# 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. Using type-labels for all subexpressions of the typed expressions, 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 $\Lambda\alpha.(\texttt{letrec}\ x :: \alpha = x :: \alpha\ \texttt{in}\ x :: \alpha)$ 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. The 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 proof.

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. 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`(written as $[\cdot]$) 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.

**Syntax of Types** (Quantifier-free) Types $T$ in the polymorphic extended lambda-calculus have the following syntax $T ::= X \mid (T \to T) \mid (K\ T_1 \ldots T_{ar(K)})$, 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. Note that the variables stand for unquantified 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 type $T$, perhaps quantified, is denoted as $FTV(T)$. For instance, $FTV(\forall a, b.(a \to c) \to d \to a) = \{c, d\}$. 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.

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

Binding positions are $x, t$ in "$x = t$" and are called *let-positions*, where the other positions are called *non-let-positions*. We assume that for every universal and other 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 $x$. At binding position the variable must be labeled with its built-in type.

There is a scoping for type variables within expressions, which is similar to the scoping of system $F$, using the convention that the universal type $s :: \forall\mathcal{X}.T$ binds the free type-variables occurring in the types of subexpressions of $s$. $L_{LLC}$*-Contexts* $C$ are now typed, where the the hole is also labeled with a type, written $\mathbb{C}[\cdot :: T]$, where $T$ may be quantified if the hole is at a let-position. An example for the type labels is $\lambda x.x$ which can be sensibly labeled as $(\lambda x :: a.x :: a) :: (\forall a.a \to a)$. Accordingly, an example of a $L_{LLC}$-context is $(\lambda x :: a.[\cdot :: a]) :: (\forall a.a \to a)$.

**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* $\rho$ 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 $\mathrm{VCod}(\rho) := \bigcup_{X \in \mathrm{Dom}(\rho)} FTV(\rho(X))$. We say $\forall\mathcal{Y}.T'$ is an *instance* of a type $\forall\mathcal{X}.T$, denoted as $\forall\mathcal{Y}.T' \preceq \forall\mathcal{X}.T$, iff there is a substitution $\sigma$ with $\mathrm{Dom}(\sigma) \subseteq \mathcal{X}$, $\sigma(T) = T'$ and $\mathcal{Y} \subseteq \mathrm{VCod}(\sigma) \setminus FTV(\forall\mathcal{X}.T)$. The latter condition prevents a variable capture.

**Example 2.1.** *Let $T$ be the type $\forall a, b.a \to b$. Then* `Int` $\to$ `Int` *is an instance of $T$, as well as $\forall a.a \to$* `Int`*, where the latter has a variable name in common with $T$. A slightly more complex case is that $\forall a.[a] \to$* `Int` $\to c$ *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*

| | | |
|---|---|---|
| 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 $MonoTp$

*variables are not repeated) for the composition may be:*

$$(\lambda f :: (b \to c).(\lambda g :: (a \to b).\\ (\lambda x :: a.(f\ (g\ x) :: b) :: c) :: (a \to c)) :: ((a \to b) \to a \to c))\\ :: \forall a, b, c.(b \to c) \to (a \to b) \to a \to c$$

## 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.

To correctly deal with the scoping of type variables, we need some preparations. The problems are similar to system F, where forming $\Lambda X.t$ for a type variable $X$ and an expression is only permitted, if there are no free (expression-)variables in $t$ whose type contains $X$ as free type variable (see e.g. [GTL94]).

We define the following two sets of free type variables:

$$\begin{array}{lcl} FTTV(t,x) & := & \{a \mid a \in FTV(S), (x :: S) \text{ is a free occurrence of } x \text{ in } t\} \\ FTFV(t) & := & \bigcup_{x \in FV(t)} FTTV(t,x) \end{array}$$

We have two *Scoping Assumptions* for types: Expression must be *well-scoped*, where we say that $t$ is well-scoped (written $ws(t)$), iff for every subexpression $s :: \forall \mathcal{X}.T$, we have $\mathcal{X} \cap FTFV(s) = \emptyset$. This condition prevents unwanted capture of type variables. The second assumption is the *distinct type variable assumption*: bound type variables are all distinct and also distinct from all free type-variables. The latter can be achieved for well-scoped terms 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.

