sim_c \mathsf{append}\ (\mathsf{reverse}\ ys)\ (\mathsf{reverse}\ (\mathtt{Cons}\ x\ xs)) & \text{(correctness of (SC}\beta\text{) and (case))}
\end{array}$$

▶ **Example 2.66.** We show that the equivalence

$$\mathsf{reverse}\ xs \sim_c \mathsf{revacc}\ xs\ \mathtt{Nil}$$

holds for all list expressions xs. However, the induction proof will fail for the predicate $\mathcal{P}(xs) = \mathsf{reverse}\ xs \sim_c \mathsf{revacc}\ xs\ \mathtt{Nil}$, since in the induction step we get the expression $\mathsf{append}\ (\mathsf{revacc}\ xs\ \mathtt{Nil})\ (\mathtt{Cons}\ x\ \mathtt{Nil})$ for the left hand side (after applying the induction hypothesis), and $\mathsf{revacc}\ xs\ (\mathtt{Cons}\ x\ \mathtt{Nil})$ for the right hand side and there is no rule to make both expressions equal.

Hence, we prove a more general claim and show that the predicate

$$\mathcal{P}(xs) := \forall acc.\mathsf{append}\ (\mathsf{reverse}\ xs)\ acc \sim_c \mathsf{revacc}\ xs\ acc$$

holds for all list expressions xs. The original equivalence can then be derived by setting $acc = \mathtt{Nil}$ and applying the equation $\mathsf{append}\ (\mathsf{reverse}\ xs)\ \mathtt{Nil} \sim_c \mathsf{reverse}\ xs$ (proved in Exercise 2.63).

We use Theorem 2.59 for the predicate $\mathcal{P}$.

- **Case $\mathcal{P}(\bot)$:** Let e be arbitrary but fixed. We show $\mathsf{append}\ (\mathsf{reverse}\ \bot)\ e \sim_c \mathsf{revacc}\ \bot\ e$.

$$\begin{array}{lr}
\mathsf{append}\ (\mathsf{reverse}\ \bot)\ e & \\
\sim_c \mathsf{append}\ \bot\ e & \text{(correctness of (SC}\beta\text{) and (case}\bot\text{))} \\
\sim_c \bot & \text{(correctness of (SC}\beta\text{) and (case}\bot\text{))} \\
\sim_c \mathsf{revacc}\ \bot\ e & \text{(correctness of (SC}\beta\text{) and (case}\bot\text{))}
\end{array}$$

- **Case $\mathcal{P}(\mathtt{Nil})$:** Let e be arbitrary but fixed. We show $\mathsf{append}\ (\mathsf{reverse}\ \mathtt{Nil})\ e \sim_c \mathsf{revacc}\ \mathtt{Nil}\ e$.

$$\begin{array}{lr}
\mathsf{append}\ (\mathsf{reverse}\ \mathtt{Nil})\ e & \\
\sim_c \mathsf{append}\ \mathtt{Nil}\ e & \text{(correctness of (SC}\beta\text{) and (case))} \\
\sim_c e & \text{(Exercise 2.63)} \\
\sim_c \mathsf{revacc}\ \mathtt{Nil}\ e & \text{(correctness of (SC}\beta\text{) and (case))}
\end{array}$$

- **Case $\mathcal{P}(xs) \implies \mathcal{P}(\mathtt{Cons}\ x\ xs)$:** Assume that $\mathcal{P}(xs)$ holds for some arbitrary list expression xs, and let x, e be arbitrary expressions.
  We show $\mathsf{append}\ (\mathsf{reverse}\ (\mathtt{Cons}\ x\ xs))\ e \sim_c \mathsf{revacc}\ (\mathtt{Cons}\ x\ xs)\ e$.

$$\begin{array}{lr}
\mathsf{append}\ (\mathsf{reverse}\ (\mathtt{Cons}\ x\ xs))\ e & \\
\sim_c \mathsf{append}\ (\mathsf{append}\ (\mathsf{reverse}\ xs)\ (\mathtt{Cons}\ x\ \mathtt{Nil}))\ e & \\
 & \text{(correctness of (SC}\beta\text{) and (case))} \\
\sim_c \mathsf{append}\ (\mathsf{reverse}\ xs)\ (\mathsf{append}\ ((\mathtt{Cons}\ x\ \mathtt{Nil})\ e)) & \\
 & \text{(associativity of } \mathsf{append}\text{, Example 2.64)} \\
\sim_c \mathsf{append}\ (\mathsf{reverse}\ xs)\ (\mathtt{Cons}\ x\ e) & \text{(correctness of (SC}\beta\text{), (case))} \\
\sim_c \mathsf{revacc}\ xs\ (\mathtt{Cons}\ x\ e) & \text{(induction hypothesis)} \\
\sim_c \mathsf{revacc}\ (\mathtt{Cons}\ x\ xs)\ e & \text{(correctness of (SC}\beta\text{) and (case))}
\end{array}$$

  Note that the universal quantification in the predicate is crucial, since the induction hypothesis is applied for the accumulator being $\mathtt{Cons}\ x\ e$.

▶ **Exercise 2.67.** Show that the program transformation

$$\mathsf{reverse}\ (\mathsf{reverse}\ xs) \to xs$$

is *not* correct for all lists xs.

Show that the transformation

$$\mathsf{reverse}\ (\mathsf{reverse}\ xs) \to xs$$

is correct for all *finite* lists where you can reuse the result from Example 2.65.

Note that there are far more equations and program transformation which hold for list expressions and can be shown correct by the induction schemes. Several interesting equations can be found in [5, 6]. Note also that there are similar induction schemes for other data types than lists, and note also that there is an induction scheme for all (also untyped) expressions where as an additional base case, also $\mathcal{P}(\lambda x.e)$ for all e has to be shown.

3 The Call-by-Need Lambda Calculus LR

In this section we introduce and consider the calculus LR [37, 36, 35], which syntactically extends the calculus *LNAME* by recursive `letrec`-expressions and it does not include supercombinators, since they can also be defined by using `letrec`-expressions.

Semantically[6], the LR-calculus is different from *LNAME*, since it uses a call-by-need reduction, i.e. instead of copying and substituting arbitrary expressions – as in *LNAME* – the operational semantics uses sharing to avoid duplicating unevaluated expressions and to avoid duplicated evaluations.

The call-by-need strategy is used in all efficient implementations of lazy functional programming languages like Haskell, so it makes sense to analyze its semantics.

We introduce `letrec`-*expressions*. A `letrec`-expression is of the form

$$\texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e$$

where the variables $x_1, \ldots, x_n$ must be pairwise distinct. The construct $x_i = e_i$ is called a `letrec`-*binding* and the collection of bindings $x_1 = e_1, \ldots, x_n = e_n$ is called the `letrec`-*environment*. We view a `letrec`-environment as a set of bindings and thus the order of the bindings is irrelevant. The scope of the variables $x_1, \ldots, x_n$ are all expressions $e_1, \ldots, e_n$ and e, i.e. the bindings are *recursive*. To abbreviate the `letrec`-environment – or parts of it – we will use the meta-symbol *Env* as a placeholder for a non-empty set of `letrec`-bindings. We also write $\{x_i = e_i\}_{i=1}^n$ as an abbreviation for an environment $x_1 = e_1, \ldots, x_n = e_n$. For a chain of variable-to-variable bindings $x_j = x_{j-1}, x_{j+1} = x_j, \ldots, x_m = x_{m-1}$ we use the abbreviation $\{x_i = x_{i-1}\}_{i=j}^m$.

Using `letrec`-expressions, we can define recursive functions "inline" as expressions and thus supercombinators and their definitions (as in *LNAME*) are not required.

▶ **Example 3.1.** The function *map* which maps a function to all elements of a list can be defined using `letrec` and `case` as follows:

$$\begin{array}{l}\texttt{letrec}\ \mathit{map} = \lambda f, xs.\texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\\ \qquad\qquad \{\texttt{Nil} \to \texttt{Nil};\\ \qquad\qquad (\texttt{Cons}\ y\ ys) \to \texttt{Cons}\ (f\ y)\ (\mathit{map}\ f\ ys)\}\\ \texttt{in}\ \mathit{map}\end{array}$$

[6] Here we mean the operational behavior, from the viewpoint of contextual equivalence and the corresponding induced semantics it was shown in [36] that the languages are isomorphic w.r.t the equivalence classes of $\sim_c$. However, this changes if the languages are extended by non-determinism (see e.g. [16, 25, 27]) or when evaluation steps are counted (see e.g. [15, 14, 35]).

We formally introduce the syntax of LR:

▶ **Definition 3.2.** The syntax of the language LR is defined by the following grammar, where $x, x_i \in \mathit{Var}$ and $c_{T,i} \in D_T$:

$$\begin{array}{lcl} e, e_i \in E_{LR} & ::= & x \mid \lambda x.e \mid (e_1\ e_2) \mid (c_{T,i}\ e_1\ \ldots\ e_{ar(c_{T,i})})\\ & & \mid \texttt{case}_T\ e\ \texttt{of}\ \{pat_{T,1} \to e_1; \ldots; pat_{T,|T|} \to e_{|T|}\} \mid \texttt{seq}\ e_1\ e_2\\ & & \mid \texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e\\ pat_{T,i} & ::= & (c_{T,i}\ x_{i,1}\ \ldots\ x_{i,ar(c_{T,i})}) \end{array}$$

The sets $FV(e)$ of free and $BV(e)$ of bound variables of an expression e are defined as in *LNAME* (see Figure 1) where additionally the following holds for `letrec`-expressions:

$$\begin{array}{lcl} FV(\texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e) & = & \left(FV(e) \cup \bigcup_{i=1}^n FV(e_i)\right) \setminus \{x_1, \ldots, x_n\}\\ BV(\texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e) & = & \left(BV(e) \cup \bigcup_{i=1}^n BV(e_i)\right) \cup \{x_1, \ldots, x_n\} \end{array}$$

We again assume that the distinct variable convention holds, i.e. bound variables are pairwise distinct, and bound variables are distinct from free variables. We assume that α-renaming is implicitly performed to always obey this convention.

A *context* is an expression with a hole $[\cdot]$ at expression position, i.e. contexts in LR are generated by the following grammar:

$$\begin{array}{rcl} C \in C_{LR} & ::= & [\cdot] \mid \lambda x.C \mid (C\ e) \mid (e\ C) \mid (c_{T,i}\ e_1\ \ldots\ e_{i-1}\ C\ e_{i+1} \ldots\ e_{ar(c_{T,i})})\\ & & \mid (\texttt{seq}\ C\ e) \mid (\texttt{seq}\ e\ C)\\ & & \mid (\texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ C)\\ & & \mid (\texttt{letrec}\ x_1 = e_1, \ldots, x_i = C, \ldots, x_n = e_n\ \texttt{in}\ e)\\ & & \mid (\texttt{case}_T\ C\ \texttt{of}\ \mathit{alts}) \mid (\texttt{case}_T\ e\ \texttt{of}\ \{\ldots; (c_{T,i}\ x_1 \ldots x_{ar(c_{T,i})}) \to C; \ldots\}) \end{array}$$

Surface contexts S are contexts where the hole is not in an abstraction, i.e. they are generated by the grammar:

$$\begin{array}{rcl} S \in S_{LR} & ::= & [\cdot] \mid (S\ e) \mid (e\ S) \mid (c_{T,i}\ e_1\ \ldots\ e_{i-1}\ S\ e_{i+1} \ldots\ e_{ar(c_{T,i})})\\ & & \mid (\texttt{seq}\ S\ e) \mid (\texttt{seq}\ e\ S)\\ & & \mid (\texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ S)\\ & & \mid (\texttt{letrec}\ x_1 = e_1, \ldots, x_i = S, \ldots, x_n = e_n\ \texttt{in}\ e)\\ & & \mid (\texttt{case}_T\ S\ \texttt{of}\ \mathit{alts}) \mid (\texttt{case}_T\ e\ \texttt{of}\ \{\ldots; (c_{T,i}\ x_1 \ldots x_{ar(c_{T,i})}) \to S; \ldots\}) \end{array}$$

3.1 Normal Order Reduction

A *value* in LR is an abstraction $\lambda x.e$ or a constructor application $(c\ e_1\ \ldots\ e_{ar(c)})$. The reduction rules of the calculus are defined in Figure 2, where the role of the labels sub, top, vis, and nontarg will be explained below in Definition 3.4. The calculus LR has much more reduction rules than *LNAME*, since the call-by-need evaluation carefully avoids duplication of work, i.e. of redexes. We briefly explain the different reduction rules: The rule (lbeta) is the sharing variant of classical β-reduction which avoids substitution of the argument into the body of the abstraction, and thus it shares the argument of an application by a new `letrec`-binding.

