Contextual Equivalence in Lambda-Calculi extended with letrec and with a Parametric Polymorphic Type System

Manfred Schmidt-Schauß and David Sabel and Frederik Harwath

Technical Report Frank-36

Research group for Artificial Intelligence and Software Technology
Institut für Informatik,
Fachbereich Informatik und Mathematik,
Johann Wolfgang Goethe-Universität,
Postfach 11 19 32, D-60054 Frankfurt, Germany
schauss@cs.uni-frankfurt.de

26. January 2009

Abstract. This paper describes a method to treat contextual equivalence in polymorphically typed lambda-calculi, and also how to transfer equivalences from the untyped versions of lambda-calculi to their typed variant, where our specific calculus has letrec, recursive types and is non-deterministic. An addition of a type label to every subexpression is all that is needed, together with some natural constraints for the consistency of the type labels and well-scopedness of expressions. One result is that an elementary but typed notion of program transformation is obtained and that untyped contextual equivalences also hold in the typed calculus as long as the expressions are well-typed. In order to have a nice interaction between reduction and typing, some reduction rules have to be accompanied with a type modification by generalizing or instantiating types.

1 Introduction

The semantics of programming languages based on syntax and operational semantics using Morris' style contextual equivalence is a successful approach to program semantics for a wide variety of program calculi and programming concepts, 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. If it comes to modeling languages and to investigate semantic issues, in particular correctness of program transformations, then the picture changes: Generally, investigations on contextual semantics try to cut down the complexity of the analysis by a restriction to untyped or very weakly typed calculi or core programming languages. There are some exceptions, like e.g. an investigation on simply-typed PCF [Gor99], 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 [VJ07,SP07,LL08]. However, those results cannot be used to argue for a smooth relationship between polymorphically typed calculi (in particular non-deterministic ones with letrec and recursive types) and their untyped variants w.r.t. contextual equivalence, nor for an intuitive notion of program transformation. The calculi in [Pit00,VJ07,SP07] change the termination behavior w.r.t. their untyped variants, and in particular, it is not clear how to extend the methods to letrec-calculi that also comprise various forms of non-determinism. However, these approaches using logical relations do not give an immediate insight into polymorphically typed contextual equivalence nor into the relationship between the contextual equivalence of typed and untyped versions of calculi.

An example for the inherent problems is the program transformation $\texttt{if}\ x\ \texttt{then}\ x\ \texttt{else}\ x \quad \to \quad x$, which is restricted to the type `Bool` before the transformation and unrestricted after the transformation.

Our overall goal is to investigate contextual equivalence for polymorphic and extended, non-deterministic program calculi such that the results and methods can be transferred to polymorphically typed programming languages.

One principle of adding polymorphic types is the following: Typed terms, typed WHNFs, typed contexts, and typed normal-order reductions should be terms, WHNFs, contexts, and normal-order reductions after omitting the types. For normal-order reductions, the stronger requirement is that given two typed terms s, t, then $s \to t$ is a typed normal-order reduction iff it is an untyped normal-order reduction. We show that a nice consequence will be that untyped contextual equalities $s \sim t$ between expressions s, t of equal type also hold using the typed calculi (under well-scopedness restrictions), since there are only fewer contexts.

There are several main approaches to add typing: (i) The Church-style that adds type-labels to every object of the language, (ii) the derivation-style, where no type-labels are necessary, and the types are computed w.r.t. type-environments for free variables; and (iii) the Curry-style, where types are part of the syntax and reduction rules also deal with types. The inclusion of types into the syntax and reduction rules, like in system F, severely changes the operational properties, e.g. termination properties of expressions are different from the untyped version, so we only investigate the other approaches.

We add a polymorphic type system using labels to the calculus of [SSS08,Sab08], which is a lambda-calculus extended with letrec, case, constructors, seq, and with McCarthy's non-deterministic primitive `amb`. Our calculus embodies let-polymorphism, i.e. we use forall-types for let-bound variables, and we permit instance types at occurrences of the variables. This gives enough flexibility during

reduction to show that reduction does not lead to type errors. The typing of the letrec-construct and its interaction with the reduction rules enforces a dynamic, but obvious, generalization of the forall-types, which also solves the confusing problem that (llet-e)-reductions may violate the type-scope. Our method to add type labels also has the potential to simplify the type treatment in the core languages of compilers of functional programming languages.

The main results of this paper are: (i) the construction of a polymorphic type system for a lambda-calculus with letrec, case, constructors and non-deterministic primitive `amb` which uses type labels fixing the type of all subexpressions such that also dynamic type are prevented. (ii) We show a natural relationship between typed and untyped may- and must-convergence, where we use the translation method [SSNSS08], to lift all equivalences of the untyped calculus (see [SSS08,Sab08]) to the typed version. We also show that there are some additional correct program transformations that depend on the types of expressions and which do not make sense without types. The reasoning requires a context-lemma, which also holds for our typed calculus. Thus we demonstrate that all problems of introducing polymorphic typing to an untyped program calculus can be overcome without sacrificing generality.

Outline First we define the language, then describe a derivation-style polymorphic type system for the unlabeled language. After describing the consistency rules for the type-labeling we introduce the small-step operational semantics of the language and show that the reductions keeps typing. Then we define contextual equivalence and transfer a set of program transformations from the untyped setting in the typed calculus. Finally, we prove that a type dependent program transformation preserves contextual equivalence.

2 Syntax of the Polymorphic Typed Lambda Calculus

We describe the polymorphically typed language L_{PLC}, a polymorphically typed variant of the calculus in [Sab08,SSS08], which employs cyclic sharing using a letrec [AK97].

Syntax of Expressions 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 type constructors `Bool` and $[\cdot]$ for lists 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 the 0-ary constructors `True`, `False` for type constructor `Bool`, and the 0-ary constructor `Nil` and the infix binary constructor ":" for lists with type constructor $[\cdot]$ are among the constructors.

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

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

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. 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` is recursive, 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.

We assume that every subexpression and every pattern of L_{PLC} has a type label, which will be explained and discussed later.

Syntax of Types Types T without quantifier in the polymorphic extended lambda-calculus have the following syntax.

$$T ::= X \mid (T \to T) \mid (K\ T_1 \ldots T_{ar(K)})$$

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 may add a quantifier at the top of types: $\forall X_1, \ldots, X_n.T$, where X_i are type variables. 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 $FTV(T)$. Additionally we require the notion of *contexts* C, which are like expressions with the difference that there exists a single occurrence of a constant (the hole $[\cdot]$) at a subterm position, that is also labeled with a type, written $C[\cdot :: T]$.

Example 2.1. The polymorphic type of the identity $\lambda x.x$ is $\forall a.a \to a$. The type of the function composition $\lambda f, g, x.f\ (g\ x)$ is $\forall a, b, c.(b \to c) \to (a \to b) \to a \to c$.

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.

3 Derivation System for Types

In figure 1 a derivation system for polymorphic types of expressions (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 constructs (`amb` s t) and (`seq` s t) is omitted, since it is the same as for an application, where the types of the constants are `amb` $:: \forall a : a \to a \to a$ and `seq` $:: \forall a, b.a \to b \to b$.

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

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

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

(Cons) $$\frac{\Gamma \vdash s_1 :: S_1\ ; \ldots ; \Gamma \vdash s_n :: S_n \quad \Gamma, y :: \mathit{typeOf}(c) \vdash (y\ s_1 \ldots s_n) :: T}{\Gamma \vdash (c\ s_1 \ldots s_n) :: T} \text{ if } ar(c) = n$$

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

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

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

Fig. 1. The type-derivation rules

Using the rules of the derivation system, a standard polymorphic type system can be implemented that computes types as greatest fixpoints using iterative processing. By standard reasoning, there is a most general type of every ex-

pression, however, typability is undecidable, since the semi-unification problem [KTU93] can be encoded. Stopping the iteration, like in Milner's type system, leads to a decidable, but incomplete type system.
A *(type-)substitution* ρ substitutes types for type variables, where we define $\mathrm{Dom}(\rho)$ to be the set of type variables X with $\rho(X) \neq X$, $\mathrm{Cod}(\rho) = \{\rho(x) \mid x \in \mathrm{Dom}(\rho)\}$ 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 σ 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. We also allow the notion of instance, if before and/or after the substitution, the bound variables are renamed, where capture of type variables is disallowed.

