Reasoning about Contextual Equivalence: From Untyped to Polymorphically Typed Calculi

David Sabel, Manfred Schmidt-Schauß, and Frederik Harwath
Goethe-University Frankfurt am Main

{sabel,schauss,harwath}@informatik.uni-frankfurt.de

Abstract: This paper describes a syntactical method for contextual equivalence in polymorphically typed lambda-calculi. Our specific calculus has letrec as cyclic let, data constructors, case-expressions, seq, and recursive types. The typed language is a subset of the untyped language. Normal-order reduction is defined for the untyped language. Since there are less typed contexts the typed contextual preorder and equivalence are coarser than the untyped ones. We use type-labels for all subexpressions of the typed expressions, and prove a context lemma for the type-labeled calculus. We show how to reason about correctness of program transformations in the typed language, and how to easily transfer the methods and results from untyped program calculi to polymorphically typed ones.

1 Introduction

A successful approach for the semantics of programming languages is Morris' style contextual equivalence which is based on the syntax and the operational interpretation of programs. Contextual equivalence is used for a wide variety of programming calculi, including lambda-calculi, deterministic and non-deterministic constructs, lazy as well as strict functional programming languages, languages with mutable storage, and process calculi.

The use of parametric polymorphic types in programming languages is popular and used in several modern programming languages, where among the advantages are that the type system is rather expressive and that static type-checking (Hindley-Milner type-checking) is possible and is efficient in all practical cases.

There are investigations in typed program calculi, mainly for call-by-value languages: for simply-typed PCF [Gor99], for a monomorphically typed fragment of ML and a nondeterministic extension of it [Las98], a simply-typed calculus [Pit02], an F2-polymorphic calculus [Pit00] and also [SP07, LL08]. An interesting discussion on parametric polymorphic calculi, extensions by `seq`, operational semantics and relations to programming languages is in [VJ07].

The calculi in [Pit00, VJ07, SP07] change the termination behavior w.r.t. their untyped variants, since they use F-type polymorphism where in general the convergence is not invariant under removal of types, e.g. $t = (\texttt{letrec}\ x = x\ \texttt{in}\ x)$ is an untyped and diverging expression, whereas the type-F-polymorphic term $\Lambda\alpha.(\texttt{letrec}\ x :: \alpha = x ::$

α `in` $x :: \alpha$) is converging. This observation is also made in [JV09] which analyses the semantics of a call-by-name polymorphic lambda calculus in Haskell style, but without a cyclic let. Several papers e.g. [Pit00, JV09] use logical relations to define semantic equivalences. Using syntactical methods to describe equivalences of expressions appears to lead to the same equivalences for deterministic languages. Note that since our language allows seq and letrec, as is usual in functional core-languages, results from pure parametric polymorphic lambda calculi cannot be used to justify equalities (see e.g. [VJ07]).

In [SSSS08], we used contextual equivalence for an untyped deterministic call-by-need language that can be seen as a core language for Haskell to show safety of a strictness optimization, and also correctness of a lot of program transformations. Investigations for untyped non-deterministic call-by-need program calculi are [MSC99, SSS08, Sab08] where correctness is shown for large sets of program transformations. Those investigations have in common that they use *untyped* or very weakly typed calculi. There are several justifications for this approach: one is that adding types would make the syntactic analysis of reductions far too complex, the other is that lots of interesting program transformations can already be shown in the untyped case, and finally the authors' (sometimes implicit) claim that the results can be transferred to the typed case. However, those papers give no hints on the exact connection between typed and untyped calculi and also did not mention how to prove correctness of program transformations that hold in the typed calculi, but not in the untyped ones.

The overall goal of this paper is to bridge this gap by showing that the relation between typed and untyped equivalences is as trivial as claimed for polymorphically typed program calculi, and also to transfer our reasoning method to typed calculi. In particular this is done for call-by-need lambda-calculi that permit data constructors, case, and a cyclic let. Our approach is designed to meet the following requirements: (i) The equivalences for the untyped calculus also hold in the typed calculus. (ii) The syntactic proof methods developed and used for the untyped calculi are also applicable to the typed calculus after adapting them to types. (iii) Applicability of typed program transformations can be decided locally.

As already mentioned, the formulation of the polymorphic extensions makes (i) trivial. In order to achieve (ii), polymorphic types are added as labels to subexpressions. Moreover, the normal-order reduction can be described also within the typed setting by accompanying every reduction rule with a corresponding operation on the type label. Though this is not necessary for convergence, it greatly helps reasoning, since then we can show that reduction and/or transformations do not lead to non-well-typed expressions, which is heavily used in induction proofs. Similarly, requirement (iii) can be shown to hold for our program transformations exploiting the type labeling for defining the transformations and for their usually inductive correctness proofs.

Our main results are: a modelling of a (predicative) polymorphically typed call-by-need calculus with cyclic let, case and constructors, a context lemma for a polymorphically typed calculus, and a demonstration that the syntactic diagram methods based on induction proofs can be made to work also in the polymorphically typed setting.

Outline. First we define the untyped and typed language with type labels. Well-typed expressions are subsequently determined by consistency rules for the type-labeling. Then

we introduce the small-step reduction semantics which operates on untyped expressions, i.e. for typed expressions on the type-erasure. After defining contextual semantics we lift (proper restrictions of) known untyped program equivalences into the typed setting. The typed small-step operational semantics on typed expressions is defined and shown to be equal to untyped one. We show a context lemma for the typed calculus. After introducing the method of forking and commuting diagrams, we apply it to an example and derive a type dependent program transformation that preserves contextual equivalence.

2 Syntax of the Polymorphic Typed Lambda Calculus

We describe the polymorphically typed language L_{PLC} which employs cyclic sharing using a letrec [AK97] and is like a Haskell-core-language.

Syntax of Expressions, Untyped First we describe the syntax of the untyped language L_{LC}. We assume that there are type-constructors K given with their respective arity, denoted $ar(K)$, similar as Haskell-style data- and type constructors (see [Pey03]). We assume that the constant type constructors `Bool` and the unary `List` are already defined. For every type-constructor K, there is a set $D_K \neq \emptyset$ of data constructors, such that $K_1 \neq K_2 \implies D_{K_1} \cap D_{K_2} = \emptyset$. Every (data) constructor comes with a fixed arity. We assume that $D_{\texttt{Bool}} = \{\texttt{True}, \texttt{False}\}$, where these constructors are 0-ary, and that $D_{\texttt{List}} = \{\texttt{Nil}, \texttt{Cons}\}$, where `Nil` is 0-ary and `Cons` is 2-ary.

The syntax of L_{LC}-expressions is as follows, where V is a nonterminal generating variables, E means expressions, c, c_i are data constructors, and Alt is a `case`-alternative:

$$\begin{aligned} E \quad &::= V \mid (E\ E) \mid \lambda V.E \mid (\texttt{seq}\ E\ E) \mid (\texttt{letrec}\ V_1 = E_1, \ldots, V_n = E_n\ \texttt{in}\ E) \\ &\quad \mid\ (c_i\ E_1 \ldots E_{ar(c_i)}) \mid (\texttt{case}_K\ E\ \texttt{of}\ Alt_1 \ldots Alt_{|D_K|}) \\ Alt_i &::= ((c_i\ V_1 \ldots V_{ar(c_i)})\ \texttt{->}\ E) \end{aligned}$$

Note that data constructors can only be used with all their arguments present: partial applications are disallowed. We assume that there is a $\texttt{case}_K$ for every type constructor K, which is the only place where types are visible in L_{LC}. The $\texttt{case}_K$-construct is assumed to have a case-alternative $((c_i\ x_1 \ldots x_{ar(c_i)})\ \texttt{->}\ e)$ for every constructor $c_i \in D_K$, where the variables in a pattern have to be distinct. The scoping rules in expressions are as usual, where `letrec` behaves as cyclic let, and hence the scope of x_i in $(\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ t)$ is the terms $s_1, \ldots, s_n$ and t. We use $FV(t)$ to denote the set of free variables in t. The sequence of the bindings in the `letrec`-environment may be interchanged. We assume that expressions satisfy the distinct variable convention before reduction is applied, which can be achieved by a renaming of bound variables. Additionally we require the notion of *contexts* $\mathbb{C}$, which are like expressions over a syntax with the additional atomic expression $[\cdot]$, the *hole*, where a context must contain exactly one occurrence of the hole. We will also need the class of *surface-context*, which are contexts where the hole is not in an abstraction.

Syntax of Types (Quantifier-free) Types T in the polymorphic extended lambda-calculus have the following syntax $T ::= X \mid (T_1 \to T_2) \mid (K\ T_1 \ldots T_{ar(K)})$, where the symbols

X, X_i are type variables, T, T_i are types, and K is a type constructor. We use the usual conventions for bracketing of function types, i.e. $T_1 \to T_2 \to T_3$ means $T_1 \to (T_2 \to T_3)$. We also allow *(universal) types* with the following syntax: $\forall X_1, \ldots, X_n.T$, where X_i are type variables and T is a quantifier-free type. If we speak of types we mean non-quantified types and if we speak of universal types, we assume that non-quantified types are included. Note that the variables stand for quantifier-free types (i.e. we have predicative polymorphism). The sequence of variables in the quantifier does not play any role, so we may also use $\forall \mathcal{X}.T$, where $\mathcal{X}$ is a set of type variables, and T a quantifier-free type. The set of free type variables in a universal type T is denoted as $FTV(T)$. For instance, $FTV(\forall a, b.(a \to c) \to d \to a) = \{c, d\}$.

