Contextual Equivalences in Call-by-Need and Call-By-Name Polymorphically Typed Calculi (Preliminary Report)

Manfred Schmidt-Schauß and David Sabel

Goethe-University, Frankfurt am Main
{schauss,sabel}@ki.cs.uni-frankfurt.de

Abstract

This paper presents a call-by-need polymorphically typed lambda-calculus with letrec, case, constructors and seq. The typing of the calculus is modelled in a system-F style. Contextual equivalence is used as semantics of expressions. We also define a call-by-name variant without letrec. We adapt several tools and criteria for recognizing correct program transformations to polymorphic typing, in particular an inductive applicative simulation.

1998 ACM Subject Classification F.4.2 Grammars and Other Rewriting Systems, F.3.2 Semantics of Programming Languages, D.3.1 Formal Definitions and Theory

Keywords and phrases functional programming, polymorphic typing, contextual equivalence, semantics

Digital Object Identifier 10.4230/OASIcs.WPTE.2014.63

1 Introduction

The goal of this paper is to present theoretical tools for recognizing correct program transformation in polymorphically typed, lazy functional programming languages like Haskell [4]. The intention is to take care of all program constructs of Haskell with operational significance. Thus the set of constructs like abstractions, applications, constructors, and case-expressions has to be extended by seq, the sequential-evaluation operator available in Haskell.

Our notion of correctness is based on the contextual equivalence, which equates programs if their termination behavior is identical if they are plugged in any surrounding larger program (i.e. any program context). Early work on the semantics of call-by-need evaluation can be found e.g. in [7]. Deep analyses of the contextual semantics of Haskell’s core-language by investigating extended lambda-calculi were done e.g. in [8, 21, 20], but all these works consider the untyped variant of the core language.

In untyped calculi all program contexts have to be considered for the contextual equivalence while in typed calculi only typed programs are compared and only correctly typed contexts have to be considered. Hence in the typed setting the set of testing contexts is a subset of the used contexts in the untyped setting. Consequences are that correct program transformation in the untyped calculi are also correct in the typed calculi (provided that the transformation is type-preserving) and more importantly that typed calculi allow more correct program transformations than untyped calculi since the set of contexts is smaller and thus the contextual equivalence is less discriminating than in the untyped calculi.

Thus it is reasonable to also explore the semantics and the correctness of program transformations of typed program calculi. There are also some investigations in calculi with polymorphic types, letrec and seq [22] adapting parametricity to polymorphic calculi with
Type variables: \( a \in \mathcal{A} \) where \( \mathcal{A} \) is the set of type variables

Term variables: \( x, x_i \in \mathcal{X} \) where \( \mathcal{X} \) is the set of term variables

Types: \( \tau \in \text{Typ} := a \mid (\tau_1 \to \tau_2) \mid (K \tau_1 \ldots \tau_{ar(K)}) \)

Polymorphic types: \( \rho \in \text{PTyp} := \tau \mid \lambda a.\rho \)

Expressions: \( e \in \text{Expr}_F := x : \rho \mid u \mid (e \rho) \mid (e_1 e_2) \mid (c : \tau e_1 \ldots e_{ar(c)}) \mid (\text{seq} e_1 e_2) \mid \text{letrec} x_1 : \rho_1 = e_1, \ldots, x_n : \rho_n = e_n \text{ in } e \mid \text{case}_K e \text{ of } (p_1 \to e_1) \ldots (p_n \to e_n) \)

where there is a pattern \( p \) for every constructor in \( D_K \)

Polymorphic expressions: \( u \in \text{PExpr}_F := x : \lambda a.\rho \mid \lambda x : \tau.e \mid (u \tau) \mid \Lambda a.\rho \)

Case-patterns: \( p := (c : \tau x_1 : \tau_1 \ldots x_{ar(c)} : \tau_{ar(c)}) \)

where \( x_i \) are different term variables

Figure 1 Types and expressions of the language \( L_F \).

The results in this paper are: The correctness of a large set of program transformations in \( L_F \) and \( L_P \) (see Corollary 4.5 and Proposition 5.3). By stand-alone proofs in the respective calculi we obtain a context lemma in \( L_F \) (Proposition 4.7); and a sound and complete applicative simulation \( \preceq_P \) in \( L_P \) (Theorem 5.6), which implies a ciu-Theorem 5.9 as a replacement for the context-lemma in \( L_P \). By analogy, and since the calculi are deterministic, we obtain that the fixpoint operators for \( \preceq_P \) are continuous, and so \( \preceq_P = \preceq_P^{\omega} \) where the latter is defined inductively (see Theorem 4.10). Using the fully abstract typed translation \( T : L_F \to L_P \), the results in \( L_P \) can be transferred to \( L_F \), using the methods on \( Q \)-similarity in [20].