Example 3.1. Let T be the type $\forall a, b.a \to b$. Then $\texttt{Int} \to \texttt{Int}$ is an instance of T, as well as $\forall a.a \to \texttt{Int}$, where the latter has a variable name in common with T. A slightly more complex case is that $\forall a.[a] \to \texttt{Int} \to c$ is an instance of $\forall a, b.a \to b \to c$; note that c is a free type variable in this case.

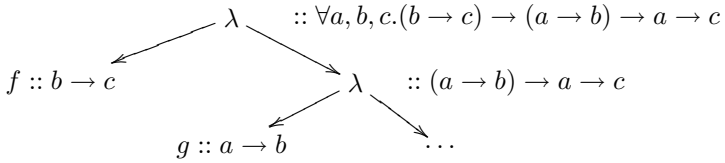
4 Type Consistency Rules in the Parametric Polymorphic Lambda Calculus

In this section we will detail the assumptions on the Church-style polymorphic type system that fixes the type also of subexpressions. We will define consistency rules that ensure that the labeling of the subexpressions is not contradictory.
We assume that for every type T, including quantified types and types containing free type variables, 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. This means that renamings of bound variables now have to keep exactly the type. We also add a scoping for type variables within expressions, using the convention that the $\forall$-quantifier binds the free type-variables of the types of labeled subexpressions.
Quantifiers are only permitted at x, t of letrec-bindings $x = t$, or at the top term. These positions are called *let-positions*, the other positions are called *non-let-positions*.

Example 4.1. This example shows a type-labeled expression. The type of the composition is $\texttt{(.)} :: \forall a, b, c.(b \to c) \to (a \to b) \to a \to c$. A type labeling (the types of some variables are not repeated) for the composition may be:

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

An illustration is as follows:



The set $FTFV(t)$ of all free type variables of free occurrences of variables is defined as $FTFV(t) := FTFV_{\emptyset}(t)$, where $FTFV_W$ computes the free type variables of free (term) variables excluding a set of free variables W (which is usually a set of let-bound variables) as follows:

$$FTTV(t,x) := \{a \mid a \in FTV(S), (x :: S) \text{ is a free occurrence of } x \text{ in } t\}$$
$$FTFV_W(t) := \bigcup_{x \in FV(t) \setminus W} FTTV(t,x)$$

Our *Scoping Assumptions* are as follows. For bound type variables, we assume that they are all distinct and also distinct from all free type-variables. An expression t is called *well-scoped*, $ws(t)$, 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. Note that the derivation system in figure 1 satisfies the well-scoped condition.
For a type judgment or labeled subexpression $t :: T$ we can compute a corresponding maximally quantified type w.r.t. a set W of variables as

$$GenType_W(t,T) ::= \forall \mathcal{X}.T, \text{ where } \mathcal{X} := FTV(T) \setminus FTFV_W(t).$$

For convenience we define $GenType(t,T) := GenType_{\emptyset}(t,T)$ for the maximally quantified type of $t :: T$ w.r.t. the empty set.
The computation of the maximally quantified type is justified by the generalization rule of the type derivation system. Below we use the condition $T \preceq GenType_W(s,S)$ in letrec-expressions, which means that a type label first has to be generalized to a maximum using quantifiers, and then an instance T is used.

Type-Constraints: The type-label S of a variable $x \in V_T$ must be an instance of the built-in type T of x. Lambda-bound variables and variables in case-patterns must have a quantifier-free (but not necessarily ground) type which must be its built-in type. Let-bound variables have their built-in type as type-label at the let-positions (which may be quantified) and on non-let positions the type-label must be a quantifier-free instance of the built-in type.
We also assume that constants and constructors have a type-label S . The label S of a constructor c must be an instance of the predefined type of c, i.e. for constructor occurrences $c :: S$, the constraint $S \preceq typeOf(c)$ must hold. Similarly for `amb`and `seq` which are of the types $\forall a.a \to a \to a$ or $\forall a,b.a \to b \to b$, respectively.
The (non-let)-type of an expression can be computed by the function *nltype* defined in figure 2 based on the types of the subexpressions. For non-let-positions, the constraint is that the type-label must be identical to the computed type by *nltype*.
In the typing of a case-expression $\texttt{case}_K\ s :: S\ \texttt{of}\ (pat_1\ \texttt{->}\ t_1) \ldots (pat_n\ \texttt{->}\ t_n)$, the types of s and the patterns (which are typed like constructor expressions) must be identical; also the types of the subexpressions t_i must be identical.

Application
$(s :: S_1 \to S_2\ t :: S_1) \mapsto S_2$
Constructor expressions
$(c :: (S_1 \to \ldots \to S_n \to S)\ s_1 :: S_1 \ldots s_n :: S_n) \mapsto S$
Abstractions
$(\lambda x :: S_1.s :: S_2) \mapsto S_1 \to S_2$
Case-expression
$$\left.\begin{array}{l}(\texttt{case}_K\ s :: S\ \texttt{of}\ ((c_{K,1}\ x_{1,1} \ldots x_{1,n_1}) :: S\ \texttt{->}\ t_i :: T) \\ \qquad \ldots \\ \qquad ((c_{K,m}\ x_{m,1} \ldots x_{m,n_m}) :: S\ \texttt{->}\ t_m :: T))\end{array}\right\} \mapsto T$$
Let-expression
$(\texttt{letrec}\ x_1 :: S_1 = t_1 :: T_1, \ldots, x_n :: S_n = t_n :: T_n\ \texttt{in}\ t :: S) \mapsto S$

Fig. 2. Computation of *nltype*

The constraint for the permitted types of the bindings in a let-expression is a bit more complex. In the expression

$$(\texttt{letrec}\ x_1 :: S_1 = t_1 :: T_1, \ldots, x_n :: S_n = t_n :: T_n\ \texttt{in}\ t),$$

let $A_i = \mathit{nltype}(t_i)$ for $i = 1, \ldots, n$. Then the constraints $S_i \preceq T_i, i = 1, \ldots, n$ and $A_i \preceq T_i \preceq \mathit{GenType}_{\{x_1,\ldots,x_n\}}(t_i, A_i)$, $i = 1, \ldots, n$ must be satisfied.
For the top-expression we can assume that the most general type is used.

Definition 4.2. *If an expression $t :: T$ satisfies all the type constraints above, then we call the type labeling* admissible*, and the expression $t :: T$* well-typed*.*

Example 4.3. The expression $\texttt{letrec}\ id = \lambda x.x\ \texttt{in}\ (id\ \texttt{True}, id\ \texttt{Nil})$ is well-typed, where the types are as follows: $id :: \forall a.a \to a$. The two occurrences of id are differently typed as $\texttt{Bool} \to \texttt{Bool}$ and $[\texttt{Bool}] \to [\texttt{Bool}]$. A variant of this example is $\texttt{letrec}\ id = \lambda x.x, y = id\ \texttt{in}\ (id\ \texttt{True}, y\ \texttt{Nil})$, where $y :: \forall b.[b] \to [b]$ at the let-position and $y :: [c] \to [c]$ 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 is bound.

Example 4.4. The expression $(\texttt{letrec}\ x = x\ \texttt{in}\ (x\ \ x))$ can be type-labeled as follows: $(\texttt{letrec}\ x :: (\forall a.a) = x\ \texttt{in}\ (x :: \texttt{Bool} \to \texttt{Bool}\ \ x :: \texttt{Bool}))$. The type of the whole expression is $\texttt{Bool}$. Note that the two occurrences of x in the expression $(x\ x)$ must be labeled differently.

From a typing point of view, the derivation system and the type-labeling are equivalent mechanisms.

5 Small-Step Operational Semantics of L_{PLC}

A reduction step consists of two operations: first finding the normal-order redex, then applying a reduction rule. For the search we use three atomic labels

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

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

Fig. 3. Searching the normal-order redex using labels

sub, top, vis, where top means the top-expression, sub means a subterm reduction, and vis means visited. For an expression s the shifting algorithm, which is specified in figure 3, starts with s^{top} and uses the given rules exhaustively. It 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*.

Normal-order reduction rules are defined in figure 4, without mentioning types, where we assume that a non-failing execution of the nondeterministic labeling algorithm was used before. If all possible executions fail, then no normal-order reduction is applicable. In figure 4, a cv-expression is an expression of the form $(c\ x_1 \dots x_n)$ where c is a constructor and x_i are variables. A *value* is an abstraction or a constructor-expression $(c\ t_1 \dots t_n)$. Normal-order reduction is non-deterministic due to the amb. Note that evaluation should be fair to implement the desired behavior of the amb-operator. Nevertheless, for our equational theory based on may- and must-convergence we can omit fairness, since our must-convergence has some kind of fairness built-in (see [CHS05,RV07,SSS08,Sab08]). The inheritance of the typing to the result is standard in most cases. The rules where the typing is not standard will be discussed in the next subsection.