The Type-Labeled Language. The language L_{LLC} consists of L_{LC}-expressions where type labels are added to subexpressions, patterns and constructors, and where quantified types are only permitted at x in `letrec`-bindings "$x = t$". We use $s :: T$ as notation for the labels. For L_{LLC}-expressions t, the type-erasure is denoted as $\varepsilon(t) \in L_{LC}$.

The positions of x in the `letrec`-binding "$x = t$" is called *let-position*. We assume that for every universal type T there is an infinite set V_T of variables of this type. If $x \in V_T$, then T is called the *built-in* type of the variable x. We define $bt(x) = T$ iff $x \in V_T$. At its binding position the variable must be labeled with its built-in type. L_{LLC}*-Contexts* C are typed, where the the hole is also labeled with a type, written $\mathbb{C}[\cdot :: T]$, where T may only be a non-quantified type. An example for the type labels is $\lambda x.x$ which can be labeled as $(\lambda x :: a.x :: a) :: (a \to a)$. Accordingly, an example of a L_{LLC}-context is $(\lambda x :: a.[\cdot :: a]) :: (a \to a)$.

Types of Data Constructors Let K be a type constructor with constructors D_K. Then the type of every constructor $c_{K,i} \in D_K$ must be of the form

$$\forall X_1, \ldots, X_{ar(K)}.T_{K,i,1} \to \ldots \to T_{K,i,m_i} \to K\ X_1 \ldots X_{ar(K)},$$

where $m_i = ar(c_{K,i})$, $X_1, \ldots, X_{ar(K)}$ are distinct type variables, and only X_i occur as free type variables in $T_{K,i,1}, \ldots, T_{K,i,m_i}$. The function *typeOf* will be used to give the type of data constructors.

A *(type-)substitution* ρ substitutes types for type variables. Let $\mathrm{Dom}(\rho) := \{X \mid \rho(X) \neq X\}$, $\mathrm{Cod}(\rho) = \{\rho(X) \mid X \in \mathrm{Dom}(\rho)\}$ and $\mathrm{VCod}(\rho) := \bigcup_{X \in \mathrm{Dom}(\rho)} FTV(\rho(X))$. With $\gamma(T)$ we denote the *semantics of universal types* defined as $\gamma(\forall \mathcal{X}.T) = \{\sigma(T) | \mathrm{Dom}(\sigma) \subseteq \mathcal{X}\}$. We say T_1 is an *instance* of a type T_2, denoted as $T_1 \preceq T_2$, iff $\gamma(T_1) \subseteq \gamma(T_2)$.

Example 2.1. *Let T be the type $\forall a, b.a \to b$. Then* `Bool` $\to$ `Bool` *is an instance of T, as well as $\forall a.a \to$* `Bool`*, where the latter has a variable name in common with T. A slightly more complex case is that $\forall a.$*(`List` a) $\to$ `Bool` $\to c$ *is an instance of $\forall a, b.a \to b \to c$; note that c is a free type variable in this case.*

Example 2.2. *This example shows a type-labeled expression. The type of the composition is* `(.)` $:: \forall a, b, c.(b \to c) \to (a \to b) \to a \to c$*. A type labeling (the types of some variables are not repeated) for the composition may be:*

$$\begin{aligned}(\lambda f :: (b \to c).(\lambda g :: (a \to b).\\ (\lambda x :: a.(f\ (g\ x) :: b) :: c) :: (a \to c)) :: ((a \to b) \to a \to c))\\ :: \forall a, b, c.(b \to c) \to (a \to b) \to a \to c\end{aligned}$$

Application	$(s :: S_1 \to S_2\ t :: S_1)$	$\mapsto S_2$
Constant	$(c :: (S_1 \to \ldots \to S_n \to S)\ s_1 :: S_1 \ldots s_n :: S_n)$	$\mapsto S$
Abstraction	$(\lambda x :: S_1.s :: S_2)$	$\mapsto S_1 \to S_2$
Case	$(\texttt{case}_K\ s :: S\ \texttt{of}\ ((c_{K,1}\ x_{1,1} \ldots x_{1,n_1}) :: S \texttt{->}\ t_i :: T)$	$\mapsto T$
	$\ldots$	
	$((c_{K,m}\ x_{m,1} \ldots x_{m,n_m}) :: S \texttt{->}\ t_m :: T))$	
Letrec	$(\texttt{letrec}\ x_1 :: S_1 = t_1 :: T_1, \ldots, x_n :: S_n = t_n :: T_n\ \texttt{in}\ t :: S)$	$\mapsto S$

Figure 1: Computation of $MonoTp$

3 Type Constraints in the Polymorphic Lambda Calculus L_{PLC}

A standard derivation system for polymorphic types of L_{LC} can be found e.g. in [SSSH09] where the types as labels of expressions could be computed for well-typed expressions. Instead of this derivational approach, we will describe a set of conditions on the type labels and conditions among the type labels that must hold for well-typed expressions. This labels-plus-conditions-approach is a bit more complex than the derivational one, but there will be a pay-off if it comes to applications of program transformations and to syntactical reasoning on reduction sequences since reasoning can be localized to the subexpressions.

We use the *distinct type variable assumption*: bound type variables are all distinct and also distinct from all free type-variables. The latter can be achieved by a renaming of bound type variables as well as bound variables whose type contains one of the type variables to be renamed.

In Figure 1 a (one-step) derivation method for the type of an expression, based on the type-labels of subexpressions, is given. In case of a successful derivation of type T for expressions t, we write $MonoTp(t) = T$. This is like a monomorphic type derivation, but for types with occurrences of type variables, which are treated like constants during the derivation.

Type-Constraints: Now we define all the type constraints:

1. For every type-label S of every occurrence of a variable $x \in V_T$, $S \preceq T$ must hold.
2. Lambda-bound variables and variables in case-patterns must have a quantifier-free (but not necessarily ground) type which must be their built-in type.
3. Let-bound variables have their built-in type as type-label at the let-positions.
4. For every occurrence $c :: S$ of a constructor, the constraint $S \preceq typeOf(c)$ must hold; similarly for `seq` with built-in type $\forall a, b.a \to b \to b$.
5. For compound expressions, i.e. the expression is not a variable, not a constructor and not `seq`, the constraint is that the type-label must be the type derived by $MonoTp$ (see Figure 1).
6. In $(\texttt{letrec}\ x_1 :: S_1 = t_1 :: T_1, \ldots, x_n :: S_n = t_n :: T_n\ \texttt{in}\ t)$, for $i = 1, \ldots, n$, the constraints $S_i \preceq T_i'$, $T_i' = \forall \mathcal{X}.T_i$ with $MonoTp(t_i) = T_i$ and

$\mathcal{X} = FTV(T_i) \setminus \left(\bigcup_{x \in FV(t_i)} FTV(bt(x))\right)$ must be satisfied.
(This means S_i must be an instance of T_i after applying the type derivation rule "Generalize").

Definition 3.1. *We also assume that the* distinct variable convention for type variables *holds, which means that the quantifiable type variables in every letrec-binding are different from all other used variables. This can be achieved by a series of type variable renamings, which rename the quantifiable type variables in an expressions t in a letrec-binding $x = t$ with fresh variable names, where perhaps also fresh expression variables have to be introduced.*

Definition 3.2. *If a L_{LLC}-expression $t :: T$ satisfies all the type constraints above, then the expression $t :: T$ is* well-typed. *The language L_{PLC} consists exactly of the well-typed L_{LLC}-expressions.*

Example 3.3. *The expression* $(\texttt{letrec}\ id = \lambda x.x\ \texttt{in}\ (id\ \texttt{True}, id\ \texttt{Nil}))$ *with type-labels is well-typed, where the types are given below:* $id :: \forall a.a \to a$. *The two occurrences of id are differently type labeled as* $\texttt{Bool} \to \texttt{Bool}$ *and* $(\texttt{List Bool}) \to (\texttt{List Bool})$. *A variant of this example is* $(\texttt{letrec}\ id = \lambda x.x, y = id\ \texttt{in}\ (id\ \texttt{True}, y\ \texttt{Nil}))$, *where* $y :: \forall b.(\texttt{List}\ b) \to (\texttt{List}\ b)$ *at the let-position and* $y :: (\texttt{List}\ c) \to (\texttt{List}\ c)$ *at the other occurrence.*

4 Small-Step Operational Semantics of L_{PLC}

The operational semantics of L_{PLC} is defined on the type erasure $\varepsilon(t)$ of typed L_{PLC}-expressions t. This corresponds to usual compilers for functional programming languages, where at run-time type information of expressions is not available. Nevertheless, later we show that the reduction can be turned into a typed one, which allows us to prove correctness of (typed) program transformations using the syntactic method of complete sets of forking diagrams and commuting diagrams.