**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 non-let-positions, 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 $\mathcal{X} = FTV(T_i') \setminus \left(\bigcup_{x \in FV(t_i) \setminus \{x_1, \ldots, x_n\}} FTTV(t_i, x)\right)$ must be satisfied.

**Definition 3.1.** *If a $L_{LLC}$-expression $t :: T$ satisfies all the type constraints above, and if $ws(t)$, then the expression $t :: T$ is* well-typed. *The language $L_{PLC}$ consists exactly of the well-typed $L_{LLC}$-expressions.*

**Example 3.2.** *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 and type labels are as follows:* $id :: \forall a.a \to a$. *The two occurrences of $id$ are differently type labeled as* $\texttt{Bool} \to \texttt{Bool}$ *and* $\texttt{List}(\texttt{Bool}) \to \texttt{List}(\texttt{Bool})$. *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.*
*A non-well-scoped expression is* $(\texttt{letrec}\ y = ((x :: a)\ y) :: \forall a.a\ \texttt{in}\ y)$, *since $x$ is free, but its type variable $a$ is bound.*

## 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. 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 transformation 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

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

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

Figure 2: Searching the normal-order redex using labels

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 $s^{\mathsf{top}}$, 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 $(c\ x_1 \ldots x_n)$ 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 $(\mathtt{letrec}\ \mathit{Env}\ \mathtt{in}\ v)$.

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 $t \Uparrow_{no}$.*

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

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

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

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

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

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

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

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

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

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

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

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

Figure 3: Normal-order rules

**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* ws-invariant *(denoted with $s \approx_{ws} t$) iff for all $\mathbb{C}[\cdot :: T]$: $ws(\mathbb{C}[s]) \iff ws(\mathbb{C}[t])$. Contextual equivalence $\sim_T$ is defined as follows:*

$$s \leq_T t \quad \textit{iff} \quad s \approx_{ws} t \wedge \forall \mathbb{C}[\cdot :: T] : ws(\mathbb{C}[s]) \implies (\varepsilon(\mathbb{C}[s])\downarrow_{no} \Rightarrow \varepsilon(\mathbb{C}[t])\downarrow_{no})$$
$$s \sim_T t \quad \textit{iff} \quad s \leq_T t \wedge t \leq_T s$$

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

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 neither well-scopedness nor 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 to a type $T$ is denoted $P_T$. A program transformation $P$ is called *correct* iff for all $(s,t) \in P_T$, the relation $s \sim_T t$ holds. Analogously, for untyped programs, a program transformation is a binary relation

(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 $t_j$

(ucp) $(\texttt{letrec}\ x = s, Env\ \texttt{in}\ \mathbb{S}[x]) \to (\texttt{letrec}\ Env\ \texttt{in}\ \mathbb{S}[s])$
if $\mathbb{S}$ is a surface context[1] and $x$ does not occur in $s, S$ and $Env$.

(ucp) $(\texttt{letrec}\ x = s, y = \mathbb{S}[x], Env\ \texttt{in}\ t) \to (\texttt{letrec}\ y = \mathbb{S}[s], Env\ \texttt{in}\ t)$
if $\mathbb{S}$ is a surface context and $x$ does not occur in $s, \mathbb{S}$, $Env$ and $t$.

(ucp) $(\texttt{letrec}\ x = s\ \texttt{in}\ \mathbb{S}[x]) \to \mathbb{S}[s]$
if $\mathbb{S}$ is a surface context and $x$ does not occur in $s$ and $\mathbb{S}$.

Figure 4: Further program transformations

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. [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 extended calculus can be found in [SSSH09]). This implies that all reduction rules of Figure 3 and the optimizations *garbage collection* (gc) and *unique copying* (ucp) (see Figure 4) are correct (untyped) program transformations for $L_{LC}$, which 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 $\mathbb{C}[s] \approx_{ws} \mathbb{C}[t]$, then $\mathbb{C}[s] \to \mathbb{C}[t]$ is a correct typed program transformation for $L_{PLC}$.*

A sufficient condition for ws-invariance is $FV$-closedness: We say a program transformation $P$ is *$FV$-closed* iff for all $(s,t) \in P$ it holds $FV(s) = FV(t)$ and for all $x \in FV(s) : FTTV(s,x) = FTTV(t,x)$.