A weak head normal form (WHNF) is a value v or an expression $(\texttt{letrec}\ Env\ \texttt{in}\ v)$, where v is a value.

Note that there may be closed and stuck expressions that are not WHNFs like $(\texttt{letrec}\ x = \texttt{seq}\ x\ x\ \texttt{in}\ x)$; these terms are contextually equivalent to non-terminating expressions (of the same type).

5.1 Reduction and Types

We have to show that all (normal-order) reductions keep the type of the expression (or generalize it), which will show that well-typed terms do not lead to a dynamic type error. We will explain the interaction of the rules with the type system, where the rule (llet-e) may generalize and (cp) may instantiate types during reduction. The type of the redex does not change, and the binding structure of types remains intact, i.e. expressions remain well-scoped.

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

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

(cp-e) $(\texttt{letrec}\ x = v^{\mathsf{sub}}, y = C[x^{\mathsf{vis}}], Env\ \texttt{in}\ r)$
$\to (\texttt{letrec}\ x = v, y = C[v], Env\ \texttt{in}\ r)$
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) $C[(\texttt{case}\ (c\ t_1 \dots t_n)^{\mathsf{sub}}\ \texttt{of}\ \dots((c\ y_1 \dots y_n)\ \texttt{->}\ s)\dots)]$
$\to C[(\texttt{letrec}\ y_1 = t_1, \dots, y_n = t_n\ \texttt{in}\ s)]$

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

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

(ambl) $C[(\texttt{amb}\ v^{\mathsf{sub}}\ s)] \to C[v]$ if v is a value

(ambr) $C[(\texttt{amb}\ s\ v^{\mathsf{sub}})] \to C[v]$ 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) $C[((\texttt{letrec}\ Env\ \texttt{in}\ s)^{\mathsf{sub}}\ t)] \to C[((\texttt{letrec}\ Env\ \texttt{in}\ (s\ t))]$

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

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

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

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

Fig. 4. Normal-order rules

Example 5.1. The expression

$$t := \texttt{letrec}\ x = (\texttt{letrec}\ y = \lambda u.u\ \texttt{in}\ \lambda z.\texttt{amb}\ (y\ z)\ z) :: (\forall a.a \to a)\ \texttt{in}\ x\ \texttt{True} : (x\ \texttt{Nil} : \texttt{Nil})$$

requires a generalization after (llet-e): The type labels are as follows $y = \lambda u.u$ and $\lambda z.\texttt{amb}\ (y\ z)\ z$ have type $a \to a$; the type of u and z is a. Using a (llet-e)-rule results in $(\texttt{letrec}\ x = \lambda z.(\texttt{amb}\ (y\ z)\ z), y = \lambda u.u\ \texttt{in}\ x\ \texttt{True} : (x\ \texttt{Nil} : \texttt{Nil}))$. Now it would be incorrect to use $x :: \forall a.a \to a$, and $y :: a$, since then the scoping is violated. It is also not correct to simply omit the quantifier, since then the two applications of x in the in-term cannot be correctly typed. The correct type-modification (as in Definition 5.2) is to distribute the $\forall$-quantifier to the bindings, making the different occurrences of the type variable a independent. The new types are: $y :: \forall b.b \to b$, $\lambda u.u$ has to be type-renamed to $\lambda u'.u'$ with $u' :: b$ and y in $\lambda z.(\texttt{amb}\ (y\ z)\ z)$ is labeled with type $a \to a$. The consistency constraints are satisfied after the reduction.

Definition 5.2 (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:*

$$\texttt{letrec}\ Env_1, x = (\texttt{letrec}\ x_1 = t_1 :: T_1, \ldots, x_n = t_n :: T_n\ \texttt{in}\ s) :: \forall \mathcal{X}.T\ \texttt{in}\ t \\ \rightarrow \texttt{letrec}\ Env_1, x = s' :: \forall \mathcal{X}.T, x'_1 = t'_1 :: T'_1, \ldots, x'_n = t'_n :: T'_n\ \texttt{in}\ t$$

where the following must hold. $T'_i := GenType_W(t_i, T_i)$ *for* $i = 1, \ldots, n$ *and* $W = \{x_1, \ldots, x_n\}$*. The variables* x'_i *are fresh ones with built-in type* T'_i*, i.e. the type label of the binding is changed; and the substitution* $\delta := [x'_i/x_i]$ *has been applied such that* $s' = \delta(s)$*,* $t'_i := \delta(t_i)$ *for* $i = 1, \ldots, n$*. After the type generalization, a type variable renaming has to be done for the terms in the bindings, which can be justified, since the (global) types of* `amb`*,* `seq`*, and the constructors are closed quantified types.*

Example 5.3. We give an extended example for the quantifier distribution.

$$\begin{array}{lll} \texttt{letrec}\ x = (\ \texttt{letrec}\ y_1 = \texttt{amb}\ (\lambda u.u)\ y_2; & & :: a \rightarrow a \\ \qquad\qquad\qquad\qquad\ y_2 = \texttt{amb}\ (\lambda u', u')\ y_1 & & :: a \rightarrow a \\ \qquad\qquad\ \texttt{in} \quad \lambda z.\texttt{amb}\ (y_1\ z)\ (\texttt{amb}(y_2\ z)\ z)) & & :: \forall a.a \rightarrow a \\ \texttt{in} \quad [x\ \texttt{True}, x\ \texttt{Nil}] & & \end{array}$$

Further types not shown are $u, z, u' : a$, $\texttt{amb} : (a \rightarrow a) \rightarrow (a \rightarrow\ a) \rightarrow a \rightarrow a$. We show some intermediate steps in applying (llet-e) and the quantifier-distribution:

$$\begin{array}{lll} \texttt{letrec}\ x\ = \lambda z.\texttt{amb}\ (y_1\ z)\ (\texttt{amb}(y_2\ z)\ z)) & & :: \forall a.a \rightarrow a \\ \qquad\ y_1 = \texttt{amb}\ (\lambda u.u)\ y_2; & & :: \forall a.a \rightarrow a \\ \qquad\ y_2 = \texttt{amb}\ (\lambda u', u')\ y_1 & & :: \forall a.a \rightarrow a \\ \texttt{in} \quad [x\ \texttt{True}, x\ \texttt{Nil}] & & \end{array}$$

Now the type a can be renamed in the bindings of y_1, y_2. This is permitted, since the type variable a is not a free type variable of a free occurrence of a variable: There are occurrences of `amb`, and the abstraction $\lambda u.u$, where the type can be renamed, either due to an instantiation from the general type of `amb`, or since the u in $\lambda u.u$ is bound and its type can be renamed. Due to our syntax, type renamings of variables mean also to rename the variable, but this is no problem:

$$\begin{array}{lll} \texttt{letrec}\ x\ = \lambda z.\texttt{amb}\ (y_1\ z)\ (\texttt{amb}(y_2\ z)\ z)) & & :: \forall a.a \rightarrow a \\ \qquad\ y_1 = \texttt{amb}\ (\lambda u.u)\ y_2; & & :: \forall a'.a' \rightarrow a' \\ \qquad\ y_2 = \texttt{amb}\ (\lambda u', u')\ y_1 & & :: \forall a''.a'' \rightarrow a'' \\ \texttt{in} \quad [x\ \texttt{True}, x\ \texttt{Nil}] & & \end{array}$$

E.g. the `amb` in the expressions for y_2 has now type $\texttt{amb} : (a'' \rightarrow a'') \rightarrow (a'' \rightarrow\ a'') \rightarrow a'' \rightarrow a''$.