A reduction step consists of two operations: first finding a normal-order redex, then applying a reduction rule. We define the search by using a labeling algorithm which uses three atomic labels $\mathsf{sub}, \mathsf{top}, \mathsf{vis}$, where top means the top-expression, sub means a subterm reduction, and vis means visited. For an expression s the shifting algorithm starts with s^{top}, where s has no further labels $\mathsf{sub}, \mathsf{top}, \mathsf{vis}$. Then the rules of Figure 2 are applied exhaustively. The shifting algorithm fails, if a loop is detected, which happens if a to-be-labeled position is already labeled vis, and otherwise, if no more rules are applicable, it succeeds. If we apply the labeling algorithm to contexts, then the contexts where the hole will be labeled with sub, top or vis are called the *reduction contexts*. We denote reduction contexts with $\mathbb{R}$.

Normal-order reduction rules are defined in Figure 3, where we assume that a non-failing execution of the labeling algorithm was used before. Otherwise no normal-order reduction is applicable. In Figure 3, a *cv-expression* is an expression of the form $(c\ x_1 \ldots x_n)$

$$
\begin{array}{ll}
(s\ t)^{\mathsf{sub}\vee\mathsf{top}} & \to (s^{\mathsf{sub}}\ t)^{\mathsf{vis}} \\
(\mathtt{letrec}\ Env\ \mathtt{in}\ t)^{\mathsf{top}} & \to (\mathtt{letrec}\ Env\ \mathtt{in}\ t^{\mathsf{sub}})^{\mathsf{vis}} \\
(\mathtt{letrec}\ x = s, Env\ \mathtt{in}\ \mathbb{C}[x^{\mathsf{sub}}]) & \to (\mathtt{letrec}\ x = s^{\mathsf{sub}}, Env\ \mathtt{in}\ \mathbb{C}[x^{\mathsf{vis}}]) \\
(\mathtt{letrec}\ x = s, y = \mathbb{C}[x^{\mathsf{sub}}], Env\ \mathtt{in}\ r) & \to (\mathtt{letrec}\ x = s^{\mathsf{sub}}, y = \mathbb{C}[x^{\mathsf{vis}}], Env\ \mathtt{in}\ r), \\
 & \qquad\qquad \text{if } \mathbb{C}[x] \neq x \\
(\mathtt{seq}\ s\ t)^{\mathsf{sub}\vee\mathsf{top}} & \to (\mathtt{seq}\ s^{\mathsf{sub}}\ t)^{\mathsf{vis}} \\
(\mathtt{case}\ s\ \mathtt{of}\ alts)^{\mathsf{sub}\vee\mathsf{top}} & \to (\mathtt{case}\ s^{\mathsf{sub}}\ \mathtt{of}\ alts)^{\mathsf{vis}}
\end{array}
$$

$\mathsf{sub} \vee \mathsf{top}$ means label sub or top.

Figure 2: Searching the normal-order redex using labels

where c is a constructor and x_i are variables. A *value* is an abstraction or a constructor-expression $(c\ t_1 \ldots t_n)$. A *weak head normal form* (WHNF) is a value v or an expression $(\mathtt{letrec}\ Env\ \mathtt{in}\ v)$.

We illustrate normal-order reduction by evaluating $\varepsilon(((\lambda x.\lambda y.y)\ \mathtt{True}\ \mathtt{False}) :: \mathtt{Bool})$:

$$
\begin{array}{ll}
 & (((\lambda x.\lambda y.y)^{\mathsf{sub}}\ \mathtt{True})^{\mathsf{vis}}\ \mathtt{False})^{\mathsf{vis}} \\
\xrightarrow{\text{lbeta}} & (\mathtt{letrec}\ x = \mathtt{True}\ \mathtt{in}\ ((\lambda y.y)^{\mathsf{sub}}\ \mathtt{False})^{\mathsf{vis}})^{\mathsf{vis}} \\
\xrightarrow{\text{lbeta}} & (\mathtt{letrec}\ x = \mathtt{True}\ \mathtt{in}\ (\mathtt{letrec}\ y = \mathtt{False}\ \mathtt{in}\ y)^{\mathsf{sub}})^{\mathsf{vis}} \\
\xrightarrow{\text{llet-in}} & (\mathtt{letrec}\ x = \mathtt{True}, y = \mathtt{False}^{\mathsf{sub}}\ \mathtt{in}\ y^{\mathsf{vis}})^{\mathsf{vis}} \\
\xrightarrow{\text{cp-in}} & (\mathtt{letrec}\ x = \mathtt{True}, y = \mathtt{False}\ \mathtt{in}\ \mathtt{False})
\end{array}
$$

4.1 Contextual Equivalence

In this section we define Morris-style contextual equivalence for typed expressions. We introduce convergence as the observable behavior of expressions. Expressions are contextually equivalent if their convergence behavior is indistinguishable in all program contexts.

Definition 4.1. *Let $t \in L_{LC}$. A normal order reduction sequence of t is called a* (normal-order) evaluation *if the last term is a WHNF. We write $t \downarrow_{no}$ (t converges) iff there is an evaluation starting from t. Otherwise, if there is no evaluation of t, we write $t\Uparrow_{no}$.*

Definition 4.2 (Contextual Preorder and Equivalence)**.** *Let T be a type and let $s :: T, t :: T$ be L_{PLC}-expressions. We say s and t are* type-invariant *(denoted with $s \approx_{wt} t$) iff for all $\mathbb{C}[\cdot :: T]$: $\mathbb{C}[s] \in L_{PLC} \iff \mathbb{C}[t] \in L_{PLC}$. Contextual equivalence $\sim_T$ is defined as follows:*

$$
\begin{array}{lll}
s \leq_T t & \textit{iff} & s \approx_{wt} t \wedge\ \forall \mathbb{C}[\cdot :: T] : \mathbb{C}[s] \in L_{PLC}\ \Longrightarrow\ (\varepsilon(\mathbb{C}[s])\downarrow_{no} \Rightarrow \varepsilon(\mathbb{C}[t])\downarrow_{no}) \\
s \sim_T t & \textit{iff} & s \leq_T t \wedge t \leq_T s
\end{array}
$$

A relation P on L_{PLC} is *wt-compatible* iff $P \subseteq \approx_{wt}$ and for all T and for all $(s,t) \in P$ of type T: for all $\mathbb{C}[\cdot :: T] : \mathbb{C}[s] \in L_{PLC}\ \Longrightarrow\ \mathbb{C}[s]\ P\ \mathbb{C}[t]$. It is straightforward to show that $\leq_T$ is a precongruence, and that $\sim_T$ is a congruence, where a precongruence is a wt-compatible partial order, and a congruence is a wt-compatible equivalence relation.

(lbeta) $\mathbb{C}[((\lambda x.s)^{\mathsf{sub}}\ r)] \to \mathbb{C}[(\texttt{letrec}\ x = r\ \texttt{in}\ s)]$

(cp-in) $(\texttt{letrec}\ x = v^{\mathsf{sub}}, Env\ \texttt{in}\ \mathbb{C}[x^{\mathsf{vis}}]) \to (\texttt{letrec}\ x = v, Env\ \texttt{in}\ \mathbb{C}[v])$
where v is an abstraction, a variable or a cv-expression

(cp-e) $(\texttt{letrec}\ x = v^{\mathsf{sub}}, y = \mathbb{C}[x^{\mathsf{vis}}], Env\ \texttt{in}\ r)$
$\to (\texttt{letrec}\ x = v, y = \mathbb{C}[v], Env\ \texttt{in}\ r)$
where v is an abstraction, a variable or a cv-expression

(abs) $(\texttt{letrec}\ x = (c\ t_1 \dots t_n)^{\mathsf{sub}}, Env\ \texttt{in}\ r) \to$
$\texttt{letrec}\ x = (\texttt{letrec}\ x_1 = t_1, \dots, x_n = t_n\ \texttt{in}\ (c\ x_1 \dots x_n)), Env\ \texttt{in}\ r$
if $(c\ t_1 \dots t_n)$ is not a cv-expression, where x_i are fresh let-variables

(case) $\mathbb{C}[(\texttt{case}\ (c\ t_1 \dots t_n)^{\mathsf{sub}}\ \texttt{of}\ \dots ((c\ y_1 \dots y_n)\ \texttt{->}\ s) \dots)]$
$\to \mathbb{C}[(\texttt{letrec}\ y_1 = t_1, \dots, y_n = t_n\ \texttt{in}\ s)]$

(case) $\mathbb{C}[(\texttt{case}\ c^{\mathsf{sub}}\ \texttt{of}\ \dots (c\ \texttt{->}\ s) \dots)] \to \mathbb{C}[s]$

(seq) $\mathbb{C}[(\texttt{seq}\ v^{\mathsf{sub}}\ t)] \to \mathbb{C}[t]$ if v is a value

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

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

(lapp) $\mathbb{C}[((\texttt{letrec}\ Env\ \texttt{in}\ s)^{\mathsf{sub}}\ t)] \to \mathbb{C}[((\texttt{letrec}\ Env\ \texttt{in}\ (s\ t))]$

(lseq) $\mathbb{C}[(\texttt{seq}\ (\texttt{letrec}\ Env\ \texttt{in}\ s)^{\mathsf{sub}}\ t)] \to \mathbb{C}[(\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{seq}\ s\ t))]$

(lcase) $\mathbb{C}[(\texttt{case}\ (\texttt{letrec}\ Env\ \texttt{in}\ t)^{\mathsf{sub}}\ \texttt{of}\ alts)]$
$\to \mathbb{C}[(\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{case}\ t\ \texttt{of}\ alts))]$

Figure 3: Normal-order rules

We will also use contextual equivalence $\sim$ for the fully untyped calculus which has L_{LC} as expressions and normal order reduction as operational semantics. The definition of $\sim$ is completely analogous to $\sim_T$ with the only difference that there are no typing conditions on contexts and expressions.