The rules (cp-in) and (cp-e) copy abstractions. The rules (llet-in) and (llet-e) join two `letrec`-environments. The rules (lapp), (lcase), and (lseq) float-out a `letrec` from the first

(lbeta) $C[((\lambda x.e_1)^{\mathsf{sub}}\ e_2)] \to C[(\mathtt{letrec}\ x = e_2\ \mathtt{in}\ e_1)]$

(cp-in) $(\mathtt{letrec}\ x_1 = (\lambda x.e)^{\mathsf{sub}}, \{x_i = x_{i-1}\}_{i=2}^m, Env\ \ \mathtt{in}\ C[x_m^{\mathsf{vis}}])$
$\to (\mathtt{letrec}\ x_1 = (\lambda x.e), \{x_i = x_{i-1}\}_{i=2}^m, Env\ \ \mathtt{in}\ C[(\lambda x.e)])$

(cp-e) $(\mathtt{letrec}\ x_1 = (\lambda x.e_1)^{\mathsf{sub}}, \{x_i = x_{i-1}\}_{i=2}^m, Env, y = C[x_m^{\mathsf{vis}}]\ \mathtt{in}\ e_2)$
$\to (\mathtt{letrec}\ x_1 = (\lambda x.e_1), \{x_i = x_{i-1}\}_{i=2}^m, Env, y = C[(\lambda x.e_1)]\ \mathtt{in}\ e_2)$

(llet-in) $(\mathtt{letrec}\ Env_1\ \mathtt{in}\ (\mathtt{letrec}\ Env_2\ \mathtt{in}\ e)^{\mathsf{sub}})$
$\to (\mathtt{letrec}\ Env_1, Env_2\ \mathtt{in}\ e)$

(llet-e) $(\mathtt{letrec}\ Env_1, x = (\mathtt{letrec}\ Env_2\ \mathtt{in}\ e_x)^{\mathsf{sub}}\ \mathtt{in}\ e)$
$\to (\mathtt{letrec}\ Env_1, Env_2, x = e_x\ \mathtt{in}\ e)$

(lapp) $C[((\mathtt{letrec}\ Env\ \mathtt{in}\ e_1)^{\mathsf{sub}}\ e_2)] \to C[(\mathtt{letrec}\ Env\ \mathtt{in}\ (e_1\ e_2))]$

(lcase) $C[(\mathtt{case}_T\ (\mathtt{letrec}\ Env\ \mathtt{in}\ e)^{\mathsf{sub}}\ \mathtt{of}\ alts)]$
$\to C[(\mathtt{letrec}\ Env\ \mathtt{in}\ (\mathtt{case}_T\ e\ \mathtt{of}\ alts))]$

(lseq) $C[(\mathtt{seq}\ (\mathtt{letrec}\ Env\ \mathtt{in}\ e_1)^{\mathsf{sub}}\ e_2)] \to C[(\mathtt{letrec}\ Env\ \mathtt{in}\ (\mathtt{seq}\ e_1\ e_2))]$

(seq-c) $C[(\mathtt{seq}\ v^{\mathsf{sub}}\ e)] \to C[e]$ if v is a value

(seq-in) $(\mathtt{letrec}\ x_1 = (c\ \overrightarrow{e})^{\mathsf{sub}}, \{x_i = x_{i-1}\}_{i=2}^m, Env\ \mathtt{in}\ C[(\mathtt{seq}\ x_m^{\mathsf{vis}}\ e')])$
$\to (\mathtt{letrec}\ x_1 = (c\ \overrightarrow{e}), \{x_i = x_{i-1}\}_{i=2}^m, Env\ \mathtt{in}\ C[e'])$

(seq-e) $(\mathtt{letrec}\ x_1 = (c\ \overrightarrow{e})^{\mathsf{sub}}, \{x_i = x_{i-1}\}_{i=2}^m, Env, y = C[(\mathtt{seq}\ x_m^{\mathsf{vis}}\ e')]\ \mathtt{in}\ e'')$
$\to (\mathtt{letrec}\ x_1 = (c\ \overrightarrow{e}), \{x_i = x_{i-1}\}_{i=2}^m, Env, y = C[e']\ \mathtt{in}\ e'')$

(case-c) $C[\mathtt{case}_T\ (c_{T,i}\ \overrightarrow{e})^{\mathsf{sub}}\ \mathtt{of}\ \{\ldots; (c_{T,i}\ \overrightarrow{y}) \to e'; \ldots\}]$
$\to C[(\mathtt{letrec}\ \{y_i = e_i\}_{i=1}^n\ \mathtt{in}\ e')]$ if $n = ar(c_{T,i}) \geq 1$

(case-c) $C[\mathtt{case}_T\ c_{T,i}^{\mathsf{sub}}\mathtt{of}\{\ldots; c_{T,i} \to e; \ldots\}] \to C[e]$ if $ar(c_{T,i}) = 0$

(case-in) $\mathtt{letrec}\ x_1 = (c_{T,i}\ \overrightarrow{e})^{\mathsf{sub}}, \{x_i = x_{i-1}\}_{i=2}^m, Env$
$\mathtt{in}\ C[\mathtt{case}_T\ x_m^{\mathsf{vis}}\ \mathtt{of}\{\ldots; (c_{T,i}\ \overrightarrow{z}) \to e'; \ldots\}]$
$\to \mathtt{letrec}\ x_1 = (c_{T,i}\ \overrightarrow{y}), \{y_i = e_i\}_{i=1}^n, \{x_i = x_{i-1}\}_{i=2}^m, Env$
$\mathtt{in}\ C[(\mathtt{letrec}\ \{z_i = y_i\}_{i=1}^n\ \mathtt{in}\ e')]$
where $n = ar(c_{T,i}) \geq 1$ and y_i are fresh variables

(case-in) $\mathtt{letrec}\ x_1 = c_{T,i}^{\mathsf{sub}}, \{x_i = x_{i-1}\}_{i=2}^m, Env\ \mathtt{in}\ C[\mathtt{case}_T\ x_m^{\mathsf{vis}}\ \mathtt{of}\{\ldots; (c_i \to e); \ldots\}]$
$\to \mathtt{letrec}\ x_1 = c_{T,i}, \{x_i = x_{i-1}\}_{i=2}^m, Env\ \mathtt{in}\ \ C[e],$ if $ar(c_{T,i}) = 0$

(case-e) $\mathtt{letrec}\ x_1 = (c_{T,i}\ \overrightarrow{e})^{\mathsf{sub}}, \{x_i = x_{i-1}\}_{i=2}^m,$
$u = C[\mathtt{case}_T\ x_m^{\mathsf{vis}}\mathtt{of}\{\ldots; (c_{T,i}\ \overrightarrow{z}) \to e'; \ldots\}], Env\ \mathtt{in}\ e''$
$\to \mathtt{letrec}\ x_1 = (c_{T,i}\ \overrightarrow{y}), \{y_i = e_i\}_{i=1}^n, \{x_i = x_{i-1}\}_{i=2}^m,$
$u = C[(\mathtt{letrec}\ \{z_i = y_i\}_{i=1}^n\ \mathtt{in}\ e')], Env\ \mathtt{in}\ e''$
where $n = ar(c_{T,i}) \geq 1$ and y_i are fresh variables

(case-e) $\mathtt{letrec}\ x_1 = c_{T,i}^{\mathsf{sub}}, \{x_i = x_{i-1}\}_{i=2}^m,$
$u = C[\mathtt{case}_T\ x_m^{\mathsf{vis}}\ \mathtt{of}\ \{\ldots; (c_{T,i} \to e'); \ldots\}], Env\ \mathtt{in}\ e''$
$\to \mathtt{letrec}\ x_1 = c_{T,i}, \{x_i = x_{i-1}\}_{i=2}^m, \ldots, u = C[e'], Env\ \mathtt{in}\ e''$ if $ar(c_{T,i}) = 0$

Figure 2 Reduction rules of LR

argument of an application, a `case`-expression, or a `seq`-expression. The rules (seq-c), (seq-in), and (seq-e) evaluate a `seq`-expression, provided that the first argument is a value (or a variable that is bound (via indirections) to a constructor application). The rules (case-c), (case-in), and (case-e) evaluate a `case`-expression provided that the first argument is (or is a variable which is bound to) a constructor application of the right type.

▶ **Definition 3.3.** We define unions for the rules in Figure 2: (case) is the union of (case-c), (case-in), (case-e); (seq) is the union of (seq-c), (seq-in), (seq-e); (cp) is the union of (cp-in), (cp-e); (llet) is the union of (llet-in), (llet-e); and (lll) is the union of (llet), (lapp), (lcase), and (lseq).

The normal order reduction strategy of the calculus LR is a call-by-need strategy, which is a call-by-name strategy adapted to sharing. It applies the reduction rules at specific positions. Instead of defining the call-by-need evaluation in terms of a syntactic definition of reduction contexts (using a context free grammar), we provide an algorithm to find the position of a redex and also to describe the syntactic form of so-called reduction contexts[7].

▶ **Definition 3.4** (Labeling Algorithm)**.** The labeling algorithm detects the position to which a reduction rule will be applied according to normal order. It uses the labels: top,sub,vis,nontarg where top means reduction of the top term, sub means reduction of a subexpression, vis marks already visited subexpressions, and nontarg marks already visited variables that are not target of a (cp)-reduction. For an expression e the labeling algorithm starts with e^{top}, where no other subexpression in s is labeled and proceeds by applying the rules given in Figure 3 exhaustively.

Note that the labeling algorithm does not descend into sub-labeled `letrec`-expressions. If the labeling algorithm does not fail, then a potential normal order redex is found, which can only be a superterm of the sub-marked subexpression. However, it is possible that there is no normal order reduction, if the evaluation is already finished, or no rule is applicable.

Note that the labeling algorithm to compute reduction positions does not remove labels (but sometimes replaces them by other labels). This is a difference to labeling algorithms presented before for the lambda calculus and for the calculus $LNAME$. A further difference is that the rules are only applied to the given meta-expression and not inside the expressions.

▶ **Definition 3.5** (Normal Order Reduction of LR)**.** Let e be an expression. Then a single *normal order reduction step* $e \xrightarrow{no} e'$ is defined by first applying the labeling algorithm to e, and if the labeling algorithm terminates successfully, then one of the rules in Figure 2 has to be applied, if possible, where the labels sub, vis must match the labels in the expression e (e may have more labels), and e' is the result after erasing all labels.