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

Definition 5.4 (Type Instantiation). *The (cp-in)-rule with type-instantiation is:* $(\texttt{letrec}\ x = v :: T, Env\ \texttt{in}\ C[x :: S]) \to (\texttt{letrec}\ x = v :: T, Env\ \texttt{in}\ C[\rho(v) :: S])$*, where ρ is the type-substitution with $\rho(T) = S$, and where $\rho(v)$ means the expression v, such that the instantiation ρ is applied 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 5.5. 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:

$$\texttt{letrec}\ concatMap = \lambda f, xs.\texttt{case}\ldots, id = \lambda x.x, \ldots\ \texttt{in}\ (concatMap\ id)$$

A consistent typing of the subexpression (*concatMap id*) will be $[[c]] \to [c]$, where *concatMap* is typed $([c] \to [c]) \to [[c]] \to [c]$, and *id* as $[c] \to [c]$. 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

$$\begin{array}{l}\texttt{letrec}\ concatMap = \lambda f, xs.\texttt{case}\ldots, id = \lambda x.x, \ldots \\ \texttt{in}\ \texttt{letrec}\ f' = id\ \texttt{in}\ (\lambda xs'.\texttt{case}\ldots)\end{array}$$

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

Theorem 5.6. *If $s \xrightarrow{no} t$ and $s :: S$ is well-typed, then $t :: S'$ is also well-typed, and the type S is an instance of type S'.*

Proof. (Sketch) We inspect the effect of all normal-order reduction rules on the type labeling, where the type modification for (llet) and (cp) has to be taken into account. It can be checked that the type as computed by *nltype* is not changed by the reduction rules. We have to distinguish the cases where the redex is a non-let-position, where the type is not changed and exactly equal to *nltype*, let-positions, where the quantified type is kept, and the top position, where the type is either kept or generalized. (llet)-rules and (cp)-rules require special attention: For the (llet-e)-rule, we already argued that the type scope is correctly modified. The effect of the (cp)-rules is a type-instantiation of a value, which leaves the subexpression well-typed, such that the type constraints are satisfied after the application. In summary, we can show that the reduct is well-typed, and that the top expression may have the same or a more general type after a normal-order reduction step.

6 Typed Contextual Equality

The main advantage of the type labeling shows up in this section: the definitions of contextual equivalence can be done in the style of untyped lambda-calculi, and the relation between typed and untyped definitions is rather intuitive.

Definition 6.1. *Let t be a (possibly open) well-typed L_{PLC}-expression. A normal order reduction sequence of t is called an* (normal-order) evaluation *if the last term is a WHNF.*
We write $t\downarrow$ (may-convergence) iff there is an evaluation starting from t. Otherwise, if there is no evaluation of t, we write $t\Uparrow$. An expression t is called must-convergent ($t\Downarrow$), iff $t \xrightarrow{no,*} t'$ implies $t'\downarrow$. The negation of must-convergence is called may-divergence and wirtten as $t\uparrow$.

Definition 6.2 (Contextual Preorder and Equivalence). *Let T be a type and let s, t be well-typed expressions (with type labeling) of type T. Then the equivalence $\sim_T$ w.r.t may- and must-convergence is defined as follows:*

$$\begin{array}{lll} s \leq_{T,\downarrow} t & \text{iff} & \forall C[\cdot :: T]: \quad C[s]\downarrow \Rightarrow C[t]\downarrow \\ s \leq_{T,\Downarrow} t & \text{iff} & \forall C[\cdot :: T]: \quad C[s]\Downarrow \Rightarrow \ C[t]\Downarrow \\ s \leq_T t & \text{iff} & s \leq_{T,\downarrow} t \wedge s \leq_{T,\Downarrow} t \\ s \sim_T t & \text{iff} & s \leq_T t \wedge t \leq_T s \end{array}$$

It is straightforward to show that the relations $\leq_{T,\downarrow}$, $\leq_{T,\Downarrow}$, $\leq_T$ are precongruences, and that the relation $\sim_T$ is a congruence.
The advantage of our formulation is that the context-lemmas for may- and must-convergence hold, where the proof is an adaptation from [SSS08,Sab08]. Let the extended reduction contexts R^X be reduction contexts or contexts ($\texttt{letrec}\ x = (\texttt{letrec}\ y = [\cdot], Env_1\ \texttt{in}\ e)^{\mathsf{sub}}, Env_2\ \texttt{in}\ e'$) (after execution of the label-shift algorithm).

Definition 6.3. *For well-typed expressions $s, t :: T$, the relations $\leq_{R^X,T,\downarrow}$ t holds iff:*

1. *$FV(s) = FV(t)$ and for all $x \in FV(s) : FTTV(s, x) = FTTV(t, x)$.*
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: $\forall R^X[\cdot :: \rho(T)]$: If $ws(R^X[\rho(s)])$ then $(R^X[\rho(s)]\downarrow \implies R^X[\rho(t)]\downarrow)$)*
3. *for all δ where δ replaces variables with fresh variables where the built-in type of the variables may be generalized, and δ may rename type-variables,it holds: $\forall R^X[\cdot :: \delta(T)]$: If $ws(R^X[\delta(s)])$ then $(R^X[\delta(s)]\downarrow \implies R^X[\delta(t)]\downarrow)$)*

The relations $\leq_{R^X,T,\uparrow}$, $\leq_{R^X,T,\Downarrow}$ are defined accordingly where $\downarrow$ is replaced by $\uparrow$ or by $\Downarrow$.

Note that condition (1) implies that $ws(C[s]) \iff ws(C[t])$ for all $C[. :: T]$.
For the proofs that show the following context lemma, see Appendix A.

Lemma 6.4 (Context Lemma). *The following holds (see appendix A):*
$\leq_{R^X,T,\downarrow} \subseteq \leq_{T,\downarrow}$ *and*
$\leq_{R^X,T,\downarrow} \cap \leq_{R^X,T,\Downarrow} \subseteq \leq_T$.

6.1 Relation to the Call-by-Need Amb-Calculus

In this section we explain the relation of the contextual equivalence of our typed calculus to the calculus $\Lambda^{\mathtt{let}}_{\mathtt{amb}}$ (see [Sab08]).
Using the methods in [SSNSS08] on translations, we are able to show that all untyped equivalences, under some well-scoping restrictions, are also typed equivalences. First we repeat the notions in [SSNSS08], which are concerned with translations *enc* from one calculus into another. A translation *enc* is *compositional* iff $enc(C[e]) = enc(C)[enc(e)]$ for all $C[\cdot :: T]$ and $e :: T$. The translation *enc* is *convergence equivalent*, iff $(enc(e)\downarrow \iff e\downarrow) \wedge (enc(e)\Downarrow \iff e\Downarrow)$ for all e. A translation *enc* is *adequate* iff $enc(e_1) \leq_{enc(T)} enc(e_2) \implies e_1 \leq_T e_2$ for all e_1, e_2 of type T. A translation *enc* is *fully abstract* iff $enc(e_1) \leq_{enc(T)} enc(e_2) \iff e_1 \leq_T e_2$ for all e_1, e_2 of type T. The results in [SSNSS08] show that compositionality and convergence equivalence imply adequacy. Let ${L_{PLC}}^{\mathtt{untyped}}$ be the language as above, but without types: The only weakly typed construct is case. The normal-order reduction is as in figure 4. We define the following two translations η and δ: $L_{PLC} \xrightarrow{\eta} {L_{PLC}}^{\mathtt{untyped}} \xrightarrow{\delta} \Lambda^{\mathtt{let}}_{\mathtt{amb}}$, which are the identity on expressions. Note that the normal-order reductions in L_{PLC} and ${L_{PLC}}^{\mathtt{untyped}}$ are identical, however, the normal-order reductions in $\Lambda^{\mathtt{let}}_{\mathtt{amb}}$ are different.

Lemma 6.5. *(see appendix B) The translation δ and its inverse are compositional and convergence equivalent, and hence the translation δ is fully abstract.*

Lemma 6.6. *The translation η is compositional and convergence equivalent.*

Proof. Compositionality is obvious. For a typed L_{PLC}-term t, the normal-order reductions are exactly the same as the untyped normal-order reductions of the untyped expression $\eta(t)$ and also the weak head normal forms are the same. Hence, η is convergence equivalent.

Corollary 6.7. *The translation η is adequate, but not fully abstract.*

Proof. Adequacy follows from general results in [SSNSS08]. The translation is not fully abstract, since the program transformation (caseIdB) is correct in L_{PLC} (see Proposition 7.3), but cannot be correct in ${L_{PLC}}^{\mathtt{untyped}}$, since $(\mathtt{case}_{\mathtt{Bool}}\ \mathtt{Nil\ of\ True\ ->\ True;False\ ->\ False})\Uparrow$, and thus is clearly not equivalent to `Nil`.

This corollary now permits to transfer all equivalences from the untyped language $\Lambda^{\mathtt{let}}_{\mathtt{amb}}$ to the typed language L_{PLC}. The following theorem states this result for a selection of the program transformations shown correct in [Sab08].