A *typed program transformation* P is a binary relation on L_{PLC}-expressions, such that $s\ P\ t$ is only valid for s, t of equal type. The restriction of P to a type T is denoted P_T. A program transformation P is called *correct* iff for all T and all $(s, t) \in P_T$, the relation $s \sim_T t$ holds. Analogously, for untyped programs, a program transformation is a binary relation on untyped expressions and it is correct if it is a subset of $\sim$.

Disproving the correctness of a (typed or untyped) program transformation is easy, since a counter example consisting of a program context which distinguishes two related expressions by their convergence behavior is sufficient. On the other hand *proving* correctness of program transformations is in general a hard task, since all contexts need to be taken into account. In e.g. [MSC99, SSSS08, SSS08, SSM08] methods to prove correctness of program transformations for untyped letrec calculi were developed. As a first step we will use the result of [SSSS08] to lift untyped program equivalences into the typed calculus. The calculus introduced in [SSSS08] uses the same syntax as L_{LC}, the normal order reduction of this calculus is slightly different, but it is easy to show that these differences do not change the convergence behavior of untyped expressions (a proof of this coincidence for an

(gc) $(\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ t) \to t$ if no x_i occurs free in t

(gc) $(\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n, y_1 = t_1, \ldots, y_m = t_m\ \texttt{in}\ t)$
$\to (\texttt{letrec}\ y_1 = t_1, \ldots, y_m = t_m\ \texttt{in}\ t)$
if no x_i occurs free in t nor in any t_j

(gcp) $(\texttt{letrec}\ x = s, Env\ \texttt{in}\ \mathbb{C}[x]) \to (\texttt{letrec}\ x = s, Env\ \texttt{in}\ \mathbb{C}[s])$

(gcp) $(\texttt{letrec}\ x = s, y = \mathbb{C}[x], Env\ \texttt{in}\ t) \to (\texttt{letrec}\ x = s, y = \mathbb{C}[s], Env\ \texttt{in}\ t)$

(gcp) $(\texttt{letrec}\ x = \mathbb{C}[x], Env\ \texttt{in}\ t) \to (\texttt{letrec}\ x = \mathbb{C}[\mathbb{C}[x]], Env\ \texttt{in}\ t)$

Figure 4: Further program transformations

extended calculus can be found in [SSSH09]). This implies that all reduction rules of Figure 3 and the optimizations *garbage collection* (gc) and *general copying* (gcp) (see Figure 4) are correct (untyped) program transformations for L_{LC} (for (gcp) see [SS07a, SS07b]). This in turn implies:

Theorem 4.3. *For typed L_{PLC}-expressions s, t and contexts C: If $\varepsilon(s) \to \varepsilon(t)$ by a rule of Figure 3 or Figure 4 and $\mathbb{C}[s]$, $\mathbb{C}[t]$ are well-typed and of equal type, and $\mathbb{C}[s] \approx_{wt} \mathbb{C}[t]$, then $\mathbb{C}[s] \to \mathbb{C}[t]$ is a correct typed program transformation for L_{PLC}.*

A sufficient condition for wt-invariance is FV-closedness: We say a program transformation P is *FV-closed* iff for all $(s, t) \in P$ the equality $FV(s) = FV(t)$ holds.

Lemma 4.4. *Let $s, t : T$ be well-typed expressions, such that $FV(s) = FV(t)$. Then $s \approx_{wt} t$.*

Thus, Theorem 4.3 holds in particular for FV-closed relations. The perhaps non-FV-closed relations which modify the occurrences of free variable are (case), (seq) and (gc). Our result then implies that $s \xrightarrow{(case)\vee(seq)\vee(gc)} t$ is correct, provided $FV(s) = FV(t)$. Note also that the formulation of the typed program transformations in the theorem does not provide a scheme for how to obtain for a well-typed expression s a transformed expression t which remains equally typed. I.e., for automated program transformation during the compilation process further criteria are necessary. We will give those criteria (also for other reasons) in the subsequent section.

5 Proving Correctness of Type Dependent Program Transformations

So far we only showed correctness of transformations which also hold in the untyped setting. Now we will develop techniques for proving correctness of transformations which exploit the type information (and which may be wrong for untyped expressions).

A helpful tool to prove program equivalence is a so-called context lemma, which restricts the number of contexts required for proving contextual equivalence. For untyped letrec

calculi several variants of context lemmas exist [MSC99, SSSS08]. In this section we will demonstrate that normal order reduction on typed L_{PLC}-expressions can be made typed by adding appropriate type-labels. Note that this is the reason why we used the type labeling mechanism with consistency rules, instead of introducing type abstractions and applications in the term syntax. As already mentioned in the introduction, for the latter approach the convergence behavior would change when going from untyped to typed reductions. For our approach convergence remains unchanged for reductions with type labels which will enable us to prove a typed context lemma for L_{PLC}. We will also provide a diagram based proof method for proving contextual equivalence, which also requires to keep the typing through the standard reduction.

5.1 Reduction on Typed Expressions

Typed normal-order reduction $\xrightarrow{tno}$ on L_{PLC}-expressions is the same as the untyped normal-order reduction where in addition we need to specify how the type labels are inherited to the resulting expression. This is standard in most cases, the only exception occurs for the rules (cp), which has to be accompanied by an instantiation of types during reduction(see below). Moreover, the distinct-variable convention for type variables (see Definition 3.1) must hold after the reduction, which can be achieved by a renaming of type variables. Again, the only case where this may be necessary is after a (cp)-reduction.

The normal-order (cp)-rules have to be accompanied by a local type-instantiation:

Definition 5.1 (Type Instantiation for (cp-in) and (cp-e))**.** *The (cp-in)-rule with type-instantiation is:* $(\texttt{letrec}\ x = v :: T, Env\ \texttt{in}\ \mathbb{C}[x :: S]) \to (\texttt{letrec}\ x := v :: T, Env\ \texttt{in}\ \mathbb{C}[\rho(v) :: S])$*, where* ρ *is the type-substitution with* $\rho(T) = S$*, and where* $\rho(v)$ *means the expression* v*, after application of the instantiation* ρ *to all types also of subexpressions, where perhaps variables are renamed, respectively replaced, by variables of an instance type. The same for the (cp-e)-rule.*

Note that the type constraints guarantee that only quantifiable type variables of T will be instantiated by ρ.

Example 5.2. *This is an example for the (cp)-rules and their effect on types. Let concatMap*[1] *be the standard Haskell function of type* $\forall a, b.(a \to [b]) \to [a] \to [b]$*, and id be the identity function of type* $\forall a.a \to a$*. Consider the expression:*
$\texttt{letrec}$ *concatMap* $= \lambda f, xs.\texttt{case}\ldots, id = \lambda x.x, \ldots\ \texttt{in}$ (*concatMap id*).

Let the subexpression (*concatMap id*) *be typed* $[[c]] \to [c]$*, with the type labels concatMap* $:: ([c] \to [c]) \to [[c]] \to [c]$*, and id* $:: [c] \to [c]$*. An application of the reduction rule (cp) results in:*
$\texttt{letrec}$ *concatMap* $= \lambda f, xs.\texttt{case}\ldots, id = \lambda x.x, \ldots\ \texttt{in}\ (\lambda f', xs'.\texttt{case}\ldots)$ *id.*

where $f' :: [c] \to [c]$ *and* $xs' :: [[c]]$ *are fresh variables. The type of the copied body of concatMap is an instance of the type computed at the binding, namely* $([c] \to [c]) \to$

[1] We denote the list types as in Haskell, i.e. $[T]$ means $(\texttt{List}\ T)$

$[[c]] \to [c]$*, and then (lbeta) will result in*
`letrec` *concatMap* $= \ldots$, *id* $= \lambda x.x, \ldots$ `in letrec` $f' =$ *id* `in` $(\lambda xs'.\texttt{case}\ldots)$
where id $:: [c] \to [c]$*, and* $(\lambda xs'.\texttt{case}\ldots)$ *is labeled with type* $[[c]] \to [c]$.

Note the example also indicates that (lbeta) does not create variables with a quantified type, i.e. lambda-bound variables and their descendents remain monomorphic.
We will show that normal-order reductions keep the type of the expression, which will show that reduction is type-safe and hence reduction of well-typed terms does not lead to a dynamic type error. This holds since the type of the redex does not change.

Inspecting all reduction rules and their corresponding type modifications shows:

Proposition 5.3. *Let* $s :: T$ *be a* L_{PLC}*-expression. Then* $s \xrightarrow{tno} t$ *implies that* $t :: S$ *is also well-typed and that* $\varepsilon(s) \xrightarrow{no} \varepsilon(t)$*. On the other hand, if* $\varepsilon(s) \xrightarrow{no} t$*, then there exists* $t' :: T \in L_{PLC}$ *such that* $s \xrightarrow{tno} t'$ *and* $\varepsilon(t') = t$*.*
Moreover, typed normal-order reduction is unique up to renaming of type variables.