▶ **Example 3.6.** For $(\mathtt{letrec}\ w = \lambda x.x, y = w, z = (y\ y)\ \mathtt{in}\ z)$ the labeling algorithm proceeds as follows:

$$
\begin{aligned}
&(\mathtt{letrec}\ w = \lambda x.x, y = w, z = (y\ y)\ \mathtt{in}\ z)^{\mathsf{top}}\\
&\to (\mathtt{letrec}\ w = \lambda x.x, y = w, z = (y\ y)\ \mathtt{in}\ z^{\mathsf{sub}})^{\mathsf{vis}}\\
&\to (\mathtt{letrec}\ w = \lambda x.x, y = w, z = (y\ y)^{\mathsf{sub}}\ \mathtt{in}\ z^{\mathsf{vis}})^{\mathsf{vis}}\\
&\to (\mathtt{letrec}\ w = \lambda x.x, y = w, z = (y^{\mathsf{sub}}\ y)^{\mathsf{vis}}\ \mathtt{in}\ z^{\mathsf{vis}})^{\mathsf{vis}}\\
&\to (\mathtt{letrec}\ w = \lambda x.x, y = w^{\mathsf{sub}}, z = (y^{\mathsf{vis}}\ y)^{\mathsf{vis}}\ \mathtt{in}\ z^{\mathsf{vis}})^{\mathsf{vis}}\\
&\to (\mathtt{letrec}\ w = (\lambda x.x)^{\mathsf{sub}}, y = w^{\mathsf{nontarg}}, z = (y^{\mathsf{vis}}\ y)^{\mathsf{vis}}\ \mathtt{in}\ z^{\mathsf{vis}})^{\mathsf{vis}}
\end{aligned}
$$

[7] However, a syntactic (and rather complex) definition of reduction contexts by a context free grammar and the corresponding description of normal order reduction for the calculus LR can be found in [37].

$(e_1\ e_2)^{\mathsf{sub}\vee\mathsf{top}}$	$\to (e_1^{\mathsf{sub}}\ e_2)^{\mathsf{vis}}$
$(\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ e)^{\mathsf{top}}$	$\to (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ e^{\mathsf{sub}})^{\mathsf{vis}}$
$(\texttt{letrec}\ x = e, \mathit{Env}\ \texttt{in}\ C[x^{\mathsf{sub}}])$	$\to (\texttt{letrec}\ x = e^{\mathsf{sub}}, \mathit{Env}\ \texttt{in}\ C[x^{\mathsf{vis}}])$
$(\texttt{letrec}\ x = e, y = C[x^{\mathsf{sub}}], \mathit{Env}\ \texttt{in}\ e')$	$\to (\texttt{letrec}\ x = e^{\mathsf{sub}}, y = C[x^{\mathsf{vis}}], \mathit{Env}\ \texttt{in}\ e')$, where C is not trivial
$(\texttt{letrec}\ x = e, y = x^{\mathsf{sub}}, \mathit{Env}\ \texttt{in}\ e')$	$\to (\texttt{letrec}\ x = e^{\mathsf{sub}}, y = x^{\mathsf{nontarg}}, \mathit{Env}\ \texttt{in}\ e')$
$(\texttt{seq}\ e_1\ e_2)^{\mathsf{sub}\vee\mathsf{top}}$	$\to (\texttt{seq}\ e_1^{\mathsf{sub}}\ e_2)^{\mathsf{vis}}$
$(\texttt{case}_T\ e\ \texttt{of}\ \mathit{alts})^{\mathsf{sub}\vee\mathsf{top}}$	$\to (\texttt{case}\ e^{\mathsf{sub}}\ \texttt{of}\ \mathit{alts})^{\mathsf{vis}}$
$\texttt{letrec}\ x = e^{\mathsf{vis}\vee\mathsf{nontarg}}, y = C[x^{\mathsf{sub}}], \mathit{Env}\ \texttt{in}\ e'$	$\to \texttt{Fail}$
$(\texttt{letrec}\ x = C[x^{\mathsf{sub}}], \mathit{Env}\ \texttt{in}\ e)$	$\to \texttt{Fail}$

Figure 3 Computing reduction positions using labels (see Definition 3.4), where $a \vee b$ means label a or label b. The algorithm does not overwrite non-displayed labels.

The only reduction rule matching the subexpressions and the labels is the rule (cp-e), i.e. the normal order reduction is:

$$(\texttt{letrec}\ w = \lambda x.x, y = w, z = (y\ y)\ \texttt{in}\ z) \xrightarrow{no} (\texttt{letrec}\ w = \lambda x.x, y = w, z = ((\lambda x'.x')\ y)\ \texttt{in}\ z)$$

where the α-renaming is performed implicitly to ensure that the distinct variable convention holds.

Note that the label nontarg at the variable w prevents the normal order reduction to copy the sub-labeled abstraction into this position.

For the expression $\texttt{letrec}\ x = (y\ y), y = (x\ x)\ \texttt{in}\ y$ the labeling fails:

$$\begin{aligned}
&(\texttt{letrec}\ x = (y\ y), y = (x\ x)\ \texttt{in}\ y)^{\mathsf{top}}\\
&\to (\texttt{letrec}\ x = (y\ y), y = (x\ x)\ \texttt{in}\ y^{\mathsf{sub}})^{\mathsf{vis}}\\
&\to (\texttt{letrec}\ x = (y\ y), y = (x\ x)^{\mathsf{sub}}\ \texttt{in}\ y^{\mathsf{vis}})^{\mathsf{vis}}\\
&\to (\texttt{letrec}\ x = (y\ y), y = (x^{\mathsf{sub}}\ x)^{\mathsf{vis}}\ \texttt{in}\ y^{\mathsf{vis}})^{\mathsf{vis}}\\
&\to (\texttt{letrec}\ x = (y\ y)^{\mathsf{sub}}, y = (x^{\mathsf{vis}}\ x)^{\mathsf{vis}}\ \texttt{in}\ y^{\mathsf{vis}})^{\mathsf{vis}}\\
&\to (\texttt{letrec}\ x = (y^{\mathsf{sub}}\ y)^{\mathsf{vis}}, y = (x^{\mathsf{vis}}\ x)^{\mathsf{vis}}\ \texttt{in}\ y^{\mathsf{vis}})^{\mathsf{vis}}\\
&\to \texttt{Fail}
\end{aligned}$$

and thus the expression has no normal order reduction.

For the expression $(\texttt{Cons True Nil})\ (\lambda x.x)$, the labeling terminates successfully with $((\texttt{Cons True Nil})^{\mathsf{sub}}\ (\lambda x.x))^{\mathsf{vis}}$ and also for the expression $\texttt{letrec}\ x = \texttt{True}\ \texttt{in}\ x$, the labeling ends successful with $(\texttt{letrec}\ x = \texttt{True}^{\mathsf{sub}}\ \texttt{in}\ x^{\mathsf{vis}})^{\mathsf{vis}}$. However, for both expressions no normal order reduction is applicable (since the labels do not match the reduction rules). The former expression is irreducible (it is a stuck expression), since it has a "dynamic type error". The latter expression is irreducible since it is successfully evaluated, i.e. it is a weak head normal form (see Definition 3.8).

It can be verified (by a case analysis) that normal order reduction is unique, i.e. for an expression e either no normal order reduction is possible, or there is a unique expression e' (up to α-equivalence) s.t. $e \xrightarrow{no} e'$.

We sometimes attach more information to the reduction arrow, e.g. $\xrightarrow{no,lbeta}$ denotes a normal order reduction using rule (lbeta), and $+$ and $*$ are used to denote the transitive and reflexive-transitive closure. E.g., $\xrightarrow{no,*}$ denotes the reflexive-transitive closure of $\xrightarrow{no}$.

As before, we write $\xrightarrow{n}$ for exactly n $\to$-steps and we write $\xrightarrow{n\vee m}$ for either n or m steps. The notation $\xrightarrow{a\vee b}$ is also used for a and b being rule names, meaning the union of the rules a and b. For instance, $\xrightarrow{no,lbeta\vee no,lapp,0\vee 1}$ means one or none normal order reduction step using rule (lbeta) or rule (lapp).

We define reduction contexts:

▶ **Definition 3.7.** A reduction context R is any context, such that its hole will be labeled with sub or top by the labeling algorithm in Figure 3.

Clearly, every reduction context also a surface context.

▶ **Definition 3.8.** A *weak head normal form (WHNF)* is a value v, or an expression $(\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ v)$, where v is a value, or an expression $(\texttt{letrec}\ x_1 = (c\ \vec{e}), \{x_i = x_{i-1}\}_{i=2}^{m}, \mathit{Env}\ \texttt{in}\ x_m)$.

As usual for lazy functional programming languages, we consider WHNFs as successfully evaluated programs, and define the notion or convergence accordingly:

▶ **Definition 3.9.** An expression e *converges*, denoted as $e{\Downarrow}$, iff there exists a WHNF e' s.t. $e \xrightarrow{no,*} e'$. We write $e{\Uparrow}$ iff $e{\Downarrow}$ does not hold.

By inspecting the labeling algorithm and the reduction rules one can verify that every WHNF is irreducible w.r.t. normal order reduction.

▶ **Exercise 3.10.** Show that the expression

$$\begin{aligned}
e := \texttt{letrec}\ &\mathit{rep} = \lambda x.(\texttt{Cons}\ x\ \mathit{rep}),\\
&\mathit{head} = \lambda xs.\texttt{case}_{\mathit{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \bot; (\texttt{Cons}\ y\ ys)\ \to\ y\}\\
\texttt{in}\ &\mathit{head}\ (\mathit{rep}\ \texttt{True})
\end{aligned}$$

converges, i.e. reduce the expression e in normal order until a WHNF is reached.

3.2 Contextual Equivalence

As program equivalence we use contextual equivalence which equates two expressions if exchanging one program by the other program cannot be observed in any surrounding program context. This is similar to the definitions in *LNAME*. Due to the quantification over all contexts, it is sufficient to observe convergence.

▶ **Definition 3.11.** Let e_1, e_2 be two *LR*-expressions. We define *contextual equivalence* $\sim_c$ w.r.t the operational semantics of *LR*: Let $e_1 \sim_c e_2$, iff for all contexts $C \in C_{LR}$: $C[e_1]{\Downarrow} \iff C[e_2]{\Downarrow}$. The *contextual preorder* $\leq_c$ is defined as: $e_1 \leq_c e_2$, iff for all contexts $C \in C_{LR}$: $C[e_1]{\Downarrow} \implies C[e_2]{\Downarrow}$. Thus, $e_1 \sim_c e_2$ iff $e_1 \leq_c e_2$ and $e_2 \leq_c e_1$.

Contextual equivalence is a congruence, i.e. it is an equivalence relation which is compatible with contexts.

3.3 The Context Lemma

A helpful tool for proving contextual equivalences is a context lemma which restricts the set of contexts which has to be taken into account for proving contextual equivalence. In contrast to *LNAME*, no closing substitutions are required for the context lemma in *LR*, since reduction contexts in *LR* also include letrec-expressions, which can close expressions (and thus have the same effect as substitutions).

For the proof of the context lemma, we will use *multicontexts*, i.e. expressions with several (or no) holes $\cdot_i$, where every hole occurs exactly once in the term. We write a multicontext as $C[\cdot_1, \ldots, \cdot_n]$, and if the terms e_i for $i = 1, \ldots, n$ are placed into the holes $\cdot_i$, then we denote the resulting term as $C[e_1, \ldots, e_n]$.