Theorem 6.8. *Let $s \approx t$ be a deterministic reduction rule, i.e. a rule shown in figure 4 (ignoring the labels) but not (amb-l) and (amb-r), or a program transformation in figure 5. Let D be a context such that $D[s], D[t]$ are well-typed and condition* (1) *of Definition 6.3 holds for $D[s], D[t]$. Then $D[s] \to D[t]$ is a correct program transformation.*

Note that the condition (1) of Definition 6.3 is only necessary for the rules/transformations (case), (seq), (gc), and (ucp), since the occurrences of free variables might be modified.

(gc)	$(\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ t) \to t$ if no x_i does occur 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 does occur free in t nor in any t_j
(ucp)	$(\texttt{letrec}\ x = s, Env\ \texttt{in}\ S[x]) \to (\texttt{letrec}\ Env\ \texttt{in}\ S[s])$ if S is a surface context[1] and x does not occur in s, S and Env.
(ucp)	$(\texttt{letrec}\ x = s, y = S[x], Env\ \texttt{in}\ t) \to (\texttt{letrec}\ y = S[s], Env\ \texttt{in}\ t)$ if S is a surface context and x does not occur in s, S, Env and t.
(ucp)	$(\texttt{letrec}\ x = s\ \texttt{in}\ S[x]) \to S[s]$ if S is a surface context and x does not occur in s and S.
(ambid)	$\texttt{amb}\ v\ v \to v$ if v is a value
(ambcom)	$\texttt{amb}\ s\ t \to \texttt{amb}\ t\ s$
(ambassoc)	$\texttt{amb}\ v_1\ (\texttt{amb}\ v_2\ v_3) \to \texttt{amb}\ (\texttt{amb}\ v_1\ v_2)\ v_3$ if v_1, v_2, v_3 are closed values
(ambomega)	$\texttt{amb}\ s\ \Omega \to s$ if Ω is a closed and must-divergent

Fig. 5. Further program transformations

7 A Type-Dependent Program Transformation

Example 7.1. The following two functions `and1`, `and2` are wrong in the untyped calculus, but are contextually equivalent in the polymorphically typed calculus.

```
and1 = \ x y -> case_Bool x of True -> y; False -> False
and2 = \ x y -> case_Bool x of
    True -> (case_Bool y of True -> True, False -> False);
    False -> False
```

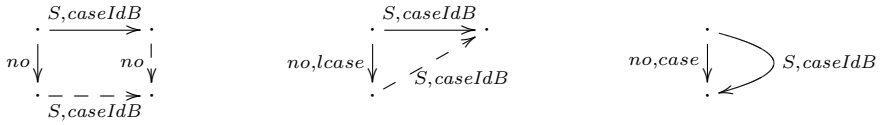
In an untyped calculus, the expressions `and1` and `and2` are different, since `and1 True []` reduces to `[]`, whereas `and2 True []` is not convergent.
In the typed calculus, where we insist on the typing `and1:: Bool -> Bool -> Bool`, the equality holds, which can be derived from the correctness of the transformation (caseIdB) shown below.

We will show that the following transformation is correct:

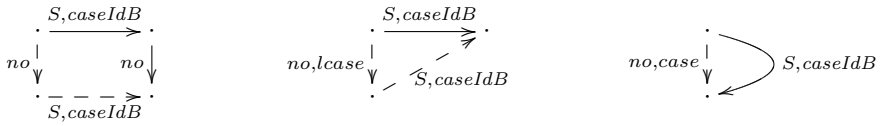
(caseIdB) $\texttt{case}_{\texttt{Bool}}\ s\ \texttt{of}\ (\texttt{True} \texttt{->} \texttt{True})\ (\texttt{False} \texttt{->} \texttt{False}) \to (s :: \texttt{Bool})$.

This rule is the heart also of other type-dependent transformations and could also be generalized to other data types. Note that condition (1) of Definition 6.3

holds for (caseIdB). A complete set of forking diagrams, i.e. overlappings of a normal-order reduction with a (caseIdB)-transformation within a surface-context are as follows:



A complete set of commuting diagrams, i.e. overlappings of a normal-order reduction with a (caseIdB)-transformation within a surface-context is as follows.



Lemma 7.2. *Let $C[t] \xrightarrow{caseIdB} C[t']$. If t is a WHNF, then t' is a WHNF, and if t' is a WHNF and t' is not a WHNF, then $t \xrightarrow{no} t''$, where t'' is a WHNF.*

Proof. The only nontrivial case is that $C[\texttt{case}\ t\ \texttt{of}\ \text{alts}] \xrightarrow{\text{caseIdB}} C[t]$, and $C[t]$ is a WHNF. Due to typing $t \in \{\texttt{True}, \texttt{False}\}$, and C is a reduction context. In this case, $\xrightarrow{\text{caseIdB}}$ is the same as a $\xrightarrow{no,case}$-reduction.

Proposition 7.3. *If $t :: T \xrightarrow{caseIdB} t' :: T$, then $t \sim_T t'$.*

Proof. This follows from the context lemmas 6.4 and from the diagrams by induction on the length of a normal-order reduction, where for the commuting diagrams one has to observe that $\xrightarrow{lcase}$ cannot be applied infinitely often. The part for must-convergence is done using the same methods as in [SSS07a]. □

Note that some theorems valid in the untyped calculus [Sab08,SSS07a] require more arguments for showing them also in the typed calculus, like $s \sim t \iff \forall C : C[s]{\Downarrow} \Leftrightarrow C[t]{\Downarrow}$, since the set of contexts is different in both calculi.

Example 7.4. Consider the following functions that are a part of modelling sequential circuits, borrowed from [SSS07b]. We assume that there are defined (logical) functions like not, parallel-or, and parallel-and.

$$\begin{aligned}&\texttt{not2} = \lambda b.\texttt{case}\ b\ \texttt{of}\ (\texttt{True}\ \texttt{->}\ \texttt{False})\ (\texttt{False}\ \texttt{->}\ \texttt{True})\\ &\texttt{not}\ \ = \lambda xs.(\texttt{map}\ \texttt{not2}\ xs)\end{aligned}$$

In the typed version, we assume that $\texttt{not2} :: \texttt{Bool} \to \texttt{Bool}$, and $\texttt{not} :: [\texttt{Bool}] \to [\texttt{Bool}]$. We are sure that the equation $\texttt{not} \,.\, \texttt{not} \sim_{[\texttt{Bool}]\to[\texttt{Bool}]} \texttt{id}$ holds and can

[1] (surface-context: the hole is not in an abstraction; see [SSS08,Sab08] for more information)

be proved, using a co-inductive argument. However, the equation is incorrect in the untyped calculus, since e.g. it is false for an untyped input argument, say `True`, which is not a list.

8 Discussing Extensions

Type classes could be added to our type system by using so-called order-sorted (meta-) typing of the type-variables, where this meta-typing are the type classes. A further adaptation is that the quantifiers may be restricted to type classes and that type-substitutions have to respect the type-class structure. For the full accommodation of Haskell's dictionary, there is a built-in data structure necessary that does not influence the treatment of behavioral equivalence since the dictionary is visible, perhaps as binding in a letrec (see also [NS91].
Another possibility is to use the translation in [WB89] of Haskell's type classes into a polymorphically typed language and to define the equivalence w.r.t. the translated language.
However, it appears to be not possible to have the type classes with subclasses, instances and type-dependent definitions of the functions that are the methods of the type classes, and in addition a polymorphic use of the methods, since this would require a run-time decision depending on the types.
The implementation of type classes using dictionaries as in Haskell enforces a so-called monomorphism restriction. An example from the manual is as follows:

```
genericLength :: Num a => [b] -> a
let { len =  genericLength xs } in (len, len)
```

The Haskell type checker produces the type

```
(\xs -> let { len = genericLength xs } in (len, len))
           :: (Num i) => [b] -> (i, i)
```

due to the monomorphism restriction. The reason is the avoidance of potential double evaluation, which may be introduced by the type class transformation as follows: A more general type is `(Num i, Num j) => [b] -> (i, j)`, which would result in the following translation by using F2-polymorphism, where t is the type (dictionary) parameter.

```
f xs x y   =   let { len = \t ->genericLength xs t } in (len x, len y)
```

An evaluation would copy the abstraction and hence evaluate the length twice. The *reason* is that an application is turned into an abstraction and hence made copyable by the translation. The translation as proposed above using the subclass-ordering would not allow this copying, and hence is safe. If this extension is done without care, then adding non-determinism to Haskell would lead to severe conflicts due to compiler-induced destruction of sharing.

It appears possible to extend the typing by so-called existential types. Technically, in data terms, i.e. constructor expressions, we have to permit local type variables that can be seen as existentially quantified. We leave a detailed construction and analysis of this extension for further research.