An expression $s \in L_{PLC}$ is a *(typed) WHNF* if $\varepsilon(s)$ is a WHNF. Let $\xrightarrow{tno,*}$ be the reflexive-transitive closure of $\xrightarrow{tno}$. If for $s \in L_{PLC}$, $s \xrightarrow{tno,*} t$ where t is a typed WHNF, then s is called *converging* (denoted with $s \downarrow_{tno}$). The following characterization of typed contextual preorder and equivalence using typed normal order reduction gives us a criterion to prove correctness of typed program transformations.

Theorem 5.4. *Let* $s, t :: T$ *be well-typed expressions. Then* $s \leq_T t$ *iff* $s \approx_{wt} t$ *and* $\forall \mathbb{C}[\cdot :: T] : \mathbb{C}[s] \in L_{PLC} \implies ((\mathbb{C}[s]) \downarrow_{tno} \Rightarrow (\mathbb{C}[t]) \downarrow_{tno})$

5.2 Proof Techniques for Typed Contextual Equality

A first advantage of Theorem 5.4 is that we are able to prove a typed context-lemma, where the proof is an adaptation from [SSSS08].

Definition 5.5. *A program transformation* P *is* ρ*-closed if it is* FV*-closed, and for all* ρ *where* ρ *is a (type-correct) type-instantiation and a variable-substitution such that variables are renamed, respectively replaced, by variables of an instance type, it holds: For all* $(s, t) \in P$*: If* $\rho(s) \in L_{PLC}$*, then* $(\rho(s), \rho(t)) \in P$

A surface context $\mathbb{S}$ is a context where the hole is not inside the body of an abstraction.

Lemma 5.6 ($\mathbb{S}$-Context Lemma)**.** *Let* P *be a* ρ*-closed program transformation, which preserves convergence, i.e. for all* $(s, t) \in P$ *and for all surface contexts* $\mathbb{S}$*:* $\mathbb{S}[s] \in L_{PLC} \implies (\mathbb{S}[s] \downarrow_{tno} \implies \mathbb{S}[t] \downarrow_{tno})$*. Then for all* T*:* $P_T \subseteq \leq_T$ *holds.*

We define forking and commuting diagrams adapted to our needs in later proofs, where $\xrightarrow{P,\mathbb{S}}$ means an application of P in an $\mathbb{S}$-context (of proper type) to a L_{PLC}-expression.

Definition 5.7. *Let P be a ρ-closed program transformation.* (Typed) forking diagrams *and* (typed) commuting diagrams *for P are meta-rewriting rules on typed reduction sequences and are of the form*

forking diagram :

$$\begin{array}{ccc} \cdot & \xrightarrow{P,\mathbb{S}} & \cdot \\ {\scriptstyle tno}\downarrow & & \downarrow{\scriptstyle tno,k'} \\ \cdot & \xrightarrow{rel} & \cdot \end{array}$$

commuting diagram :

$$\begin{array}{ccc} \cdot & \xrightarrow{P,\mathbb{S}} & \cdot \\ {\scriptstyle tno,k'}\downarrow & & \downarrow{\scriptstyle tno,k\leq 1} \\ \cdot & \xrightarrow{rel} & \cdot \end{array}$$

where $k + k' > 0$ for commuting diagrams, and rel is a relation on L_{PLC}-expressions, which may be denoted also in meta-notation using reductions and program transformations.

For a forking situation $s_1 \xleftarrow{tno} s_2 \xrightarrow{P,\mathbb{S}} s_3$ a forking diagram $\xleftarrow{tno} . \xrightarrow{P,\mathbb{S}} \rightsquigarrow \xrightarrow{tno,k'} . \xleftarrow{rel}$ is applicable *if there exists an expression s_4, such that $s_1 \xrightarrow{rel} s_4 \xleftarrow{tno,k'} s_3$ holds. For a commuting situation $s_1 \xrightarrow{P,\mathbb{S}} s_2 \xrightarrow{tno,k} s_3$ with $k \in \{0,1\}$, a commuting diagram $\xrightarrow{P,\mathbb{S}} . \xrightarrow{tno,k\leq 1} \rightsquigarrow \xrightarrow{tno,k'} . \xrightarrow{rel}$ is* applicable *if there exists s_4 such that $s_1 \xrightarrow{tno,k'} s_4 \xrightarrow{rel} s_3$ holds. A* set of forking diagrams *for P is* complete *if for every forking situation $s_1 \xleftarrow{tno} s_2 \xrightarrow{P,\mathbb{S}} s_3$ where $s_1 \neq s_3$ and s_3 is not a WHNF the set contains at least one applicable diagram. A* set of commuting diagrams *for P is* complete *if for every commuting situation $s_1 \xrightarrow{P,\mathbb{S}} s_2 \xrightarrow{tno} s_3$ where $s_1 \xrightarrow{P,\mathbb{S}} s_2$ is not a typed normal order reduction and s_1 is not a WHNF the set contains at least one applicable diagram.*

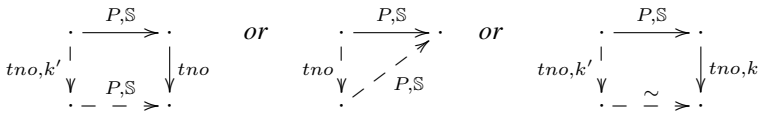
Proposition 5.8. *Let P be a ρ-closed program transformation. If all diagrams of a complete set of forking diagrams for P are of the form*

$$\begin{array}{ccc} \cdot & \xrightarrow{P,\mathbb{S}} & \cdot \\ {\scriptstyle tno}\downarrow & & \downarrow{\scriptstyle tno,k'} \\ \cdot & \xrightarrow{P,\mathbb{S}} & \cdot \end{array} \quad \textit{or} \quad \begin{array}{ccc} \cdot & \xrightarrow{P,\mathbb{S}} & \cdot \\ {\scriptstyle tno}\downarrow & & \downarrow{\scriptstyle tno,k'} \\ \cdot & \xrightarrow{\simeq} & \cdot \end{array}$$

where P preserves WHNFs, i.e. for a WHNF s with $s \xrightarrow{P,\mathbb{S}} t$ the expression t is always a WHNF, then for all T: $P_T \subseteq \leq_T$.

Proof. Let $s \xrightarrow{P} t$ and let $\mathbb{S}$ be a surface context. By induction on the length of a normal order evaluation $\mathbb{S}[s] \xrightarrow{tno,*} r$ (r a WHNF), one can show that $\mathbb{S}[t] \downarrow_{tno}$: The base case is covered, since P preserves WHNFs, hence $\mathbb{S}[t] \downarrow_{tno}$. The induction step has two options for the first reduction of $\mathbb{S}[s] \xrightarrow{tno,*} r$: either $\mathbb{S}[s] \xrightarrow{P,\mathbb{S}} s'$ is already a typed normal order-reduction, then we are finished since normal-order reduction is unique. The other case is that there is an applicable forking diagram for the first reduction of $\mathbb{S}[s] \xrightarrow{tno,*} r$ and the induction hypothesis can be applied. Hence P preserves convergence and thus by the context lemma 5.6 we have $P_T \subseteq \leq_T$ for all T. □

Proposition 5.9. *Let P be an ρ-closed program transformation. If all diagrams of a complete set of commuting diagrams for P are of the form*



and for every WHNF t with $s \xrightarrow{P,\mathbb{S}} t$, we have $s \downarrow_{tno}$; and every repeated application of the triangle diagram to $\xrightarrow{P,\mathbb{S}}$ terminates. Then for all T: $P_T \subseteq \geq_T$.

Proof. Let $s \xrightarrow{P} t$ and $\mathbb{S}$ be a surface context. By induction on the length of a normal order evaluation $\mathbb{S}[t] \xrightarrow{tno,*} r$ (r a WHNF), one can show that $\mathbb{S}[s] \downarrow_{tno}$: The base cases are covered, since if $\mathbb{S}[t]$ is a WHNF, then $\mathbb{S}[s] \downarrow_{tno}$. The induction step is as follows: Either $\xrightarrow{P,\mathbb{S}}$ is already a typed normal order reduction; then we are finished. Otherwise there is an applicable commuting (non-triangle) diagram for the first reduction of $\mathbb{S}[t] \xrightarrow{tno,*} r$ and then the induction hypothesis can be applied. Since repeated applications of the triangle diagram terminates, either a WHNF will be reached this way or one of the other diagrams must be applied after the repeated application. Then we can use the induction hypothesis. Thus by the context lemma 5.6 we have $P_T \subseteq \geq_T$ for all T. □

6 A Type-Dependent Program Transformation for Lists

We will show that the following transformation is correct:

(IdL) $(\mathtt{case}_{\mathtt{List}}\ s\ \mathtt{of}\ (\mathtt{Nil} \to \mathtt{Nil})\ ((\mathtt{Cons}\ x\ xs) \to (\mathtt{Cons}\ x\ xs))) \to (s :: [T])$

Note, that the corresponding untyped transformation is not correct, since e.g. $(\mathtt{case}_{\mathtt{List}}\ \mathtt{True}\ \mathtt{of}\ (\mathtt{Nil} \to \mathtt{Nil})\ ((\mathtt{Cons}\ x\ xs) \to (\mathtt{Cons}\ x\ xs)))\Uparrow_{no}$, but $\mathtt{True} \downarrow_{no}$

It is easy to see that (IdL) is ρ-closed. A complete set of forking diagrams and a complete set of commuting diagrams for (IdL)-transformation are in Figure 5. They can be derived by inspecting all overlappings of $\xrightarrow{\text{IdL},\mathbb{S}}$-transformations with tno-reductions.