▶ **Lemma 3.12.** *Let C be a multicontext with n holes. Then the following holds:*
If there are expressions e_i with $i \in \{1, \ldots, n\}$ such that $C[e_1, \ldots, e_{i-1}, \cdot_i, e_{i+1}, \ldots, e_n]$ is a reduction context, then there exists a hole $\cdot_j$, such that for all expressions $e'_1, \ldots, e'_n$ $C[e'_1, \ldots, e'_{j-1}, \cdot_j, e'_{j+1}, \ldots, e'_n]$ is a reduction context.

Proof. We assume there is a multicontext C with n holes and there are terms $e_1, \ldots, e_n$ with $R_i := C[e_1, \ldots, e_{i-1}, \cdot_i, e_{i+1}, \ldots, e_n]$ being a reduction context. Since R_i is a reduction context, there is an execution of the labeling algorithm starting with $C[e_1, \ldots, e_n]$ which labels e_i. We fix this execution and apply the same execution to $C[\cdot_1, \ldots, \cdot_n]$ and stop when we arrive at a hole. Either the execution stops at hole $\cdot_i$ or earlier at some hole $\cdot_j$. Since the labeling algorithm visits only positions in a reduction context, the claim follows. ◀

Note that the numbers i and j in the previous lemma are not necessarily identical, e.g. for the multicontext $(\texttt{letrec}\ x = [\cdot_1], y = [\cdot_2]\ \texttt{in}\ y)$ and the term $e_2 := x$ the context $(\texttt{letrec}\ x = [\cdot_1], y = e_2\ \texttt{in}\ y)$ is a reduction context. If we replace e_2 by another term e'_2 e.g. $e'_2 := y$ then $(\texttt{letrec}\ x = [\cdot_1], y = e'_2\ \texttt{in}\ y)$ is not a reduction context, but $(\texttt{letrec}\ x = e'_1, y = [\cdot]\ \texttt{in}\ y)$ is a reduction context for any term e'_1.

The following lemma is easy to verify:

▶ **Lemma 3.13.** *Let e_1, e_2 be expressions, ρ be a permutation on variables, then*

$$(\forall \text{ reduction contexts } R : R[e_1]\Downarrow \implies R[e_2]\Downarrow) \implies (\forall \text{ reduction contexts } R : R[\rho(e_1)]\Downarrow \implies R[\rho(e_2)]\Downarrow)$$

We now prove a lemma using multicontexts which is more general than needed, since the context lemma (Lemma 3.15) is a specialization of the claim.

▶ **Lemma 3.14.** *For all $n \geq 0$ and for all multicontexts C with n holes and for all expressions $e_1, ..., e_n$ and $e'_1, ..., e'_n$ it holds:*

If for all $i = 1, \ldots, n$: $\forall$ reduction contexts $R : (R[e_i]\Downarrow \implies R[e'_i]\Downarrow)$, then $C[e_1, \ldots, e_n]\Downarrow \implies C[e'_1, \ldots, e'_n]\Downarrow$.

Proof. The proof is by induction, where n, C, e_i, e'_i for $i = 1, \ldots, n$ are given. The induction is on the measure (l, n), where

- l is the length of the reduction sequence $C[e_1, \ldots, e_n] \xrightarrow{no,l} e$ where e is a WHNF.
- n is the number of holes in C.

We assume that the pairs are ordered lexicographically, thus this measure is well-founded. The claim holds for $n = 0$, i.e., all pairs $(l, 0)$, since if C has no holes there is nothing to show.

Now let $(l, n) > (0, 0)$. For the induction step we assume that the claim holds for all n', C', $\overline{e}_i, \overline{e}'_i$, $i = 1, \ldots, n'$ with $(l', n') < (l, n)$. Let us assume that the precondition holds, i.e., that $\forall i : \forall$ reduction contexts $R : (R[e_i]\Downarrow \implies R[e'_i]\Downarrow)$. Let $(C[e_1, \ldots, e_n]) \xrightarrow{no,l} e$ where e is a WHNF. We distinguish two cases:

- There is some index j, such that $C[e_1, \ldots, e_{j-1}, \cdot_j, e_{j+1}, \ldots, e_n]$ is a reduction context. Lemma 3.12 implies that there is a hole $\cdot_i$ such that $R_1 := C[e_1, \ldots, e_{i-1}, \cdot_i, e_{i+1}, \ldots, e_n]$ and $R_2 := C[e'_1, \ldots, e'_{i-1}, \cdot_i, e'_{i+1}, \ldots, t_n]$ are both reduction contexts.
  Let C_1 be the multicontext: $C_1 := C[\cdot_1, \ldots, \cdot_{i-1}, e_i, \cdot_{i+1}, \ldots, \cdot_n]$. From $C[e_1, \ldots, e_n] = C_1[e_1, \ldots, e_{i-1}, e_{i+1}, \ldots, e_n]$ we have $(C_1[e_1, \ldots, e_{i-1}, e_{i+1}, \ldots, e_n])\Downarrow$. Since C_1 has $n-1$ holes, we can use the induction hypothesis and derive $C_1[e'_1, \ldots, e'_{i-1}, e'_{i+1}, \ldots, e'_n]\Downarrow$, i.e. $C[e'_1, \ldots, e'_{i-1}, e_i, e'_{i+1}, \ldots, e'_n]\Downarrow$. From that we have $R_2[e_i]\Downarrow$. Using the precondition we derive $R_2[e'_i]\Downarrow$, i.e. $C[e'_1, \ldots, e'_n]\Downarrow$.
- There is no index j, such that $C[e_1, \ldots, e_{j-1}, \cdot_j, e_{j+1}, \ldots, e_n]$ is a reduction context. If $l = 0$, then $C[e_1, \ldots, e_n]$ is a WHNF and since no hole is in a reduction context, $C[e'_1, \ldots, e'_n]$ is also a WHNF, hence $C[e'_1, \ldots, e'_n]\Downarrow$.
  If $l > 0$, then let $(C[e_1, \ldots, e_n]) \xrightarrow{no} e' \xrightarrow{no,l-1} e$. The first normal order reduction $C[e_1, \ldots, e_n] \xrightarrow{no} e'$ can also be used for $C[e'_1, \ldots, e'_n]$ where this normal order reduction can modify the context C, the number of occurrences of the expressions e_i and the positions of the expressions e_i. We now argue that the elimination, duplication or variable permutation for every e_i can also be applied to e'_i. More formally, we will show if $C[e_1, \ldots, e_n] \xrightarrow{no,a} C'[r_1, \ldots, r_m]$, then there exists a variable permutation ρ such that for every $i = 1, \ldots, m$, there is some j such that $(r_i, r'_i) = (\rho(e_j), \rho(e'_j))$ and $C[e'_1, \ldots, e'_n] \xrightarrow{no,a} C'[r'_1, \ldots, r'_m]$.
  We go through the cases of the normal order reduction and figure out how the expressions e_i and e'_i are modified by the reduction step, where we only mention the interesting cases.
  - If $\cdot_i$ is in an alternative of a `case`-expression, which is discarded by a (case)-reduction, or $\cdot_i$ is in the argument of a `seq`-expression that is discarded by a (seq)-reduction, then e_i and e'_i are both eliminated.
  - If the normal order reduction is not a (cp)-reduction that copies a superterm of e_i or e'_i, and e_i and e'_i are not eliminated as mentioned in the previous item, then e_i and e'_i can only change their respective position.
  - If the normal order reduction is a (cp)-reduction that copies a superterm of e_i or e'_i, then renamed copies $r_i := \rho_{e,i}(e_i)$ and $r'_i := \rho_{e',i}(e'_i)$ of e_i and e'_i will occur, where $\rho_{e,i}, \rho_{e',i}$ are permutations on variables. W.l.o.g. for all i: $\rho_{e,i} = \rho_{e',i}$. Free variables of e_i or e'_i can also be renamed in r_i, r'_i if they are bound in the copied superterm. But with Lemma 3.13 we have: The precondition also holds for r_i, r'_i, i.e. $\forall$ reduction contexts R: $R[r_i]\Downarrow \implies R[r'_i]\Downarrow$.

  Now we can use the induction hypothesis: Since $C'[r_1, \ldots, r_m]$ has a terminating sequence of normal order reductions of length $l - 1$ we also have $C'[r'_1, \ldots, r'_m]\Downarrow$. With $C[e'_1, \ldots, e'_n] \xrightarrow{no,a} C'[r'_1, \ldots, r'_m]$ we have $C[e'_1, \ldots, e'_n]\Downarrow$. ◀

▶ **Lemma 3.15** (Context Lemma)**.** *Let e_1, e_2 be expressions. If and only if for all reduction contexts R: $(R[e_1]\Downarrow \implies R[e_2]\Downarrow)$, then $\forall C : (C[e_1]\Downarrow \implies C[e_2]\Downarrow)$; i.e. $e_1 \leq_c e_2$.*

Proof. One direction follows from Lemma 3.14 instantiating the multicontext with a single hole context. The other direction is obvious, since every reduction context is also a context. ◀

3.4 Correctness of Reductions and Program Transformations

In Figure 4 additional program transformations are defined. We use the following unions: (gc) is the union of (gc1) and (gc2); (cpx) is the union of (cpx-in) and (cpx-e); (cpcx) is the union of (cpcx-in) and (cpcx-e).

(gc1)	$(\texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n, Env\ \texttt{in}\ e) \to (\texttt{letrec}\ Env\ \texttt{in}\ e)$ if for all $i : x_i$ does not occur in Env nor in e
(gc2)	$(\texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e) \to e$ if for all $i : x_i$ does not occur in e
(cpx-in)	$(\texttt{letrec}\ x = y, Env\ \texttt{in}\ C[x]) \to (\texttt{letrec}\ x = y, Env\ \texttt{in}\ C[y])$ where y is a variable and $x \neq y$
(cpx-e)	$(\texttt{letrec}\ x = y, z = C[x], Env\ \texttt{in}\ e) \to (\texttt{letrec}\ x = y, z = C[y], Env\ \texttt{in}\ e)$ where y is a variable and $x \neq y$
(cpax)	$(\texttt{letrec}\ x = y, Env\ \texttt{in}\ e) \to (\texttt{letrec}\ x = y, Env[y/x]\ \texttt{in}\ e[y/x])$ where y is a variable, $x \neq y$ and $y \in FV(s, Env)$
(cpcx-in)	$(\texttt{letrec}\ x = c\ \overrightarrow{e}, Env\ \texttt{in}\ C[x]) \to (\texttt{letrec}\ x = c\ \overrightarrow{y}, y_1 = e_1, \ldots, y_n = e_{ar(c)}, Env\ \texttt{in}\ C[c\ \overrightarrow{y}])$
(cpcx-e)	$(\texttt{letrec}\ x = c\ \overrightarrow{e}, z = C[x], Env\ \texttt{in}\ e') \to (\texttt{letrec}\ x = c\ \overrightarrow{y}, y_1 = e_1, \ldots, y_{ar(c)} = e_{ar(c)}, z = C[c\ \overrightarrow{y}], Env\ \texttt{in}\ e')$
(abs)	$(\texttt{letrec}\ x = c\ \overrightarrow{e}, Env\ \texttt{in}\ e') \to \texttt{letrec}\ x = c\ \overrightarrow{y}, y_1 = e_1, \ldots, y_{ar(c)} = e_{ar(c)}, Env\ \texttt{in}\ e'$ where $ar(c) \geq 1$
(abse)	$(c\ \overrightarrow{e}) \to (\texttt{letrec}\ x_1 = e_1, \ldots, x_{ar(c)} = e_{ar(c)}\ \texttt{in}\ c\ \overrightarrow{x})$ where $ar(c) \geq 1$
(xch)	$(\texttt{letrec}\ x = e, y = x, Env\ \texttt{in}\ e') \to (\texttt{letrec}\ y = e, x = y, Env\ \texttt{in}\ e')$

Figure 4 Extra transformation rules

We use the unions of the reduction rules also for normal order reduction and thus e.g. write $\xrightarrow{no,llet}$ for $\xrightarrow{no,llet\text{-}in \vee no,llet\text{-}e}$.