**Lemma 4.4.** *Let $s, t : T$ be well-typed expressions, such that $FV(s) = FV(t)$ and for all $x \in FV(s) : FTTV(s,x) = FTTV(t,x)$. Then $s \approx_{ws} t$.*

Thus, we could restrict the transformation of Theorem 4.3 to $FV$-closed relations, which is only necessary for the rules which modify the occurrences of free variable (i.e. (case), (seq), (gc), and (ucp)). Note also that the formulation of the typed program transformations

[1] surface-context: the hole is not in an abstraction

in the theorem does not provide a scheme how to obtain for a well-typed expression $s$ a transformed expression $t$ which remains equally typed. I.e., for automatized 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 are in general 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 reduction 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) and (llet-e), where the rule (llet-e) may rename and add quantifiers and (cp) may instantiate types during reduction.

**Definition 5.1** (Quantifier Distribution)**.** *For the rule (llet-e) we have to define how the types are modified (i.e. generalized). The (llet-e) rule with types is as follows:*

$$\texttt{letrec}\ Env, x = (\texttt{letrec}\ x_1 = t_1 :: \forall \mathcal{Y}_1.T_1, \ldots, x_n = t_n :: \forall \mathcal{Y}_n.T_n\ \texttt{in}\ s) :: \forall \mathcal{X}.T\ \texttt{in}\ t$$
$$\rightarrow\ \texttt{letrec}\ Env, x = s :: \forall \mathcal{X}.T, x_1 = t_1 :: \forall \mathcal{X} \cup \mathcal{Y}_1.T_1, \ldots, x_n = t_n :: \forall \mathcal{X} \cup \mathcal{Y}_1.T_n\ \texttt{in}\ t$$

*After the type quantifier distribution, a type variable renaming has to be performed to satisfy the distinct variable convention for types.*

The normal-order (cp)-rules always copy to non-let positions and have to be accompanied by a local type-instantiation:

**Definition 5.2** (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* $\rho$ *is the type-substitution with* $\rho(T) = S$*, and where* $\rho(v)$ *means the expression* $v$*, after application of the instantiation* $\rho$ *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.*

**Example 5.3.** *This is an example for the (cp)-rules and their effect on types. Let concatMap 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 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* $xs' :: [[c]]$ *and* $f' :: [c] \to [c]$ *are fresh variables. The type of the copied body of concatMap is an instance of the type computed at the binding, namely* $([c] \to [c]) \to [[c]] \to [c]$*, and then (lbeta) will result in*
$\texttt{letrec}$ *concatMap* $= \ldots, id = \lambda x.x, \ldots\ \texttt{in letrec}\ f' =$ *id* $\texttt{in}\ (\lambda xs'.\texttt{case}\ldots)$

*where id* $:: [c] \to [c]$*, and* $(\lambda xs'.\texttt{case}\ldots)$ *is labeled with type* $[[c]] \to [c]$*.*

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, and the binding structure of types remains intact, i.e. expressions remain well-scoped.

Inspecting all reduction rules and their corresponding type modifications shows:

**Proposition 5.4.** *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* $t' \in L_{PLC}$ *such that* $s \xrightarrow{tno} t'$ *and* $\varepsilon(t') = t$*.*

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

## 5.2 Proof Techniques for Typed Contextual Equality

A first advantage of Theorem 5.5 is that we are able to prove a typed context-lemma, where the proof is an adaptation from [SSSS08].

**Definition 5.6.** *A program transformation $P$ is $\delta\rho$-closed if it is $FV$-closed, and:*

- *For all $\rho$ where $\rho$ 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 $ws(\rho(s))$, then $(\rho(s), \rho(t)) \in P$*
- *For all $\delta$ where $\delta$ may replace variables with fresh variables where the built-in type of the variables may be generalized by extending the top-quantifier, and $\delta$ may rename type-variables, it holds: For all $(s,t) \in P$: If $ws(\delta(s))$, then $(\delta(s), \delta(t)) \in P$.*

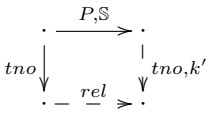
A surface context $\mathbb{S}$ is a context where the hole is not inside the body of an abstraction.