Lemma 6.1. *Let $t \xrightarrow{IdL,\mathbb{S}} t'$. If t is a WHNF, then t' is a WHNF, and if t' is a WHNF and t is not a WHNF, then $t \xrightarrow{tno} t'' \xrightarrow{gcp} . \xrightarrow{gcp} \xrightarrow{gc} t'$ with $t \downarrow_{tno}$ by Theorem 4.3.*

Proof. The only nontrivial case is that $t = \mathtt{case}\ r\ \mathtt{of}\ \mathrm{alts}$ and $\mathbb{R}[t] \xrightarrow{\text{IdL},\mathbb{S}} \mathbb{R}[r]$, and $\mathbb{R}[r]$ is a WHNF. Due to typing $r \in \{\mathtt{Nil}, (\mathtt{Cons}\ s_1\ s_2)\}$, and $\mathbb{R}$ is a reduction context. Obviously, $\mathbb{R}[t] \xrightarrow{\mathtt{case},tno} . \xrightarrow{gcp.gcp.gc} \mathbb{R}[r]$. The claim $t \downarrow_{tno}$ follows from Theorem 4.3. □

Proposition 6.2. *If $t :: [T] \xrightarrow{IdL} t' :: [T]$, then $t \sim_{[T]} t'$.*

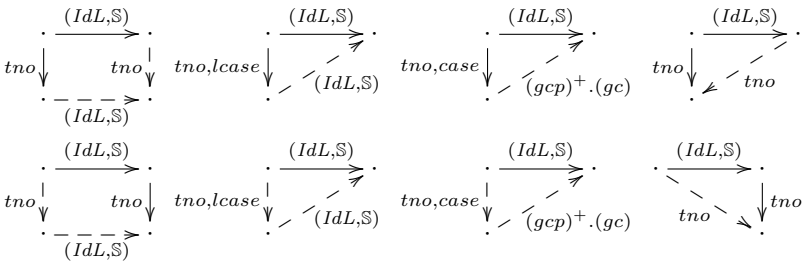


Figure 5: Complete sets of forking (top) and commuting diagrams (bottom) for IdL

Proof. This follows from Proposition 5.8 and Proposition 5.9, and by Lemma 6.1, since (gcp) and (gc) are applied as FV-closed and are correct thus program transformations by Theorem 4.3. □

7 Conclusion

We have developed a type-labeling of expressions in a polymorphically typed λ-calculus with cyclic let that demonstrates how to integrate polymorphic typing and contextual equivalence in a lambda-calculus extended with letrec, seq, case and constructors. We are convinced that the methods to add polymorphic typing and typed contextual equivalence and the corresponding reasoning methods can be applied to further equations in our calculus and also to other extensions of lambda-calculi like non-deterministic ones.

References

[AK97] Zena M. Ariola and Jan Willem Klop. Lambda Calculus with Explicit Recursion. *Inform. and Comput.*, 139(2):154–233, 1997.

[Gor99] Andrew D. Gordon. Bisimilarity as a theory of functional programming. *Theoret. Comput. Sci.*, 228(1-2):5–47, October 1999.

[JV09] Patricia Johann and Janis Voigtländer. A family of syntactic logical relations for the semantics of Haskell-like languages. *Information and Computation*, 207(2):341–368, 2009.

[Las98] Søren Bøgh Lassen. *Relational Reasoning about Functions and Nondeterminism*. PhD thesis, Faculty of Science, University of Aarhus, 1998.

[LL08] Søren B. Lassen and Paul Blain Levy. Typed Normal Form Bisimulation for Parametric Polymorphism. In *LICS 2008*, pages 341–352, 2008.

[MSC99] Andrew K. D. Moran, David Sands, and Magnus Carlsson. Erratic Fudgets: A semantic theory for an embedded coordination language. In *Coordination '99*, volume 1594 of *LNCS*, pages 85–102. Springer-Verlag, 1999.

[Pey03] Simon Peyton Jones. *Haskell 98 language and libraries: the Revised Report*. Cambridge University Press, 2003. `www.haskell.org`.

[Pit00] Andrew M. Pitts. Parametric Polymorphism and Operational Equivalence. *Math. Structures Comput. Sci.*, 10:321–359, 2000.

[Pit02] Andrew M. Pitts. Operational Semantics and Program Equivalence. In J. T. O'Donnell, editor, *Applied Semantics*, volume 2395 of *Lecture Notes in Comput. Sci.*, pages 378–412. Springer-Verlag, 2002.

[Sab08] David Sabel. *Semantics of a Call-by-Need Lambda Calculus with McCarthy's amb for Program Equivalence*. Dissertation, Goethe-Universität Frankfurt, Inst. für Informatik. FB Informatik und Mathematik, 2008.

[SP07] Eijiro Sumii and Benjamin C. Pierce. A bisimulation for type abstraction and recursion. *J.ACM*, 54(5), 2007.

[SS07a] Manfred Schmidt-Schauß. Correctness of Copy in Calculi with Letrec. In *Term Rewriting and Applications (RTA-18)*, volume 4533 of *LNCS*, pages 329–343. Springer, 2007.

[SS07b] Manfred Schmidt-Schauß. Correctness of Copy in Calculi with Letrec, Case and Constructors. Frank report 28, Institut für Informatik. Fachbereich Informatik und Mathematik. J. W. Goethe-University Frankfurt am Main, 2007.

[SSM08] Manfred Schmidt-Schauß and Elena Machkasova. A Finite Simulation Method in a Non-Deterministic Call-by-Need Calculus with letrec, constructors and case. In *Proc. of RTA 2008*, number 5117 in LNCS, pages 321–335. Springer-Verlag, 2008.

[SSS08] David Sabel and Manfred Schmidt-Schauß. A Call-by-Need Lambda-Calculus with Locally Bottom-Avoiding Choice: Context Lemma and Correctness of Transformations. *Math. Structures Comput. Sci.*, 18(03):501–553, 2008.

[SSSH09] Manfred Schmidt-Schauß, David Sabel, and Frederik Harwath. Contextual Equivalence in Lambda-Calculi extended with letrec and with a Parametric Polymorphic Type System. Frank report 36, Inst. f. Informatik, Goethe-Univ., Frankfurt, 2009.

[SSSS08] Manfred Schmidt-Schauß, Marko Schütz, and David Sabel. Safety of Nöcker's Strictness Analysis. *J. Funct. Programming*, 18(04):503–551, 2008.

[VJ07] Janis Voigtländer and Patricia Johann. Selective strictness and parametricity in structural operational semantics, inequationally. *Theor. Comput. Sci*, 388(1–3):290–318, 2007.

A The Derivation System for L_{LC}

In Figure 6 a derivation system for polymorphic types of untyped expressions in L_{LC} (i.e., ignoring the type labels) is defined, where the explicit typing of variables is placed into a type environment, i.e. variables have no built-in type for this derivation system. An environment Γ is a mapping from variables to types, where $\mathrm{Dom}(\Gamma)$ is the set of variables that are mapped by Γ. The notation $\Gamma, x :: T$ means a new environment where $x \notin \mathrm{Dom}(\Gamma)$. If the type may have a quantifier-prefix, then this is written explicitly. The only places where quantifiers are necessary, are the bindings in a `letrec`. Typing the construct $(\texttt{seq}\ s\ t)$ is omitted, since it is the same as for an application, where the constant `seq` has type $\texttt{seq} :: \forall a, b.a \to b \to b$.

(Var)
$$\Gamma, \{x :: S\} \vdash x :: S$$

(App)
$$\frac{\Gamma \vdash s :: S_1 \to S_2 \;\; \Gamma \vdash t :: S_1}{\Gamma \vdash (s\ t) :: S_2}$$

(Abs)
$$\frac{\Gamma, x :: S_1 \vdash s :: S_2}{\Gamma \vdash (\lambda x.s) :: S_1 \to S_2}$$

(Cons)
$$\frac{\Gamma, y :: \mathit{typeOf}(c) \quad \vdash \quad (y\ s_1 \ldots s_n) :: T}{\Gamma \vdash (c\ s_1 \ldots s_n) :: T} \text{ if } ar(c) = n$$

(Case)
$$\frac{\begin{array}{lll} \Gamma & \vdash & s :: K\ S_1 \ldots S_m \\ \Gamma, x_{1,1} :: T_{1,1}, \ldots x_{1,n_1} :: T_{1,n_1} & \vdash & t_1 :: T \\ \Gamma, x_{1,1} :: T_{1,1}, \ldots x_{1,n_1} :: T_{1,n_1} & \vdash & (c_1\ x_{1,1} \ldots x_{1,n_1}) :: K\ S_1 \ldots S_m \\ \cdots & & \cdots \\ \Gamma, x_{k,1} :: T_{k,1}, \ldots x_{k,n_k} :: T_{k,n_k} & \vdash & t_k :: T \\ \Gamma, x_{k,1} :: T_{k,1}, \ldots x_{k,n_k} :: T_{k,n_k} & \vdash & (c_k\ x_{k,1} \ldots x_{k,n_1}) :: K\ S_1 \ldots S_m \end{array}}{\Gamma \vdash (\texttt{case}_K\ s\ \texttt{of}\ ((c_1\ x_{1,1} \ldots x_{1,n_1}) \texttt{->}\ t_1) \ldots) :: T}$$