We briefly explain the additional transformations: The transformation (gc) performs garbage collection by removing unused `letrec`-environments, and (cpx), (cpax) copy variables, and can be used to shorten chains of indirections. The transformation (cpcx) copies a constructor application into a referenced position, where the arguments are shared by new `letrec`-bindings. Similarly, (abs) and (abse) perform this sharing without copying the constructor application.

Correctness of program transformations in *LR* is defined analogous to the definition in *LNAME*:

▶ **Definition 3.16.** A program transformation P is *correct*, if it preserves contextual equivalence, i.e. $P \subseteq \sim_c$.

For proving correctness of the reduction and transformations, we use the following notations: Non-normal order reduction steps for the language *LR* are called *internal* and denoted by a label i. An internal reduction in a reduction context is marked by iR, and an internal reduction in a surface context by iS.

3.4.1 The Reductions (case-c), (seq-c), (lbeta), (lapp), (lcase), (lseq)

▶ **Lemma 3.17.** *Every $\xrightarrow{R,a}$-reduction in a reduction context where $a \in \{$(case-c), (seq-c), (lbeta), (lapp), (lcase), (lseq)$\}$ is a normal order reduction.*

Proof. This follows by checking the possible term structures in a reduction context. ◀

▶ **Proposition 3.18.** *The transformations (case-c), (seq-c), (lbeta), (lapp), (lcase), (lseq) are correct. I.e., $e_1 \xrightarrow{a} e_2$ with $a \in \{$(case-c), (seq-c), (lbeta), (lapp), (lcase), (lseq)$\}$ implies $e_1 \sim_c e_2$.*

Proof. This follows from the context lemma 3.15: It is sufficient to consider $R[e_1]$ and $R[e_2]$. From $e_1 \xrightarrow{a} e_2$ and Lemma 3.17 it follows that $R[e_1] \xrightarrow{no,a} R[e_2]$.

Since normal order reduction is unique, it follows $R[e_1]{\Downarrow}$ iff $R[e_2]{\Downarrow}$. Now we apply the context lemma. ◀

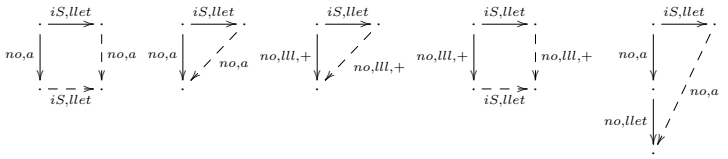
The reductions (lll), (cp), (case-e), (case-in), (seq-e), (seq-in) may be non-normal order in a reduction context, so other arguments are required. We will use the diagram-based proof technique – already introduced for *LNAME* in Section 2.3.3 – for the calculus *LR*. We omit the formal definition of the diagrams and the completeness of sets of diagrams, since they are analogous to *LNAME*, where expressions and normal order reduction of *LNAME* are replaced by expressions and normal order reduction of *LR*.

3.4.2 Correctness of (llet) and (cp)

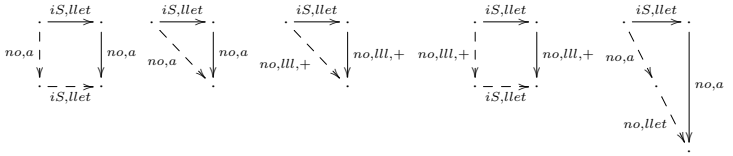
We prove correctness of the reductions (llet) and (cp) by presenting the required forking and commuting diagrams and answer diagrams and then using the TRS-encoding of the diagrams to finish the proof. Note that the diagrams are more complex than the ones shown for *LNAME* in Section 2.3.3. Hand-made inductive proofs (without automation by proving termination of TRSs) and more information on computing the complete sets of diagrams can be found in [37].

For the reduction (llet), we use the reductions in surface contexts instead of reduction contexts for the following reason: they are more general, cover all reduction contexts, and the set of forkings diagrams is simpler.

▶ **Lemma 3.19.** *A complete set of forking diagrams for (iS,llet) is:*



A complete set of commuting diagrams for (iS,llet) is:



Proof. This follows by inspecting all overlappings between an (iS,llet)-transformation and a normal order reduction. ◀

▶ **Lemma 3.20.** *If $e_1 \xrightarrow{iS,llet} e_2$, then e_1 is a WHNF iff e_2 is a WHNF.*

▶ **Proposition 3.21.** *If $e_1 \xrightarrow{llet} e_2$, then $e_1 \sim_c e_2$.*

```
(STRATEGY INNERMOST)
(VAR x k)
(RULES
 iS_llet(no_lll(x)) -> no_lll(iS_llet(x))
 iS_llet(no_llet(x)) -> no_llet(iS_llet(x))
 iS_llet(no_a(x)) -> no_a(iS_llet(x))
 iS_llet(no_lll(x)) -> no_lll(x)
 iS_llet(no_llet(x)) -> no_llet(x)
 iS_llet(no_a(x)) -> no_a(x)
 iS_llet(rem(x)) -> exp(k,x)
 iS_llet(no_lll(x)) -> iS_llet(rem(x))
 iS_llet(rem(no_lll(x))) -> iS_llet(rem(x))
 exp(s(k),x) -> no_lll(exp(k,x))
 exp(s(z),x) -> no_lll(x)
 iS_llet(rem2(x)) -> exp2(k,x)
 iS_llet(no_lll(x)) -> iS_llet(rem2(x))
 iS_llet(rem2(no_lll(x))) -> iS_llet(rem2(x))
 exp2(s(k),x) -> no_lll(exp2(k,x))
 exp2(s(z),x) -> no_lll(iS_llet(x))
 iS_llet(no_lll(no_llet(x))) -> no_lll(x)
 iS_llet(no_llet(no_llet(x))) -> no_llet(x)
 iS_llet(no_a(no_llet(x))) -> no_a(x)
 iS_llet(Answer) -> Answer
)
```

Figure 5 TRS-encoding of the forking and answer diagrams for (llet)

```
(STRATEGY INNERMOST)
(VAR x k)
(RULES
 iS_llet(no_lll(x)) -> no_lll(iS_llet(x))
 iS_llet(no_llet(x)) -> no_llet(iS_llet(x))
 iS_llet(no_a(x)) -> no_a(iS_llet(x))
 iS_llet(no_lll(x)) -> no_lll(x)
 iS_llet(no_llet(x)) -> no_llet(x)
 iS_llet(no_a(x)) -> no_a(x)
 iS_llet(rem(x)) -> exp(k,x)
 iS_llet(no_lll(x)) -> iS_llet(rem(x))
 iS_llet(rem(no_lll(x))) -> iS_llet(rem(x))
 exp(s(k),x) -> no_lll(exp(k,x))
 exp(s(z),x) -> no_lll(x)
 iS_llet(rem2(x)) -> exp2(k,x)
 iS_llet(no_lll(x)) -> iS_llet(rem2(x))
 iS_llet(rem2(no_lll(x))) -> iS_llet(rem2(x))
 exp2(s(k),x) -> no_lll(exp2(k,x))
 exp2(s(z),x) -> no_lll(iS_llet(x))
 iS_llet(no_lll(x)) -> no_lll(no_llet(x))
 iS_llet(no_llet(x)) -> no_llet(no_llet(x))
 iS_llet(no_a(x)) -> no_a(no_llet(x))
 iS_llet(Answer) -> Answer
)
```

Figure 6 TRS-encoding of the commuting and answer diagrams for (llet)

Proof. For $e_1 \leq_c e_2$, we show $S[e_1]{\Downarrow} \implies S[e_2]{\Downarrow}$ for all surface contexts S and then apply the context lemma. If $S[e_1] \xrightarrow{no,llet} S[e_2]$ then the claim holds, since normal order reduction is deterministic. If the reduction is internal, then we use the forking diagrams and the answer diagram (obtained from Lemma 3.20) and show termination of the corresponding term rewrite system shown in Figure 5. Note that for reductions of the form $\xrightarrow{no,a}$ we instantiated the variable a for all reduction rules occurring in the diagrams. Other reduction rules are treated symbolically by the unary function symbol `no_a` without instantiation of a (see [21] for proofs of soundness and completeness of this approach).

An adapted version of AProVE which can handle free variables on the right hand side shows innermost termination and the certifier CeTA [7, 40] certifies the proof (see [30] for the mechanized proofs).

For proving $e_2 \leq_c e_1$ by the context lemma it suffices to show $S[e_2]{\Downarrow} \implies S[e_1]{\Downarrow}$ for all surface contexts S. Again only the case where the transformation is internal is non-trivial. An encoding of the commuting diagrams and the answer diagram as TRS is shown in Figure 6. This TRS is innermost terminating (which is proved by AProVE and certified by CeTA, see [30] for the generated proofs). ◀

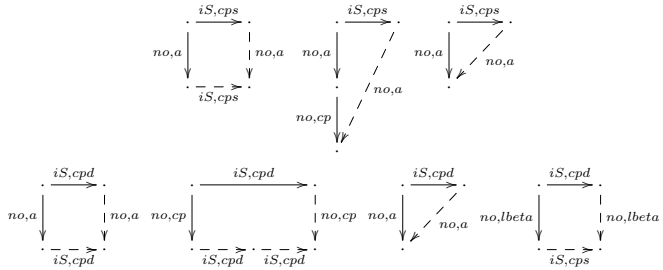
To show that the (cp)-reduction is correct we use the following approach[8]: We distinguish whether the position of the replaced variable is below an abstraction or not. Thus we define two programs transformations:

(cps) := (cp) where the position of the replaced variable is in a surface context.
(cpd) := (cp) where the position of the replaced variable is not in a surface context.

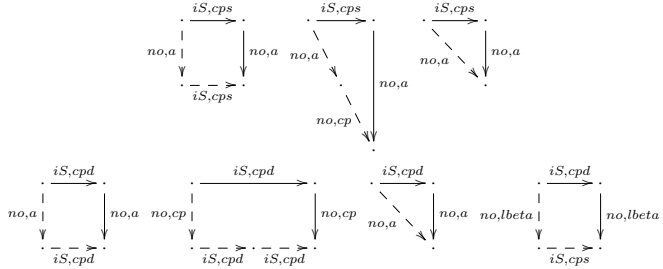
By checking all overlappings of a transformation step and a normal order reduction the following lemma is obtained:

[8] Which is required, since the direct approach does not work (see Remark 3.25)

▶ **Lemma 3.22.** *Complete sets of forking diagrams for* $\xrightarrow{iS,cps}$ *and for* $\xrightarrow{iS,cpd}$ *are:*



Complete sets of commuting diagrams for $\xrightarrow{iS,cps}$ *and for* $\xrightarrow{iS,cpd}$ *are:*



▶ **Lemma 3.23.** *If* $e_1 \xrightarrow{iS,cp} e_2$, *then* e_1 *is a WHNF iff* e_2 *is a WHNF.*

▶ **Proposition 3.24.** *If* $e_1 \xrightarrow{cp} e_2$, *then* $e_1 \sim_c e_2$.