9 Discussion of Design Decisions of the Label-Type System

This section is intended to discuss and justify some decisions that we made in designing the labeled type system. The polymorphic type system for the language L_{PLC} should be close to Haskell, and the types of constructors, the data types the letrec syntax match this condition.
Issues, which were unclear and had to be resolved:

1. Why type labels at subexpressions?
2. Should we permit that different occurrences of the same let-bound variable are differently typed?
3. Should we permit type-quantifiers only at let-positions or everywhere?
4. What is the impact of permitting type instances of the "true type" instead of the type itself?

(1): One of the problems is that a type derivation system which only answers: "yes, the expression is typeable with type ....", does not really help for a safe application of program transformations, since the types of subexpressions are left open. E.g. the transformation $C[(\texttt{case}_{\texttt{Bool}}\ (s :: \texttt{Bool})\ \texttt{of True -> True; False -> False})] \longrightarrow C[(s :: \texttt{Bool})]$ makes only sense, if after the application, i.e. in the expression $C[(s :: \texttt{Bool})]$ the type derivation system does not assign another type to s. Usually, this is not prevented by type systems, and so it may be a hard task (perhaps impossible in general) to prove for typed system that the typed program transformations are really safe.
(2),(3) The first version of the labeled type system had the property that variables and all their occurrences are only allowed with one type. This is only useful, if also type quantifiers are allowed everywhere. However, it soon became clear that the longer reduction sequences had a tendency to destroy the type. For example, the (lbeta)-reduction required several precautions and a manipulation of all the types in the body of the applied abstraction. (This is also in the current version as instantiation after (cp)). A further problem was that the generalization of types after the llet-e-reductions, as it is also done in the current system, was insufficient: the more general type of variables had to be inherited to all their positions in subexpressions, and moreover, the impact of the type generalization to the subexpressions that contain the variable had to be computed. This, however, is close to using the type-checker for the whole expression.
The third author implemented this type system [Har08], which worked perfectly for most expressions, however produced several counterexamples to our (previous) simplistic assumptions.

In the current system, this global type update is prevented by the possibility to allow type labels that are instances of the "true type". It is interesting to note that the true type is not necessary for the evaluation.

10 Conclusion

We have developed a type-labeling of expressions in a polymorphically typed λ-calculus with `letrec` and `amb` that demonstrates how to integrate polymorphic typing and contextual equivalence in a non-deterministic lambda-calculus with letrec. We are convinced that the methods can be applied to larger classes of extensions of lambda-calculi by polymorphic type systems. We conjecture that an extension to Haskell's type classes ([WB89]) and other type extensions is possible.

References

AK97. Zena M. Ariola and Jan Willem Klop. Lambda calculus with explicit recursion. *Inform. and Comput.*, 139(2):154–233, 1997.

CHS05. Arnaud Carayol, Daniel Hirschkoff, and Davide Sangiorgi. On the representation of McCarthy's amb in the pi-calculus. *Theoret. Comput. Sci.*, 330(3):439–473, 2005.

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

Har08. Frederik Harwath. Prototypische Validierung eines parametrischen polymorphen Typsystems für eine Kernsprache von Haskell, 2008. Thesis, in German.

KTU93. A. J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. The undecidability of the semi-unification problem. *Inform. and Comput.*, 102(1):83–1018, 1993.

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.

NS91. Tobias Nipokw and Gregor Snelting. Type classes and overloading resolution via order-sorted unification. In *5th ACM Conf FPCA 1991*, volume 523 of *LNCS*, pages 1–14. Springer-Verlag, 1991.

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.

RV07. Arend Rensink and Walter Vogler. Fair testing. *Inform. and Comput.*, 205(2):125–198, 2007.

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.

SSNSS08. Manfred Schmidt-Schauß, Joachim Niehren, Jan Schwinghammer, and David Sabel. Adequacy of compositional translations for observational semantics. In *5th IFIP TCS 2008*, volume 273 of *IFIP*, pages 521–535. Springer, 2008.

SSS07a. Manfred Schmidt-Schauß and David Sabel. On generic context lemmas for lambda calculi with sharing. Frank report 27, Inst. f. Informatik, J.W.Goethe-University, Frankfurt, 2007. submitted for publication.

SSS07b. Manfred Schmidt-Schauß and David Sabel. Program transformation for functional circuit descriptions. Frank report 30, Inst. f. Informatik, Goethe-Univ., Frankfurt, 2007.

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.

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

WB89. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL 1989*, pages 60–76. ACM Press, 1989.

A Context Lemmas

For an expression s let $\mathrm{CON}(s)$ be the set of all reduction sequences for s ending in a WHNF, and let $\mathrm{DIV}(s)$ be the set of all reduction sequences for s ending in a must-divergent expression. For a reduction sequence RED the function $\mathrm{rl}(RED)$ computes the length of the reduction sequence RED.
The contexts R^X can be defined syntactically with the following grammar:

$$
\begin{array}{l}
R^- ::= [\cdot] \mid (R^-\ s) \mid (\texttt{amb}\ R^-\ s) \mid (\texttt{amb}\ R^-\ s) \mid (\texttt{seq}\ R^-\ s) \mid (\texttt{case}\ R^-\ alts) \\
R^X ::= (\texttt{letrec}\ Env\ \texttt{in}\ R^-) \\
\qquad \mid (\texttt{letrec}\ Env, y_0 = R^-, y_1 = R_1^-[y_0], \ldots, y_m = R_m^-[y_{m-1}]\ \texttt{in}\ R_{m+1}^-[y_m]) \\
\qquad\quad \text{where } R_i^- \neq [\cdot] \\
\qquad \mid (\texttt{letrec}\ Env,\ y_0 = (\texttt{letrec}\ y = [\cdot], Env_2 \texttt{in}\ s), \\
\qquad\qquad\qquad\qquad y_1 = R_1^-[y_0], \ldots, y_m = R_m^-[y_{m-1}] \\
\qquad\quad \texttt{in}\ R_{m+1}^-[y_m]) \\
\qquad\quad \text{where } R_i^- \neq [\cdot]
\end{array}
$$

Note that the contexts R^X are either reduction contexts or contexts of the form $(\texttt{letrec}\ x = ((\texttt{letrec}\ y = [\cdot], Env_1\ \texttt{in}\ e)^{\mathsf{sub}}, Env_2\ \texttt{in}\ e')$ after running the label shift. 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 A.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 an R^X-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 an R^X-context.*

Proof. We assume there is a multicontext C with n holes and there are terms $s_1, \ldots, s_n$ with $R_i^X \equiv C[s_1, \ldots, s_{i-1}, \cdot_i :: T_i, s_{i+1}, \ldots, s_n]$ being an R^X-context. We distinguish two cases:

- R_i^X is 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.
- R_i^X is of the form $\texttt{letrec}\ x = (\texttt{letrec}\ y = [\cdot_i].Env_2\ \texttt{in}\ t)^{\mathsf{sub}}, Env\ \texttt{in}\ t$, then the context $\texttt{letrec}\ x = [\cdot], Env\ \texttt{in}\ t$, is a reduction context. Let C' be the context corresponding to C where the subterm $(\texttt{letrec}\ y = [\cdot_i].Env_2\ \texttt{in}\ t)$ (perhaps including some other context holes) is replaced by a new context hole. Then the labeling algorithm applied to C' either labels the new hole, or some other hole $\cdot_j$. If it stops at the new hole, then $C[t_1, \ldots, t_{i-1}, \cdot_i, t_{i+1}, \ldots, t_n]$ is an R^X-context for all terms $t_1, \ldots, t_n$. If an earlier hole $\cdot_j$ is labeled, then $C[t_1, \ldots, t_{j-1}, \cdot_j, t_{j+1}, \ldots, t_n]$ is a reduction contexts for all terms $t_1, \ldots, t_n$.

Lemma A.2. *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 $\forall C : ws(C[s]) \iff ws(C[t])$*

Proof. We only show $\forall C : ws(C[s]) \implies ws(C[t])$, since the other direction is symmetric.
Assume the claim is false, i.e. there exists a context C such that $ws(C[s])$ holds, but $C[t]$ is not well-scoped. Then there is a subterm t' of $C[t]$ of type $\forall \mathcal{X}.T'$ with $\mathcal{X} \cap FTFV(t') = \{a_1, \ldots, a_n\} \neq \emptyset$. The subterm t' must be a proper superterm of t in $C[t]$, since t is well-scoped, i.e. $C[t] = C_1[C_2[t]]$ where $t' = C_2[t]$ and $C_1[\cdot : \forall \mathcal{X}.T]$. Furthermore, $\{a_1, \ldots, a_n\}$ must also be a subset of $FTFV(t)$, since otherwise $C[s]$ cannot be well-scoped and there must exist a free variable $x \in FV(t) \cap FV(t')$ with $FTTV(t,x) \cap \{a_1, \ldots, a_n\} = X \neq \emptyset$. From the precondition we have that $FTTV(s,x) \cap \{a_1, \ldots, a_n\} = FTTV(t,x) \cap \{a_1, \ldots, a_n\} = X$. Now x must be a free variable of $C_2[s]$, and thus $C_1[C_2[s]]$ cannot be well-scoped. Hence we have a contradiction.