(Letrec)
$$\frac{\begin{array}{lll} \Gamma, x_1 :: \forall \mathcal{X}_1.T_1, \ldots, x_n :: \forall \mathcal{X}_n.T_n & \vdash & t_1 :: \forall \mathcal{X}_1.T_1 \\ \cdots & & \cdots \\ \Gamma, x_1 :: \forall \mathcal{X}_1.T_1, \ldots, x_n :: \forall \mathcal{X}_n.T_n & \vdash & t_n :: \forall \mathcal{X}_n.T_n \\ \Gamma, x_1 :: \forall \mathcal{X}_1.T_1, \ldots, x_n :: \forall \mathcal{X}_1.T_n & \vdash & t :: R \end{array}}{\Gamma \vdash (\texttt{letrec}\ x_1 = t_1, \ldots, x_n = t_n\ \texttt{in}\ t) :: R}$$

(Generalize)
$$\frac{\Gamma \vdash t :: T}{\Gamma \vdash t :: \forall \mathcal{X}.T} \quad \begin{array}{l} \text{if } \mathcal{X} = FTV(T) \setminus \mathcal{Y} \\ \text{where } \mathcal{Y} = \bigcup_{x \in FV(t)} \{FTV(S) \mid (x :: S) \in \Gamma\} \end{array}$$

(Instance)
$$\frac{\Gamma \vdash t :: \forall \mathcal{X}.S_1}{\Gamma \vdash t :: S_2} \text{ if } \rho(S_1) = S_2 \text{ with } \mathrm{Dom}(\rho) \subseteq \mathcal{X}$$

Figure 6: The type-derivation rules for L_{LC}

Proposition A.1. *Let t be a L_{LC}-expression. Then he following are equivalent:*

1. $\Gamma \vdash t :: T$, *where* $\mathrm{Dom}(\Gamma) = FV(t)$.
2. *There is some L_{PLC}-expression t' with $\varepsilon(t') = t''$, and there is some variable-permutation σ with $\sigma(t'') = t$.*

Proof. (Sketch)
If $\Gamma \vdash t :: T$ for some expression t, then the derived types also of subtypes are used as labels, where quantifiers remain only at the variables defined in letrec-environments. The renaming may be necessary to avoid clashes between the built-in types of variables in L_{PLC}.
The other direction is a bit harder and only sketched: If there is a type-labeling then the letrec-typeing rule will also be applicable, however, in general with a more general type. In order to make this proof explicit, the letrec-typing rule has to be modified into an iterative typing rule, which, given the types T_i of the $x_i's$ for $i = 1, \ldots, n$ in the environment $\{x_1 = t_1, \ldots, x_n = t_n\}$, computes the types T_i' of $T_i's$ and if $T_i' \succ T_i$, then it restarts this rules with $x_i :: T_i'$. The types become more general in every step until a fixpoint is reached. □

B Context Lemma for L_{PLC}

In this section we prove a context lemma for the language L_{PLC}, which shows that reduction contexts are sufficient for checking contextual preorder, where in contrast to the untyped context lemmas, we have to add certain conditions on type instantiations and variable occurrences.

We require the notion of *multicontexts*, i.e. terms with several (or no) typed holes $\cdot_i :: T_i$, where every hole occurs exactly once in the term. We write a multicontext as $C[\cdot_1 :: T_1, \ldots, \cdot_n :: T_n]$, and if the terms $s_i :: T_i$ for $i = 1, \ldots, n$ are placed into the holes $\cdot_i$, then we denote the resulting term as $C[s_1, \ldots, s_n]$.

Lemma B.1. *Let C be a multicontext with n holes. Then the following holds:*

If there are terms $s_i :: T_i$ with $i \in \{1, \ldots, n\}$ such that $C[s_1, \ldots, s_{i-1}, \cdot_i :: T_i, s_{i+1}, \ldots, s_n]$ is a reduction context, then there exists a hole $\cdot_j$, such that for all terms $t_1 :: T_1, \ldots, t_n :: T_n$ $C[t_1, \ldots, t_{j-1}, \cdot_j :: T_j, t_{j+1}, \ldots, t_n]$ is a reduction context, provided all expressions are well-typed.

Proof. We assume there is a multicontext C with n holes and there are terms $s_1, \ldots, s_n$ with $R_i = C[s_1, \ldots, s_{i-1}, \cdot_i :: T_i, s_{i+1}, \ldots, s_n]$ being a reduction context. Then there is an execution of the labeling algorithm starting with $C[s_1, \ldots, s_{i-1}, \cdot_i :: T_i, s_{i+1}, \ldots, s_n]$ where the hole is labeled with top, sub, or vis. We fix this execution and apply the same steps to $C[\cdot_1 :: T_1, \ldots, \cdot_n :: T_n]$ 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 unwinding algorithm then labels the hole with top, sub, or vis, the claim follows. □

A characterization of type-invariance of the pair (s, t) is as follows

Lemma B.2. *Let $s, t : T$ be well-typed expressions. Then the following two conditions are equivalent:*

1. $\{x \mid \in FV(s)$ *and* $FTV(bt(x)) \neq \emptyset\} = \{x \mid x \in FV(t)$ *and* $FTV(bt(x)) \neq \emptyset\}$.
2. $s \approx_{wt} t$

Proof. In order to show $s \approx_{wt} t$, we only show $\forall C[:: T] : C[s] \in L_{PLC} \implies C[t] \in L_{PLC}$, since the other direction is symmetric. Let $C[\cdot :: T]$ be a context such that $C[s] \in L_{PLC}$. Then it is easy to verify that $C[t] \in L_{PLC}$ must hold, since all type constraints are fulfilled, and the type labels of s and t are identical (thus the computation of $MonoTp$ above the hole of C is the same for $C[s]$ and $C[t]$) and since the constraint on `letrec`-expressions only depends on free variables and their built-in types, which are identical for s and t.

For the other direction, we have to show that s and t do not behave the same w.r.t. typing. Assume that s, t are typeable and of type T. Let $x \in FV(s) \setminus FV(t)$ such that $a \in FTV(bt(x))$. Let $C_1[\cdot :: T]$ be a context that binds all variables in $(FV(s) \cup FV(t)) \setminus \{x\}$, say $C_1[\cdot] := (\texttt{letrec}\ y_1 = y_1, \ldots y_n = y_n\ \texttt{in}\ [\cdot])$. Then $C_1[s]$ and $C_1[t]$ are bot L_{PLC}-expressions. The context C_2 is now constructed as

$$\begin{array}{ll} \texttt{letrec} & const^{::\forall c,d.c \to d \to c} = \lambda y_1.\lambda y_2.y_1 \\ & y^{::\forall a.a} = const\ y^{::a}\ [\cdot^{::T}] \\ \texttt{in} & \texttt{True} \end{array}$$

where we in addition assume that y, $const$ are not among the free variables of s, t. Then $C_2[C_1[\cdot]]$ is a context that distinguishes s, t: For t, the type variable a is quantifiable, and the pair is typeable, whereas this is false for s, since the type variable a occurring in the type of the free variable x prevents the generalization of a.

□

Lemma B.3. *Let $s, t : T$ be well-typed expressions, such that $FV(s) = FV(t)$. Then $s \approx_{wt} t$*

Proof. We only show $\forall C[:: T] : C[s] \in L_{PLC} \implies C[t] \in L_{PLC}$, since the other direction is symmetric. Let $C[:: T]$ be a context such that $C[s] \in L_{PLC}$. Then it is easy to verify that $C[t] \in L_{PLC}$ must hold, since all type constraints are full-filled, since the type labels of s and t are identical (thus the computation of $MonoTp$ above the hole of C is the same for $C[s]$ and $C[t]$) and since the constraint on `letrec`-expressions only depends on free variables and the built-in types, which are identical for s and t. □

Lemma B.4. *Let $n \geq 0$, let $s_i, t_i : T_i$ be well-typed expressions, such that for all i: $FV(s_i) = FV(t_i)$. Then $\forall C[\cdot_1 :: T_1, \ldots : \cdot_n :: T_n] : C[s_1, \ldots, s_n] \in L_{PLC} \iff C[t_1, \ldots, t_n] \in L_{PLC}$.*

Proof. We use induction on the number of holes n. If $n = 0$ then the claim obviously holds. For the induction step, let us assume that the precondition holds, and that $C[s_1, \ldots, s_n] \in L_{PLC}$. Lemma B.3 implies $C[t_1, s_2, \ldots, s_n] \in L_{PLC}$. Applying the induction hypothesis to the multicontext $C' = C[t_1, \cdot_2, \ldots, \cdot_n]$ shows the claim. □

Definition B.5. *For well-typed expressions $s, t :: T$, the relation $s \leq_{\mathbb{R},T} t$ holds iff:*