Proof. Again the context lemma allows to restrict to applications of the transformation inside surface contexts. The cases where the transformation is not internal are trivial due to determinism of the normal order reduction. For the other cases we prove innermost termination of the corresponding term rewrite systems which are obtained from the forking and the commuting diagrams and the answer diagrams (see Figures 7 and 8). For both term rewrite systems AProVE shows innermost termination and CeTA certifies the proof. ◀

▶ **Remark 3.25.** For the commuting diagrams the proof does not work if we do not distinguish between (iS,cps) and (is,cpd) transformations. The corresponding TRS is

```
(STRATEGY INNERMOST)
(VAR x)
(RULES
 iS_cp(no_a(x)) -> no_a(iS_cp(x))
 iS_cp(no_cp(x)) -> no_cp(iS_cp(x))
 iS_cp(no_lbeta(x)) -> no_lbeta(iS_cp(x))
 iS_cp(no_a(x)) -> no_a(no_cp(x))
 iS_cp(no_cp(x)) -> no_cp(no_cp(x))
 iS_cp(no_lbeta(x)) -> no_lbeta(no_cp(x))
 iS_cp(no_a(x)) -> no_a(x)
 iS_cp(no_cp(x)) -> no_cp(x)
 iS_cp(no_lbeta(x)) -> no_lbeta(x)
 iS_cp(no_cp(x)) -> no_cp(iS_cp(iS_cp(x)))
 iS_cp(Answer) -> Answer
)
```

```
(STRATEGY INNERMOST)
(VAR x)
(RULES
 iS_cps(no_a(x)) -> no_a(iS_cps(x))
 iS_cps(no_cp(x)) -> no_cp(iS_cps(x))
 iS_cps(no_lbeta(x)) -> no_lbeta(iS_cps(x))
 iS_cps(no_a(no_cp(x))) -> no_a(x)
 iS_cps(no_cp(no_cp(x))) -> no_cp(x)
 iS_cps(no_lbeta(no_cp(x))) -> no_lbeta(x)
 iS_cps(no_a(x)) -> no_a(x)
 iS_cps(no_cp(x)) -> no_cp(x)
 iS_cps(no_lbeta(x)) -> no_lbeta(x)
 iS_cpd(no_a(x)) -> no_a(iS_cpd(x))
 iS_cpd(no_cp(x)) -> no_cp(iS_cpd(x))
 iS_cpd(no_lbeta(x)) -> no_lbeta(iS_cpd(x))
 iS_cpd(no_cp(x)) -> no_cp(iS_cpd(iS_cpd(x)))
 iS_cpd(no_a(x)) -> no_a(x)
 iS_cpd(no_cp(x)) -> no_cp(x)
 iS_cpd(no_lbeta(x)) -> no_lbeta(x)
 iS_cpd(no_lbeta(x)) -> no_lbeta(iS_cps(x))
 iS_cps(Answer) -> Answer
 iS_cpd(Answer) -> Answer
)
```

Figure 7 TRS-encoding of the forking and answer diagrams for (cpd) and (cps)

```
(STRATEGY INNERMOST)
(VAR x)
(RULES
 iS_cps(no_a(x)) -> no_a(iS_cps(x))
 iS_cps(no_cp(x)) -> no_cp(iS_cps(x))
 iS_cps(no_lbeta(x)) -> no_lbeta(iS_cps(x))
 iS_cps(no_a(x)) -> no_a(no_cp(x))
 iS_cps(no_cp(x)) -> no_cp(no_cp(x))
 iS_cps(no_lbeta(x)) -> no_lbeta(no_cp(x))
 iS_cps(no_a(x)) -> no_a(x)
 iS_cps(no_cp(x)) -> no_cp(x)
 iS_cps(no_lbeta(x)) -> no_lbeta(x)
 iS_cpd(no_a(x)) -> no_a(iS_cpd(x))
 iS_cpd(no_cp(x)) -> no_cp(iS_cpd(x))
 iS_cpd(no_lbeta(x)) -> no_lbeta(iS_cpd(x))
 iS_cpd(no_cp(x)) -> no_cp(iS_cpd(iS_cpd(x)))
 iS_cpd(no_a(x)) -> no_a(x)
 iS_cpd(no_cp(x)) -> no_cp(x)
 iS_cpd(no_lbeta(x)) -> no_lbeta(x)
 iS_cpd(no_lbeta(x)) -> no_lbeta(iS_cps(x))
 iS_cps(Answer) -> Answer
 iS_cpd(Answer) -> Answer
)
```

Figure 8 TRS-encoding of the commuting and answer diagrams for (cpd) and (cps)

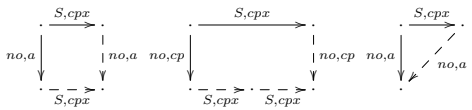
Indeed this TRS is nonterminating (which can also be verified by AProvE). A looping rewrite sequence begins as follows:

```
iS_cp(no_cp(no_cp(Answer)))
→ no_cp(iS_cp(iS_cp(no_cp(Answer))))
→ no_cp(iS_cp(no_cp(no_cp(Answer))))
→ ...
```

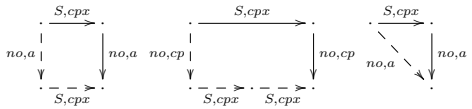
3.4.3 Correctness of Extra Transformations and (case) and (seq)

▶ **Exercise 3.26.** Consider the transformation (cpx).

A complete set of forking diagrams for $\xrightarrow{S,cpx}$ is



A complete set of commuting diagrams for $\xrightarrow{S,cpx}$ is



The following property holds for the transformation (*cpx*) which can be used to derive answer diagrams for (S, cpx): $e \xrightarrow{S,cpx} e'$, then e is a WHNF iff e' is a WHNF.

Show that (cpx) is a correct program transformation in *LR* by encoding the diagrams as term rewrite system and proving innermost termination by AProVE.

Correctness of (cpx) is proved in Exercise 3.26, we omit the correctness proofs of other transformations (they can be found in [37]):

▶ **Proposition 3.27** ([37]). *The transformations (gc), (abs), (abse), (cpx), (cpax), (xch) and (cpcx) are correct.*

Correctness of the remaining reduction rules for (seq) and (case) can be proved without diagrams, and instead using correctness of the extra transformations:

▶ **Proposition 3.28.** *The reductions (case-in) and (case-e) are correct program transformations.*

Proof. Proposition 3.18 shows that (case-c) is a correct program transformation. From Proposition 3.27 we obtain that (cpcx) and (cpx) are correct program transformations. We show by induction that a (case-e) and (case-in)-reduction is correct by using the correctness of the transformations (cpcx), (case-c) and (cpx). The induction is on the length of the variable chain in the reduction (case-in) (or (case-e), respectively). We give the proof only for (case-in), the other is a copy of this proof. For the base case the (case-in) reduction can also be performed by the sequence of reductions: $\xrightarrow{cpcx} \cdot \xrightarrow{case-c}$

$$\begin{array}{ll} & (\texttt{letrec}\ x = c\ \overrightarrow{e}, Env\ \texttt{in}\ C[\texttt{case}_T\ x\ (c\ \overrightarrow{z} \to e')\ alts]) \\ \xrightarrow{cpcx} & (\texttt{letrec}\ x = c\ \overrightarrow{y}, \{y_i = e_i\}_{i=1}^n, Env\ \texttt{in}\ C[\texttt{case}_T\ (c\ \overrightarrow{y})\ (c\ \overrightarrow{z} \to e')\ alts]) \\ \xrightarrow{case-c} & (\texttt{letrec}\ x = c\ \overrightarrow{y}, \{y_i = e_i\}_{i=1}^n, Env\ \texttt{in}\ C[(\texttt{letrec}\ \{z_i = y_i\}_{i=1}^n\ \texttt{in}\ e')]) \end{array}$$

For the induction we replace a (case-in) reduction operating on a chain $\{x_i = x_{i-1}\}_{i=2}^m$ with the sequence $\xrightarrow{cpcx} \cdot \xrightarrow{case-in} \cdot \xrightarrow{cpx,n} \cdot \xleftarrow{cpx,n} \cdot \xleftarrow{cpcx}$, where n is the arity of the constructor and the (case-in) reduction operates on the chain $\{x_i = x_{i-1}\}_{i=3}^m$:

$$\begin{array}{ll} & (\texttt{letrec}\ x_1 = c\ \overrightarrow{e}, \{x_i = x_{i-1}\}_{i=2}^m, Env\ \texttt{in}\ C[\texttt{case}_T\ x_m\ (c\ \overrightarrow{z} \to e')\ alts]) \\ \xrightarrow{cpcx} & \texttt{letrec}\ x_1 = c\ \overrightarrow{y}, \{y_i = e_i\}_{i=1}^n, x_2 = c\ \overrightarrow{y}, \{x_i = x_{i-1}\}_{i=3}^m, Env \\ & \texttt{in}\ C[\texttt{case}_T\ x_1\ (c\ \overrightarrow{z} \to e')\ alts] \\ \xrightarrow{case-in} & \texttt{letrec}\ x_1 = c\ \overrightarrow{y}, \{y_i = e_i\}_{i=1}^n, x_2 = c\ \overrightarrow{y'}, \{y'_i = y_i\}_{i=1}^n, \{x_i = x_{i-1}\}_{i=3}^m, Env \\ & \texttt{in}\ C[(\texttt{letrec}\ \{z_i = y'_i\}_{i=1}^n\ \texttt{in}\ e')] \\ \xrightarrow{cpx,n} & \texttt{letrec}\ x_1 = c\ \overrightarrow{y}, \{y_i = e_i\}_{i=1}^n, x_2 = c\ \overrightarrow{y'}, \{y'_i = y_i\}_{i=1}^n, \{x_i = x_{i-1}\}_{i=3}^m, Env \\ & \texttt{in}\ C[(\texttt{letrec}\ \{z_i = y_i\}_{i=1}^n\ \texttt{in}\ e')] \\ \xleftarrow{cpx,n} & \texttt{letrec}\ x_1 = c\ \overrightarrow{y'}, \{y_i = e_i\}_{i=1}^n, x_2 = c\ \overrightarrow{y'}, \{y'_i = y_i\}_{i=1}^n, \{x_i = x_{i-1}\}_{i=3}^m, Env \\ & \texttt{in}\ C[(\texttt{letrec}\ \{z_i = y_i\}_{i=1}^n\ \texttt{in}\ e')] \\ \xleftarrow{cpcx} & \texttt{letrec}\ x_1 = c\ \overrightarrow{y}, \{y_i = e_i\}_{i=1}^n, \{x_i = x_{i-1}\}_{i=2}^m, Env \\ & \texttt{in}\ C[(\texttt{letrec}\ \{z_i = y_i\}_{i=1}^n\ \texttt{in}\ e')] \end{array}$$

◀

▶ **Proposition 3.29.** *The reduction (case) is a correct program transformation.*

Proof. Follows from Proposition 3.28 and 3.18. ◀

▶ **Proposition 3.30.** *If $e_1 \xrightarrow{seq} e_2$, then $e_1 \sim_c e_2$.*

Proof. Let e_1, e_2 be expressions with $e_1 \xrightarrow{C,seq} e_2$. If the (seq)-reduction is a (seq-c) reduction, then the claim follows from Proposition 3.18. Otherwise, we can transform e_1 into e_2 as follows: $e_1 \xrightarrow{C,cpcx} \xrightarrow{C,seq} \xleftarrow{C,abs} e_2$. Now the claim follows from Proposition 3.18 and 3.27. ◀

4 Some Brief Notes on Non-Determinism and Concurrency

In this section we briefly discuss program transformations and their correctness and the notion of contextual equivalence in a concurrent or non-deterministic setting. We will not discuss concurrent variations of our core languages which can be used as e.g. a model of Concurrent Haskell [18]. Such models and their semantic treatment can for example be found in [28, 29], but they require several syntactic extensions e.g. for Haskell's monadic input and output. Instead we consider a minimal extension of LR (called $LR_{\texttt{choice}}$) by non-determinism: We add a binary operator `choice` which can freely choose between both of its arguments. Such a language clearly does not model concurrency (e.g. there are no threads and no synchronization between them), but several problems and design decisions for the semantics which occur in concurrent languages already occur in our minimalistic extension by non-determinism. A practical application of the core language $LR_{\texttt{choice}}$ may be functional-logical programming languages like Curry [12, 11].