**Lemma 5.7** ($\mathbb{S}$-Context Lemma)**.** *Let $P$ be a $\delta\rho$-closed program transformation, which preserves convergence, i.e. for all $(s,t) \in P$ and for all surface contexts $\mathbb{S}$: $ws(\mathbb{S}[s]) \implies (\mathbb{S}[s]\downarrow_{tno} \implies \mathbb{S}[t]\downarrow_{tno})$. Then for all $T$: $P_T \subseteq \leq_T$ holds.*

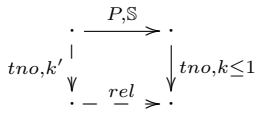
We define forking and commuting diagrams adapted to our needs in later proofs, where $\xrightarrow{P,\mathbb{S}}$ means an application of $P$ in an $\mathbb{S}$-context (of proper type) to a $L_{PLC}$-expression.

**Definition 5.8.** *Let $P$ be a $\delta\rho$-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 :*



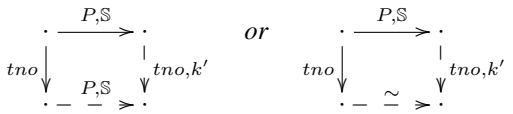

*commuting diagram :*




*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.*

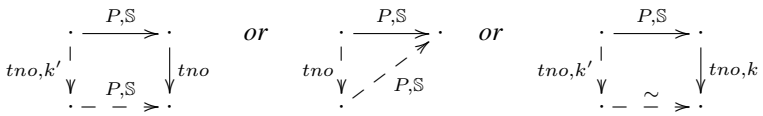
**Proposition 5.9.** *Let $P$ be a $\delta\rho$-closed program transformation. If all diagrams of a complete set of forking diagrams for $P$ are of the form*



*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 cases are covered, since either $P$ preserves WHNFs, or $\mathbb{S}[s] \sim_{T'} \mathbb{S}[t]$. 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; or 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 5.7 we have $P_T \subseteq \leq_T$ for all $T$. □

**Proposition 5.10.** *Let $P$ be an $\delta\rho$-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 the first reduction of $\mathbb{S}[t] \xrightarrow{tno,*} r$ is already a typed normal order reduction; then we are finished. Otherwise there is an applicable commuting 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 or one of the other diagrams must be applied. Then we can use the induction hypothesis. Thus by the context lemma 5.7 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) $\quad (\mathtt{case}_{\mathtt{List}}\ s\ \mathtt{of}\ (\mathtt{Nil} \mathtt{->} \mathtt{Nil})\ ((\mathtt{Cons}\ x\ xs) \mathtt{->} (\mathtt{Cons}\ x\ xs))) \to (s :: [T])$

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

It is easy to see that (IdL) is $\delta\rho$-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

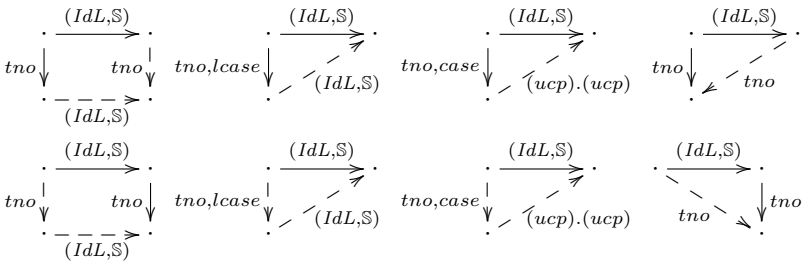



Figure 5: Complete sets of forking (top) and commuting diagrams (bottom) for $IdL$

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 $t \xrightarrow{tno} t'' \xrightarrow{ucp} . \xrightarrow{ucp} t'$ with $t \downarrow_{tno}$ by Theorem 4.3.*

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

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

*Proof.* This follows from Proposition 5.9 and Proposition 5.10, and since (ucp) is a correct program transformation by Theorem 4.3. □

## 7 Conclusion

We have developed a type-labeling of expressions in a polymorphically typed $\lambda$-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.

[GTL94] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*. Cambridge University Press, 1994.

[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.

[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.