1. $FV(s) = FV(t)$.
2. *for all ρ where ρ is a (type-correct) type-instantiation and a variable-substitution such that variables are renamed, respectively replaced, by variables of an instance type, it holds: $\forall$ reduction contexts $R[\cdot :: \rho(T)]$: If $R[\rho(s)] \in L_{PLC}$ then $(R[\rho(s)]\downarrow_{tno} \implies R[\rho(t)]\downarrow_{tno})$.*

Lemma B.6. *If $s \leq_{\mathbb{R},T} t$, then for every instantiation ρ according to Definition 5.1, it holds $\rho(s) \leq_{\mathbb{R},\rho(T)} \rho(t)$.*

Proof. It is easy to verify that the first two conditions of Definition B.5 also hold for $\rho(s), \rho(t)$, since identical types are identically renamed or substituted and since the free variables of s and t are identical by the first property of Definition B.5. Now, the remaining parts follow from the Definition of $\leq_{\mathbb{R},T}$. □

Lemma B.7 (Context Lemma)**.** $\leq_{\mathbb{R},T} \subseteq \leq_T$

Proof. We prove the more general claim:

For all $n \geq 0$ and for all multicontexts $C[\cdot_1 :: T_1, \ldots, \cdot_n :: T_n]$ and for all well-typed expressions $s_1 :: T_1, ..., s_n :: T_n$ and $t_1 :: T_1, ..., t_n :: T_n$:
If for all $i = 1, \ldots, n$: $s_i \leq_{\mathbb{R},T_i} t_i$, then $C[s_1, \ldots, s_n] \approx_{wt} C[t_1, \ldots, t_n]$ and $C[s_1, \ldots, s_n] \in L_{PLC} \implies (C[s_1, \ldots, s_n]\downarrow_{tno} \Rightarrow C[t_1, \ldots, t_n]\downarrow_{tno})$.

The proof is by induction, where n, $C[\cdot_1 :: T_1, \ldots, \cdot_n :: T_n]$, $s_i :: T_i, t_i :: T_i$ for $i = 1, \ldots, n$ are given. The induction is on the measure (l, n), where

- l is the length of the typed normal order evaluation of $C[s_1, \ldots, s_n]$.
- n is the number of holes in C.

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

Now let $(l, n) > (0, 0)$. For the induction step we assume that the claim holds for all n', C', s'_i, t'_i, $i = 1, \ldots, n'$ with $(l', n') < (l, n)$. Let us assume that the precondition holds, i.e., that $\forall i : s_i \leq_{\mathbb{R},T_i} t_i$. Let C be a multicontext such that $C[s_1, \ldots, s_n] \in L_{PLC}$ and RED be the typed normal order evaluation of $C[s_1, \ldots, s_n]$ with $\mathrm{rl}(RED) = l$. The wt-invariance follows from Lemma B.4. For proving $C[t_1, \ldots, t_n]\downarrow_{tno}$, we distinguish two cases:

- There is some index j, such that $C[s_1, \ldots, s_{j-1}, \cdot_j :: T_j, s_{j+1}, \ldots, s_n]$ is a reduction context. Lemma B.1 implies that there is a hole $\cdot_i$ such that $R_1 \equiv C[s_1, \ldots, s_{i-1}, \cdot_i :: T_i, s_{i+1}, \ldots, s_n]$ and $R_2 \equiv C[t_1, \ldots, t_{i-1}, \cdot_i ::$

$T_i, t_{i+1}, \ldots, t_n]$ are both reduction contexts. Let $C_1 \equiv C[\cdot_1 :: T_1, \ldots, \cdot_{i-1} :: T_{i-1}, s_i, \cdot_{i+1} :: T_{i+1}, \ldots, \cdot_n :: T_n]$. From $C[s_1, \ldots, s_n] \equiv C_1[s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n]$ we have: RED is a typed normal order evaluation of $(C_1[s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n])$. Since C_1 has $n-1$ holes, we can use the induction hypothesis and derive $C_1[t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n] \downarrow_{tno}$, i.e. $C[t_1, \ldots, t_{i-1}, s_i, t_{i+1}, \ldots, t_n] \downarrow_{tno}$. This implies $R_2[s_i] \downarrow_{tno}$. Using the precondition we derive $R_2[t_i] \downarrow_{tno}$, i.e. $C[t_1, \ldots, t_n] \downarrow_{tno}$.

- There is no index j, such that $C[s_1, \ldots, s_{j-1}, \cdot_j :: T_j, s_{j+1}, \ldots, s_n]$ is a reduction context. If $l = 0$, then $C[s_1, \ldots, s_n]$ is a WHNF and since no hole is in a reduction context, $C[t_1, \ldots, t_n]$ is also a WHNF, hence $C[t_1, \ldots, t_n] \downarrow_{tno}$. If $l > 0$, then the first normal order reduction of RED can also be used for $C[t_1, \ldots, t_n]$, i.e, the position of the redex and the inner redex are the same. This normal order reduction can modify the context C, the number of occurrences of the terms s_i, the positions of the terms s_i, and s_i may be modified by a type instantiation due to a (cp)-reduction.

  We now argue that the elimination, duplication or variable permutation for every s_i can also be applied to t_i. More formally, we will show if $C[s_1, \ldots, s_n] \xrightarrow{no,a} C'[s'_1, \ldots, s'_m]$, then $C[t_1, \ldots, t_n] \xrightarrow{no,a} C'[t'_1, \ldots, t'_m]$, such that $s'_i \leq_{\mathbb{R}^X, T'_i} t'_i$. We go through the cases of which normal order reduction is applied to $C[s_1, \ldots, s_n]$ to figure out how the terms s_i (and t_i) are modified by the reduction step, where we only mention the interesting cases.

  - If the position of $\cdot_i$ is in an alternative of `case`, which is discarded by a (case)-reduction, or the position of $\cdot_i$ is in the argument of a `seq`-expression that is discarded by a (seq)-reduction, then s_i and t_i are both eliminated.
  - If the reduction is not a (cp)-reduction that copies a superterm of s_i or t_i, and s_i and t_i are not eliminated, then s_i and t_i can only change their respective position.
  - If the reduction is a (cp)-reduction that copies a superterm of s_i or t_i, then renamed copies $\rho_{s,i}(s_i)$ and $\rho_{t,i}(t_i)$ of s_i and t_i will occur, where $\rho_{s,i}$, $\rho_{t,i}$ are type- and variable-substitutions according to Definition 5.1. W.l.o.g. for all i: $\rho_{s,i} = \rho_{t,i}$. Lemma B.6 and the precondition shows $\rho_{s,i}(s_i) \leq_{\mathbb{R}, \rho_{s,i}(T_i)} \rho_{s,i}(t_i)$.

  Now we can use the induction hypothesis: Since $C'[s'_1, \ldots, s'_m]$ has a terminating sequence of normal order reductions of length $l-1$ we also have $C'[t'_1, \ldots, t'_m] \downarrow_{tno}$. With $C[t_1, \ldots, t_n] \xrightarrow{no,a} C'[t'_1, \ldots, t'_m]$ we have $C[t_1, \ldots, t_n] \downarrow_{tno}$. □

C Type-Safety of Typed Normal-Order Reductions

We prove in this section that Proposition 5.3 holds:

Proposition C.1. *Let $s :: T$ be a L_{PLC}-expression. Then $s \xrightarrow{tno} t$ implies that $t :: S$ is also well-typed and that $\varepsilon(s) \xrightarrow{no} \varepsilon(t)$. On the other hand, if $\varepsilon(s) \xrightarrow{no} t$, then there exists*

$t' :: T \in L_{PLC}$ *such that* $s \xrightarrow{tno} t'$ *and* $\varepsilon(t') = t$.
Moreover, typed normal-order reduction is unique up to renaming of type variables.

Proof. We have to argue that the following properties hold: If $s \to t$, then t is well-typed. i.e. the type constraints hold, and that the type is the same as for s.

We have to scan all the reduction rules and take account of all typing constraints: First of all, for all reductions, the type label remains the same for the redex, which also implies that the type of s, t are the same. So we only check whether the constraints are satisfied after the reduction, i.e. for t.

(lbeta) Since the abstracted variable remains monomorphic, the constraints for letrec are satisfied.

(cp) The variable is replaced by an expression of the same type, so we only have to check the instance itself. Since we assume that the distinct type variable convention holds, the quantifiable variables, and hence the instantiated ones, are distinct from all other type variables. Also the type variables that introduced by the instantiation are contained in the type of the variable at the occurrence that is instantiated. Hence no new variable conflicts are introduced, and thus all the constraints can be satisfied after the instantiation, in particular the letrec-constraint.

(abs) The following property also plays in role for the (case)-rule: for a constructor application $(c\, t_1 \ldots t_n) :: T$, the types of t_i are uniquely determined by T, which follows from the assumption on the type of constructors. Hence the types of the freshly introduces variables x_i are uniquely determined.

(case) Again the property above and the type constraint for (case) shows that the types of t_i and the corresponding pattern variable y_i must be identical, hence all types of the result are uniquely determined.

(seq) All required properties follow from the type of `seq`.

(llet) Since we assume the distinct variable convention for expression variables as well as for type variables, the quantifiable type variables do not change, and hence the letrec-constraints remain satisfied.

(lapp),(lseq),(lcase) Easy.

Uniqueness follows, since the for every reduction, the types of subexpressions are uniquely determined, up to a renaming of typed variables as mentioned above for the rule (cp). □