4.1 Syntax and Operational Semantics of $LR_{\texttt{choice}}$

We briefly introduce the syntax and normal order reduction of $LR_{\texttt{choice}}$ where we do not give the complete definitions, but only describe the changes w.r.t. the calculus LR (details on such a calculus can e.g. be found in [16, 38, 32])

▶ **Definition 4.1.** Expressions of $LR_{\texttt{choice}}$ are the expressions of LR where additionally expressions of the form

$$\texttt{choice}\ e_1\ e_2$$

are permitted at any expression position (where e_1, e_2 are $LR_{\texttt{choice}}$-expressions).

The reduction rules of the calculus $LR_{\texttt{choice}}$ are the rules shown in Figure 2 and the following two additional rules for evaluating the `choice`-operator:

$$\begin{array}{ll}\text{(choice-l)} & C[(\texttt{choice}\ e_1\ e_2)^{\mathsf{sub}}] \to C[e_1] \\ \text{(choice-r)} & C[(\texttt{choice}\ e_1\ e_2)^{\mathsf{sub}}] \to C[e_2]\end{array}$$

The labeling algorithm to search the normal order redex in $LR_{\texttt{choice}}$ is exactly the algorithm from Definition 3.4 applied to the extended syntax of $LR_{\texttt{choice}}$. There are no new label shifting rules for the `choice`-operator, since the labeling stops when arriving at a subexpression ($\texttt{choice}\ e_1\ e_2$).

▶ **Definition 4.2** (Normal Order Reduction of $LR_{\texttt{choice}}$)**.** Let e be an $LR_{\texttt{choice}}$-expression. Then a single *normal order reduction step* $e \xrightarrow{no} e'$ is defined by first applying the labeling algorithm to e, and if the labeling algorithm terminates successfully, then one of the rules in Figure 2 or (choice-l) or (choice-r) has to be applied, if possible, where the labels $\mathsf{sub}, \mathsf{vis}$ must match the labels in the expression e (e may have more labels), and e' is the result after erasing all labels.

Clearly, normal order reduction in $LR_{\texttt{choice}}$ is *not* unique, since e.g. the expression $\texttt{letrec}\ x = (\texttt{choice True False})\ \texttt{in}\ x$ has two possible normal order reductions:

- $\texttt{letrec}\ x = (\texttt{choice True False})\ \texttt{in}\ x \xrightarrow{no,choice\text{-}l} \texttt{letrec}\ x = \texttt{True}\ \texttt{in}\ x$
- $\texttt{letrec}\ x = (\texttt{choice True False})\ \texttt{in}\ x \xrightarrow{no,choice\text{-}r} \texttt{letrec}\ x = \texttt{False}\ \texttt{in}\ x$

However, the "redex" of the normal order reduction is unique, i.e. the to-be-replaced subexpression is uniquely determined. This can easily be shown by inspecting the rules of the labeling algorithm.

The operator `choice` implements so-called erratic non-determinism, which means that the evaluation can freely choose between e_1 and e_2 in $\texttt{choice}\ e_1\ e_2$ independent of the operational behavior of e_1 and e_2. Other forms of non-determinism are demonic choice, where first e_1 and e_2 must be evaluated to values before the non-determinism can choose between the values, and angelic choice[9], where e_i must be chosen whenever e_j diverges (for $(i,j) \in \{(1,2),(2,1)\}$). More details on the classification of non-determinism can be found in [39, 41].

The successful outcomes of normal order reduction in $LR_{\texttt{choice}}$ are weak head normal forms which are defined as in LR:

▶ **Definition 4.3.** A *weak head normal form* (WHNF) in $LR_{\texttt{choice}}$ is a value (i.e. a constructor application or an abstraction), or an expression of the form $\texttt{letrec}\ Env\ \texttt{in}\ v$, where v is a value, or an expression of the form $\texttt{letrec}\ x_1 = c\ \overrightarrow{e}, x_2 = x_1, \ldots, x_m = x_{m-1}, Env\ \texttt{in}\ x_m$.

Considering convergence in $LR_{\texttt{choice}}$, the introduced non-determinism of reduction induces different notions of termination: On the one hand we may analyze whether it is *possible* to evaluate an expression to a WHNF, on the other hand we may analyze whether *all* evaluations end in a WHNF. The former behavior is called *may-convergence*, while the latter occurs in two forms in the literature: as *must-convergence* and as *should*-convergence. We introduce all three notions in the following definition:

▶ **Definition 4.4.** Let e be an $LR_{\texttt{choice}}$-expression. We say that e *may-converges* (denoted as $e{\downarrow}$) iff there exists a WHNF e', s.t. $e \xrightarrow{no,*} e'$. We say that e *should-converges* (denoted as $e{\Downarrow}$) iff in every evaluation of e may-convergence is never lost, i.e. $e{\Downarrow}$ iff $\forall e' : e \xrightarrow{no,*} e' \implies e'{\downarrow}$. We write $e \xrightarrow{no,\omega}$ if e has an infinite normal order reduction, i.e. there exist e_i for all $i \in \mathbb{N}$ s.t. $e_i \xrightarrow{no} e_{i+1}$ and $e \xrightarrow{no} e_1$. We say that expression e *must-converges* (denoted as $e{\downarrow\downarrow}$) iff $e{\Downarrow}$ and $\neg(e \xrightarrow{no,\omega})$ hold.

Note that in deterministic calculi like LR all the three notions of may-, should-, and must-convergence coincide.

4.2 Contextual Equivalence and Program Transformations

Contextual equivalence in non-deterministic program calculi follows the basic principle that programs (or expressions) are identified as equivalent iff their behavior cannot be distinguished if the programs are used in any context. However, it is not sufficient to look for may-convergence only. The notion we will work with is the combined test of may- and should-convergence. However, we also define a contextual preorder based on the must-convergence test to compare it to our approach and to show the differences.

▶ **Definition 4.5** (Contextual Equivalence in $LR_{\texttt{choice}}$)**.** Let e_1, e_2 be $LR_{\texttt{choice}}$-expressions. The may-, should- and must-contextual preorders are defined as follows:

- Contextual may-preorder: $e_1 \leq_{c,\downarrow} e_2$ iff $\forall C : C[e_1]{\downarrow} \implies C[e_2]{\downarrow}$
- Contextual should-preorder: $e_1 \leq_{c,\Downarrow} e_2$ iff $\forall C : C[e_1]{\Downarrow} \implies C[e_2]{\Downarrow}$
- Contextual must-preorder: $e_1 \leq_{c,\downarrow\downarrow} e_2$ iff $\forall C : C[e_1]{\downarrow\downarrow} \implies C[e_2]{\downarrow\downarrow}$

[9] In [27, 25] such a non-deterministic operator in a call-by-need setting was analyzed w.r.t. contextual equivalence

We also use $\sim_{c,\downarrow}$, $\sim_{c,\Downarrow}$, $\sim_{c,\downarrow\downarrow}$ for the symmetrizations of the preorders.

The *contextual preorder* $\leq_c$ in $LR_{\texttt{choice}}$ is defined as $e_1 \leq_c e_2$ iff $e_1 \leq_{c,\downarrow} e_2 \wedge e_1 \leq_{c,\Downarrow} e_2$. *Contextual equivalence* $\sim_c$ in $LR_{\texttt{choice}}$ is defined as $e_1 \sim_c e_2$ iff $e_1 \leq_c e_2 \wedge e_2 \leq_c e_1$.

The following example shows that observing may-convergence only is not sufficient to distinguish obviously different programs:

▶ **Example 4.6.** Let $\bot$ be a diverging expression, e.g. $\bot := \texttt{letrec}\ x = x\ \texttt{in}\ x$. Then one can show that $\texttt{choice True}\ \bot \sim_{c,\downarrow} \texttt{True}$ holds. However, $\texttt{choice True}\ \bot \not\leq_{c,\Downarrow} \texttt{True}$ and also $\texttt{choice True}\ \bot \not\leq_{c,\downarrow\downarrow} \texttt{True}$, since the empty context $[\cdot]$ distinguishes both expressions w.r.t. should- and must-convergence.

The next example shows that observing should-convergence (or also must-convergence) only is also not sufficient:

▶ **Example 4.7.** In $LR_{\texttt{choice}}$ the following equivalences hold: $\texttt{choice}\ \bot\ \texttt{True} \sim_{c,\Downarrow} \bot$ and also $\texttt{choice}\ \bot\ \texttt{True} \sim_{c,\downarrow\downarrow} \bot$, while $\bot \not\leq_{c,\downarrow} \texttt{choice}\ \bot\ \texttt{True}$.

Both examples show that a combination of may-convergence and either should- or must-convergence should be used in the definition of contextual equivalence to distinguish obviously different expressions. A difference between observing should-convergence or observing must-convergence shows the following example:

▶ **Example 4.8.** Let $e := \texttt{letrec}\ x = \lambda y.(\texttt{choice True}\ (x\ \texttt{True}))\ \texttt{in}\ (x\ \texttt{True})$. Then $e\Downarrow$, but $\neg(e\downarrow\downarrow)$. Indeed, the following (in-)equivalences hold:

- $e \sim_{c,\downarrow} \texttt{True}$
- $e \sim_{c,\Downarrow} \texttt{True}$
- $e \not\sim_{c,\downarrow\downarrow} \texttt{True}$
- $e \sim_{c,\downarrow} \texttt{choice}\ \bot\ \texttt{True}$
- $e \not\sim_{c,\Downarrow} \texttt{choice}\ \bot\ \texttt{True}$
- $e \sim_{c,\downarrow\downarrow} \texttt{choice}\ \bot\ \texttt{True}$

We use the combination of may- and should-convergence, since the expression e in the previous example can be identified with a busy wait that finally has to choose `True` in the end as result, and thus it should not be distinguished from `True`.

A further advantage of should-convergence compared to must-convergence is that the predicate construction of should-convergence is similar to the construction of modal operators in modal logic and that the corresponding contextual equivalence is closed w.r.t. to further constructions which does not hold for must-convergence (see [33]).

Finally, if fairness of scheduling and reduction comes into play in concurrent languages, should-convergence has the advantage, that reasoning (e.g. about program equivalences) does not require to treat the fairness explicitly (see e.g. [24, 27]).

▶ **Exercise 4.9.** Which of the following expressions are may-convergent, should-convergent, or must-convergent?