Lemma A.3. *Let $n \geq 0$ such that for $i = 1, \ldots, n$ let $s_i, t_i : T_i$ be well-typed expressions, such that for all i: $FV(s_i) = FV(t_i)$ and for all $x \in FV(s_i) : FTTV(s_i, x)) = FTTV(t_i, x)$. Then $\forall C[\cdot_1 :: T_1, \ldots : \cdot_n :: T_n] : ws(C[s_1, \ldots, s_n]) \iff ws(C[t_1, \ldots, t_n])$.*

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 $C[s_1, \ldots, s_n]$ is well-scoped. Lemma A.2 implies $ws(C[t_1, s_2, \ldots, s_n])$. Applying the induction hypothesis to the multicontext $C' = C[t_1, \cdot_2, \ldots, \cdot_n]$ shows the claim.

Lemma A.4. *Let $s \leq_{R^X,T,\downarrow} t$ ($s \leq_{R^X,T,\uparrow} t$, resp.).*

- *For every instantiation ρ according to Definition 5.4, it holds $\rho(s) \leq_{R^X,T,\downarrow} \rho(t)$ ($\rho(s) \leq_{R^X,T,\uparrow} \rho(t)$, resp.).*
- *For every substitution δ to Definition 5.2, it holds $\delta(s) \leq_{R^X,T,\downarrow} \delta(t)$ ($\delta(s) \leq_{R^X,T,\uparrow} \delta(t)$, resp.).*

Proof. It is easy to verify that the first two conditions of Definition 6.3 also hold for $\rho(s), \rho(t)$ and $\delta(s), \delta(t)$, since identical types are identically renamed or substituted and since the free type variables of free variables as well as the free variables of s and t are identical by the second property of Definition 6.3. Now, the remaining parts follow from the Definition of $\leq_{R^X,T,\downarrow}$ ($\leq_{R^X,T,\uparrow}$, resp.)

Lemma A.5 (Context Lemma for May-Convergence). $\leq_{R^X,T,\downarrow} \subseteq \leq_{T,\downarrow}$.

Proof. We prove a 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 $i = 1, \ldots, n$: $s_i \leq_{R^X,T,\downarrow} t_i$, then $ws(C[s_1, \ldots, s_n])$ implies $ws(C[t_1, \ldots, t_n]$ and $C[s_1, \ldots, s_n]\downarrow \Rightarrow C[t_1, \ldots, t_n]\downarrow$.
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 shortest reduction sequence in $\mathrm{CON}(C[s_1, \ldots, s_n])$.
- n is the number of holes in C.

We assume that the pairs are ordered lexicographically, thus this measure is well-founded. The claim holds for $n = 0$, i.e., all pairs $(l, 0)$, since if C has no holes there is nothing to show.
Now let $(l, n) > (0, 0)$. For the induction step we assume that the claim holds for all n', C', s'_i, t'_i, $i = 1, \ldots, n'$ with $(l', n') < (l, n)$. Let us assume that the precondition holds, i.e., that $\forall i : s_i \leq_{R^X,T,\downarrow} t_i$. Let C be a multi-context with $ws(C[s_1, \ldots, s_n])$ and RED be a shortest reduction sequence in $\mathrm{CON}(C[s_1, \ldots, s_n])$ with $\mathrm{rl}(RED) = l$. Lemma A.3 implies $ws(C[t_1, \ldots, t_n])$. For proving $C[t_1, \ldots, t_n]\downarrow$, 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 an R^X-context. Lemma A.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 R^X-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 $RED \in \mathrm{CON}(C_1[s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n])$. Since C_1 has $n - 1$ holes, we can use the induction hypothesis and derive $C_1[t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n]\ \downarrow$, i.e. $C[t_1, \ldots, t_{i-1}, s_i, t_{i+1}, \ldots, t_n]\downarrow$. This implies $R_2[s_i]\downarrow$. Using the precondition we derive $R_2[t_i]\downarrow$, i.e. $C[t_1, \ldots, t_n]\downarrow$.
- 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 an R^X-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 $C[t_1, \ldots, t_n]\downarrow$. 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 quantifier distribution due to a (llet-e)-reduction, or 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 $s'_i \leq_{T',\downarrow,R^X} t'_i$. 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`- or `amb`-expression that is discarded by a (seq)- or (amb)-reduction, then s_i and t_i are both eliminated.
  - If the normal order reduction is a (llet-e)-reduction, then s_i cannot be the right hand side of `letrec`-binding which is generalized by the quantifier distribution, since then replacing s_i by the hole would result in an R^X-context. But s_i can be a proper subterm of such a binding, then a

substitution δ according to Definition 5.2 is applied to s_i resulting in s'_i. W.l.o.g. we can assume that the same substitution is applied to t_i resulting in t'_i. Lemma A.4 and the precondition imply that $s'_i \leq_{\delta(T),\downarrow,R^X} t'_i$ must hold.

- If the normal order 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 nor affected by a type generalization as mentioned in the previous items, then s_i and t_i can only change their respective position.
- If the normal order 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-substitutions according to Definition 5.4. W.l.o.g. for all i: $\rho_{s,i} = \rho_{t,i}$. Lemma A.4 and the precondition show $\rho_{s,i}(s_i) \leq_{\rho_{s,i}(T),\downarrow,R^X} \rho_{s,i}(t_i)$.

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 $C'[t'_1,\ldots,t'_m]\downarrow$. With $C[t_1,\ldots,t_n] \xrightarrow{no,a} C'[t'_1,\ldots,t'_m]$ we have $C[t_1,\ldots,t_n]\downarrow$.

Lemma A.6 (Context Lemma). $\leq_{R^X,T,\downarrow} \cap \leq_{R^X,T,\Downarrow} \subseteq \leq_T$.

Proof. Due to Lemma A.5 it is sufficient to show $\leq_{R^X,T,\downarrow} \cap \leq_{R^X,T,\Downarrow} \subseteq \leq_{T,\Downarrow}$.
We prove a more general claim using multicontexts and using the preorder on may-divegerence which is the contrapositive of the $\leq_{R^X,T,\Downarrow}$: For all $n \geq 0$ and for all multicontexts C with n holes and for all expressions $s_1,\ldots,s_n$ and $t_1,\ldots,t_n$ holds: If $t_i \leq_{R^X,T,\uparrow} s_i \wedge s_i \leq_{R^X,T,\downarrow} t_i$ then $ws(C[t_1,\ldots,t_n])$ implies $C[t_1,\ldots,t_n]\uparrow \implies C[s_1,\ldots,s_n]\uparrow$.
The proof is by induction where n, C, s_i, t_i for $i = 1,\ldots n$ are given. The induction is on the measure (l,n), where

- l is the length of a shortest reduction sequence in $\mathrm{DIV}(C[t_1,\ldots,t_n])$.
- n is the number of holes in C.

The induction is analogous to the proof of Lemma A.5. The precondition for may-convergence is necessary for the subcase that C has no holes in a reduction context and $C[t_1,\ldots,t_n]\Uparrow$.

B Deriving Equivalences from the Untyped Calculus

The encoding δ from ${L_{PLC}}^{\mathtt{untyped}}$ into $\Lambda^{\mathtt{let}}_{\mathtt{amb}}$ [Sab08] is the identity translation, since the syntax of both calculi is identical. The translation δ is obviously compositional, but not obviously convergence equivalent, since the standard reductions are different.

Lemma B.1. *If s is a WHNF for ${L_{PLC}}^{untyped}$, then s is a WHNF for $\Lambda^{\mathtt{let}}_{amb}$. If s is a WHNF for $\Lambda^{\mathtt{let}}_{amb}$, then $s \xrightarrow{no,*} t$ in ${L_{PLC}}^{untyped}$ where t is a WHNF for ${L_{PLC}}^{untyped}$.*

Proof. The difference between WHNFs in both calculi is that in $\Lambda^{\mathtt{let}}_{\mathtt{amb}}$ expressions of the form $(\mathtt{letrec}\ Env, x_1 = (c\ s_1 \dots s_n), x_2 = x_1, \dots, x_m = x_{m-1}\ \mathtt{in}\ x_m)$ are also WHNFs. Those expressions can be reduced to a WHNF in ${L_{PLC}}^{\mathtt{untyped}}$ by first copying the variables into the `in`-expression, then using (no,abs) and (no,llet) and finally copying the cv-expression.

Proposition B.2. *For all expressions s in ${L_{PLC}}^{untyped}$:*

- *If $s\downarrow_{{L_{PLC}}^{untyped}}$ then $s\downarrow_{\Lambda^{\mathtt{let}}_{amb}}$.*
- *If $s\Downarrow_{\Lambda^{\mathtt{let}}_{amb}}$ then $s\Downarrow_{{L_{PLC}}^{untyped}}$.*

Proof. This follows from the standardization theorem in [Sab08], since all reduction rules of ${L_{PLC}}^{\mathtt{untyped}}$ are either correct program transformations or (ambs)-transformations for $\Lambda^{\mathtt{let}}_{\mathtt{amb}}$.

Lemma B.3. *Let $s \xrightarrow{no} t$ in $\Lambda^{\mathtt{let}}_{amb}$. Then $s \xrightarrow{no,+} t$ in ${L_{PLC}}^{untyped}$.*

Proof. For (lbeta), (cp), (amb-c), (seq-c), (llet-e), (lamb), (lseq), (lcase), and (lapp) the reduction is either identical or a variable-to-variable chain must be shortened in ${L_{PLC}}^{\mathtt{untyped}}$. An example is $(\mathtt{letrec}\ x = (\lambda y.s)\ t, y = x, z = y\ \mathtt{in}\ z)$. In $\Lambda^{\mathtt{let}}_{\mathtt{amb}}$ the normal order reduction is an $(lbeta)$-reduction resulting in $(\mathtt{letrec}\ x = (\mathtt{letrec}\ y = t\ \mathtt{in}\ s), y = x, z = y\ \mathtt{in}\ z)$. In ${L_{PLC}}^{\mathtt{untyped}}$ two (cp)-reductions must be performed, before $(lbeta)$-is applicable, i.e.

$$\begin{array}{ll} & (\mathtt{letrec}\ x = (\lambda y.s)\ t, y = x, z = y\ \mathtt{in}\ z) \\ \xrightarrow{no,\mathrm{cp}} & (\mathtt{letrec}\ x = (\lambda y.s)\ t, y = x, z = y\ \mathtt{in}\ y) \\ \xrightarrow{no,\mathrm{cp}} & (\mathtt{letrec}\ x = (\lambda y.s)\ t, y = x, z = y\ \mathtt{in}\ x) \\ \xrightarrow{no,\mathrm{lbeta}} & (\mathtt{letrec}\ x = (\mathtt{letrec}\ y = t\ \mathtt{in}\ s), y = x, z = y\ \mathtt{in}\ z) \end{array}$$

For (llet-in) the reductions are always identical. Thus for $a \in \{(\mathrm{lbeta}), (\mathrm{amb\text{-}c}), (\mathrm{seq\text{-}c}), (\mathrm{llet}), (\mathrm{lamb}), (\mathrm{lseq}), (\mathrm{lcase}), (\mathrm{lapp})\}$ we have: If $s \xrightarrow{no,a} t$ in $\Lambda^{\mathtt{let}}_{\mathtt{amb}}$, then $s \xrightarrow{no,\mathrm{cp},*} \xrightarrow{no,a} t$ in ${L_{PLC}}^{\mathtt{untyped}}$.

Now we consider the other reductions, i.e. (seq-in), (seq-e), (amb-in), (amb-e), (case-in), and (case-e): Then again first a variable-to-variable perhaps must be shortened. If the value used by the reduction is a constructor application then an (abs)-reduction followed by a (llet-e) and a (cp)-reduction is necessary to copy the value into target position. If the value is an abstraction, then a (cp)-reduction is necessary to copy the abstraction into the target position. Finally,

a (seq), (amb) or (case) reduction is performed. An example is:

$$
\begin{array}{ll}
 & \texttt{letrec } x = c\ t_1 \dots\ t_n, y = x, z = y \\
 & \texttt{in case } z\ \dots (c\ p_1 \dots p_n \texttt{ -> } r) \dots \\
\xrightarrow{no,\text{case-in}} & \texttt{letrec } x = c\ w_1 \dots\ w_n, w_1 = t_1, \dots, w_n = t_n, y = x, z = y \\
 & \texttt{in } (\texttt{letrec } p_i = w_i \texttt{ in } r) \\
\hline
\xrightarrow{no,\text{cp},*} & \texttt{letrec } x = c\ t_1 \dots\ t_n, y = x, z = y \\
 & \texttt{in case } x\ \dots (c\ p_1 \dots p_n \texttt{ -> } r) \dots \\
\xrightarrow{no,\text{abs}} & \texttt{letrec } x = (\texttt{letrec } w_1 = t_1, \dots, w_n = t_n \texttt{ in } c\ w_1 \dots\ w_n), y = x, z = y \\
 & \texttt{in case } x\ \dots (c\ p_1 \dots p_n \texttt{ -> } r) \dots \\
\xrightarrow{no,\text{llet}} & \texttt{letrec } x = c\ w_1 \dots\ w_n, w_1 = t_1, \dots, w_n = t_n, y = x, z = y \\
 & \texttt{in case } x\ \dots (c\ p_1 \dots p_n \texttt{ -> } r) \dots \\
\xrightarrow{no,\text{cp}} & \texttt{letrec } x = c\ w_1 \dots\ w_n, w_1 = t_1, \dots, w_n = t_n, y = x, z = y \\
 & \texttt{in case } (c\ w_1 \dots\ w_n)\ \dots (c\ p_1 \dots p_n \texttt{ -> } r) \dots \\
\xrightarrow{no,\text{case}} & \texttt{letrec } x = c\ w_1 \dots\ w_n, w_1 = t_1, \dots, w_n = t_n, y = x, z = y \\
 & \texttt{in } (\texttt{letrec } p_i = w_i \texttt{ in } r)
\end{array}
$$

Thus for $a \in \{\text{seq}, \text{amb}, \text{case}\}$: If $s \xrightarrow{no,a\text{-in}} t$ or $s \xrightarrow{no,a\text{-e}} t$ in $\Lambda^{\texttt{let}}_{\texttt{amb}}$, then $s \xrightarrow{no,\text{cp},*} \xrightarrow{no,\text{abs},0\vee1} \xrightarrow{no,\text{llet},0\vee1} \xrightarrow{no,\text{cp}} \xrightarrow{no,a} t$ in ${L_{PLC}}^{\texttt{untyped}}$. (Note that we refer to the names of the reductions as in $\Lambda^{\texttt{let}}_{\texttt{amb}}$ [Sab08]).

Proposition B.4. *If $s{\downarrow_{\Lambda^{\texttt{let}}_{amb}}}$, then $s{\downarrow_{{L_{PLC}}^{untyped}}}$. If $s{\Downarrow_{{L_{PLC}}^{untyped}}}$, then $s{\Downarrow_{\Lambda^{\texttt{let}}_{amb}}}$*

Proof. The part for may-convergence follows by the previous lemma and Lemma B.1. The part for must-convergence can be shown using may-divergence, i.e. it is sufficient to show $s \uparrow_{\Lambda^{\texttt{let}}_{\texttt{amb}}} \implies s \uparrow_{{L_{PLC}}^{\texttt{untyped}}}$. This follows by induction where the induction step follows by the previous lemma. For the base case holds if $s{\Uparrow_{\Lambda^{\texttt{let}}_{\texttt{amb}}}} \implies s{\Uparrow_{{L_{PLC}}^{\texttt{untyped}}}}$, since the implication is equivalent to $s{\downarrow_{{L_{PLC}}^{\texttt{untyped}}}} \implies s{\downarrow_{\Lambda^{\texttt{let}}_{\texttt{amb}}}}$ which is proved in Proposition B.2.

Theorem B.5. *The identity translations between ${L_{PLC}}^{untyped}$ and $\Lambda^{\texttt{let}}_{amb}$ are fully-abstract.*

Corollary B.6. *All correct program transformations shown for $\Lambda^{\texttt{let}}_{amb}$ in [Sab08] are correct in ${L_{PLC}}^{untyped}$.*