1. `True`
2. $\lambda x.(\texttt{letrec}\ y = y\ \texttt{in}\ y)$
3. `choice True False`
4. $\texttt{letrec}\ x = \texttt{choice}\ x\ y, y = \texttt{Nil}\ \texttt{in}\ x$
5. $\texttt{letrec}\ x = \texttt{choice}\ x\ y, y = \texttt{Nil}\ \texttt{in}\ y$
6. $\texttt{letrec}\ x = \lambda z.(\texttt{choice}\ (x\ \texttt{Nil})\ y), y = \texttt{Nil}\ \texttt{in}\ (x\ \texttt{Nil})$

4.2.1 Reasoning on Should-Convergence

A further more technical advantage of should-convergence compared to must-convergence is that our reasoning tools (the context lemma and the diagram-based proof method) are adaptable.

▶ **Definition 4.10.** Let *may-divergence* be the negation of should-convergence, i.e. an $LR_{\texttt{choice}}$-expression e *may-diverges* (written as $e\uparrow$) if $e\Downarrow$ does not hold. Let *must-divergence* be the negation of may-convergence, i.e. an $LR_{\texttt{choice}}$-expression e *must-diverges* (written as $e\Uparrow$) if $e\downarrow$ does not hold.

The following lemma shows that may-divergence can be inductively (i.e. by considering the existence of a finite reduction sequence) characterized:

▶ **Lemma 4.11.** *For all $LR_{\texttt{choice}}$-expressions e: $e\uparrow$ iff $\exists e' : e \xrightarrow{no,*} e' \wedge e'\Uparrow$.*

Note that this definition looks very similar to the definition of may-convergence: only the property that e' is a WHNF is replaced by the property that $e'\Uparrow$. Hence, most of our techniques (the proof of the context lemma and the diagram based technique, which in its core uses an induction on reduction sequences of finite length) can be applied in the same manner, where only the proof-part for the base case changes. Let $\xrightarrow{P}$ be a program transformation. The base case in our proofs for showing correctness w.r.t. may-convergence is of the form, that whenever $e \xrightarrow{P} e'$ and e is a WHNF then also e' is a WHNF (or at least $e'\downarrow$), and vice versa. For reasoning on the may-divergence part, this base case becomes: whenever $e \xrightarrow{P} e'$ and $e\Uparrow$ then $e'\Uparrow$ (or at least $e'\uparrow$). Indeed in most of the cases we get this part for free from the reasoning about may-convergence. Assume that $P \subseteq \leq_{c,\downarrow}$ has been shown. Then $C[e]\downarrow \implies C[e']\downarrow$ for all e, e' with $e \xrightarrow{P} e'$. However this implication is equivalent to $\neg(C[e']\downarrow) \implies \neg(C[e]\downarrow)$ which is the same as $C[e']\Uparrow \implies C[e]\Uparrow$ and exactly describes the desired base case in induction proofs that show $e'\uparrow \implies e\uparrow$.

We will not further extend this part on non-determinism, but we hopefully made clear that our reasoning techniques also work for the combination of may- and should-convergence. The calculus $LR_{\texttt{choice}}$ was analyzed in [38] w.r.t. correctness of program transformations. The results are, that a context lemma holds in $LR_{\texttt{choice}}$ and that all reduction rules are correct program transformations except for the two choice-reductions (*choice-l*) and (*choice-r*). This result is expected, since the choice-reductions are the real source of non-determinism.

References

1 Abramsky, S.: The lazy lambda calculus. In Turner, D.A., ed.: Research Topics in Functional Programming, Addison-Wesley (1990) 65–116
2 AProVE: Homepage of AProVE (2015) `http://aprove.informatik.rwth-aachen.de`.
3 Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, New York, NY, USA (1998)
4 Barendregt, H.P.: The Lambda Calculus. Its Syntax and Semantics. North-Holland, Amsterdam, New York (1984)
5 Bird, R.: Introduction to Functional Programming using Haskell. Prentice-Hall, London (1998)
6 Bird, R.: Thinking functionally with Haskell. Cambridge University Press, Cambridge, UK (2014)
7 CeTA: Homepage of CeTA (2015) `http://cl-informatik.uibk.ac.at/software/ceta/`.

8 Fuhs, C., Giesl, J., Plücker, M., Schneider-Kamp, P., Falke, S.: Proving termination of integer term rewriting. In Treinen, R., ed.: Proc. 20th International Conference on Rewriting Techniques and Applications (RTA 2009). Volume 5595 of Lecture Notes in Comput. Sci., Springer (2009) 32–47

9 Giesl, J., Brockschmidt, M., Emmes, F., Frohn, F., Fuhs, C., Otto, C., Plücker, M., Schneider-Kamp, P., Ströder, T., Swiderski, S., Thiemann, R.: Proving termination of programs automatically with AProVE. In Demri, S., Kapur, D., Weidenbach, C., eds.: Proc. 7th International Joint Conference on Automated Reasoning (IJCAR'14). Volume 8562 of Lecture Notes in Comput. Sci., Springer (2014) 184–191

10 Goldberg, M.: A variadic extension of curry's fixed-point combinator. Higher Order Symbol. Comput. **18**(3-4) (2005) 371–388

11 Hanus, M.: Functional logic programming: From theory to Curry. In: Programming Logics - Essays in Memory of Harald Ganzinger. Volume 7797 of Lecture Notes in Comput. Sci., Springer (2013) 123–168

12 Hanus (ed.), M.: Curry: An integrated functional logic language (vers. 0.8.3). Available at `http://www.curry-language.org` (2012)

13 Haskell-community: The Haskell Programming Language (2015) `http://www.haskell.org`.

14 Moran, A.K.D., Sands, D.: Improvement in a lazy context: An operational theory for call-by-need. In: Proc. 26th ACM Principles of Programming Languages (POPL 1999), ACM Press (1999) 43–56

15 Moran, A.: Call-by-name, Call-by-need, and McCarthy's Amb. PhD thesis, Department of Computing Science, Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden (1998)

16 Moran, A.K.D., Sands, D., Carlsson, M.: Erratic fudgets: A semantic theory for an embedded coordination language. Sci. Comput. Program. **46**(1-2) (2003) 99–135

17 Morris, J.: Lambda-Calculus Models of Programming Languages. PhD thesis, MIT (1968)

18 Peyton Jones, S., Gordon, A., Finne, S.: Concurrent Haskell. In H.-J. Boehm, G.L.S.J., ed.: Proc. 23th ACM Principles of Programming Languages (POPL 1996), ACM (1996) 295–308

19 Peyton Jones, S.L.: The Implementation of Functional Programming Languages. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1987)

20 Plotkin, G.D.: Call-by-name, call-by-value, and the lambda-calculus. Theoret. Comput. Sci. **1** (1975) 125–159

21 Rau, C., Sabel, D., Schmidt-Schauß, M.: Correctness of program transformations as a termination problem. In Gramlich, B., Miller, D., Sattler, U., eds.: IJCAR. Volume 7364 of Lecture Notes in Comput. Sci., Springer, Heidelberg (2012) 462–476

22 Rau, C., Schmidt-Schauß, M.: Computing overlappings by unification in the deterministic lambda calculus LR with letrec, case, constructors, seq and variable chains. Frank report 46, Institut für Informatik. Fachbereich Informatik und Mathematik. Goethe-Universität Frankfurt am Main (2011) `http://www.ki.informatik.uni-frankfurt.de/papers/frank/`.

23 Rau, C., Schmidt-Schauß, M.: A unification algorithm to compute overlaps in a call-by-need lambda-calculus with variable-binding chains. In: Proc. 25th International Workshop on Unification (UNIF 2011). (2011) 35–41

24 Rensink, A., Vogler, W.: Fair testing. Inform. and Comput. **205**(2) (2007) 125–198

25 Sabel, D.: Semantics of a Call-by-Need Lambda Calculus with McCarthy's amb for Program Equivalence. Dissertation, J. W. Goethe-Universität Frankfurt, Institut für Informatik. Fachbereich Informatik und Mathematik (2008)

26 Sabel, D.: Structural rewriting in the pi-calculus. In Schmidt-Schauß, M., Sakai, M., Sabel, D., Chiba, Y., eds.: Proc. First International Workshop on Rewriting Techniques for Program Transformations and Evaluation (WPTE 2014). Volume 40 of OASICS., Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2014) 51–62

27 Sabel, D., Schmidt-Schauß, M.: A call-by-need lambda-calculus with locally bottom-avoiding choice: Context lemma and correctness of transformations. Math. Structures Comput. Sci. **18**(03) (2008) 501–553

28 Sabel, D., Schmidt-Schauß, M.: A contextual semantics for Concurrent Haskell with futures. In Schneider-Kamp, P., Hanus, M., eds.: Proc. 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming (PPDP 2011), ACM (2011) 101–112

29 Sabel, D., Schmidt-Schauß, M.: Conservative concurrency in Haskell. In Dershowitz, N., ed.: Proc. 27th IEEE Symposium on Logic in Computer Science (LICS 2012), IEEE (2012) 561–570

30 Sabel, D., Schmidt-Schauß, M.: Website to the course on "Rewriting Techniques for Correctness of Program Transformations" held on the International School on Rewriting (ISR 2015) (2015) `http://www.ki.cs.uni-frankfurt.de/events/isr15`.

31 Schmidt-Schauß, M., Machkasova, E., Sabel, D.: Extending Abramsky's lazy lambda calculus: (non)-conservativity of embeddings. In van Raamsdonk, F., ed.: Proc. 24th International Conference on Rewriting Techniques and Applications (RTA 2013). Volume 21 of LIPIcs., Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2013) 239–254

32 Schmidt-Schauß, M., Machkasova, E.: A finite simulation method in a non-deterministic call-by-need calculus with letrec, constructors and case. In: Proc. 19th International Conference on Rewriting Techniques and Applications (RTA 2008). Volume 5117 of Lecture Notes in Comput. Sci., Springer-Verlag (2008) 321–335

33 Schmidt-Schauß, M., Sabel, D.: Closures of may-, should- and must-convergences for contextual equivalence. Inform. Process. Lett. **110**(6) (2010) 232 – 235

34 Schmidt-Schauß, M., Sabel, D.: On generic context lemmas for higher-order calculi with sharing. Theoret. Comput. Sci. **411**(11-13) (2010) 1521 – 1541

35 Schmidt-Schauß, M., Sabel, D.: Improvements in a functional core language with call-by-need operational semantics. In Falaschi, M., Albert, E., eds.: Proc. 17th International Symposium on Principles and Practice of Declarative Programming (PPDP 2015), ACM (2015) 220–231

36 Schmidt-Schauß, M., Sabel, D., Machkasova, E.: Simulation in the call-by-need lambda-calculus with letrec, case, constructors, and seq. Logical Methods in Computer Science **11**(1) (2015)

37 Schmidt-Schauß, M., Sabel, D., Schütz, M.: Safety of Nöcker's strictness analysis. J. Funct. Programming **18**(04) (2008) 503–551

38 Schmidt-Schauß, M., Schütz, M., Sabel, D.: On the safety of Nöcker's strictness analysis. Frank report 19, Institut für Informatik. Fachbereich Informatik und Mathematik. Goethe-Universität Frankfurt am Main (2004) `http://www.ki.informatik.uni-frankfurt.de/papers/frank/`.

39 Søndergaard, H., Sestoft, P.: Nondeterminism in functional languages. Computer Journal **35**(5) (1992) 514–523

40 Thiemann, R., Sternagel, C.: Certification of termination proofs using CeTA. In: Proc. 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009). Volume 5674 of Lecture Notes in Comput. Sci., Springer (2009) 452–468

41 Walicki, M., Meldal, S.: Algebraic approaches to nondeterminism – an overview. ACM Computing Surveys **29**(1) (1997) 30–81