2 Syntax of the Polymorphic Typed Call-By-Need Lambda Calculus

We define the polymorphically typed language \( L_F \) which employs cyclic sharing using a letrec [2] and is like a core-language of Haskell [4], and uses ideas of system-F polymorphism. The syntax of types and expressions is shown in Fig. 1. There are two classes of types. Types \( \tau \in \text{Typ} \) are like extended monomorphic types, where type variables are allowed, but are treated more or less as constants. Polymorphic types \( \rho \in \text{PTyp} \) allow to explicitly quantify type variables by \( \lambda \). Expressions are built from variables (which always occur with their type), abstractions, applications, type abstractions and applications, seq-expressions to seq, but not analyzing program transformations in depth. System F polymorphic calculi were first described in [3], are used in programming languages [10], and a variant of it is used in a Haskell compiler [4, 23].

In this paper we focus our investigations on a polymorphically typed calculus and thus we introduce a polymorphically typed lambda-calculus \( L_F \) with letrec, case, constructors and seq that models sharing on the expression level. The (predicative) polymorphism in the calculus is modelled in a system-F style by type abstractions. Predicative typing shows up in the formation rule for application, where the argument is not permitted to be polymorphic. As a second calculus, we present a call-by-name calculus \( L_P \) without letrec, together with a fully abstract translation \( T : L_F \to L_P \). There are type-erasing translations into untyped variants of the calculi (see [20]).
A generalized monomorphic type-system is used to form correctly typed expressions where $\lambda$ is also permitted in the syntax of types. The typing rules are in Fig. 2. Typing the polymorphic combinator $K := \Lambda a. \Lambda b. \lambda x : a. \lambda y : b. x$. It is polymorphic in the type variables $a, b$.

**Remark 2.2.** It is known that there is a practically problematic danger of growth of type expressions during reduction, as reported in [23], but this could be defeated by using bags for types. We could model this by a let for types, which, however, would lead to notational overhead. So, for simplicity, we treat the types in a non-sharing way.

A generalized monomorphic type-system is used to to form correctly typed expressions where $\lambda$ is also permitted in the syntax of types. The typing rules are in Fig. 2. Typing the polymorphic combinator $K := \Lambda a. \Lambda b. \lambda x : a. \lambda y : b. x$. results in $\Lambda a. \Lambda b. a \rightarrow (b \rightarrow a)$.

**Definition 2.3 (Contexts).** An $L_F$-context $C \in \text{Ctx}_F$ is an $L_F$-expression that has a single occurrence of the hole $[\cdot : \rho]$ of (polymorphic) type $\rho$ and is itself well-typed.

This represents contexts, where the hole maybe at polymorphic expressions.
We also assume that the topmost redex according to the rule is the top expression.

A reduction step consists of: (1) finding a normal-order redex, then (2) applying a reduction rule. Instead of defining step (1) by a syntactic definition reduction contexts – which is notionally complex in $L_F$ (see e.g. [21] for a similar language), we define the search by a labeling algorithm which uses two atomic labels sub, top, where top means the top-expression, and sub means “subterm” (in a letrec-expression). For an expression $e$ the labeling algorithm starts with $e^{top}$, where $e$ has no further inner labels top or sub. Then the rules of Fig. 3 are applied exhaustively. The role of top and sub is to prevent to label positions inside deep letrec-expressions. The labeling algorithm fails, if a loop is detected, which happens if a to-be-labeled position is already labeled sub, and otherwise, if no more rules are applicable or if the labeling algorithm has to stop, it succeeds. If we apply the labeling algorithm to contexts, then the contexts where the hole will be labeled with sub, or top are called the reduction contexts. We denote reduction contexts with $R$. Note that for the ease of reading, we omit the types of variables and constructors in the notation.

**Definition 3.1.** Normal-order reduction rules are defined in Fig. 4, where we assume that the labeling algorithm was used successfully before. Otherwise no normal-order reduction is applicable. In the presentation of the rules we only present the to-be-reduced subexpression. We also assume that the topmost redex according to the rule is the normal-order redex.

Note that the guiding principle in the cp-rules is to copy only values, i.e. polymorphic abstractions or cv-expressions. It is easy to verify that normal-order reduction is unique.

**Definition 3.2.** 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 polymorphic abstraction is an expression of the form $\Lambda a_1, \ldots, a_n. u$, where $n \geq 0$ and $u$ is an abstraction. Let $W \in WCtxt$ denote contexts according to the grammar $W \in WCtxt := [\ ] | (\text{letrec } Env \ in \ [\ ]).$ A weak head normal form (WHNF) is a polymorphic abstraction, or of the form $W[\bar{w}]$, where $w$ is a constructor application or an abstraction.

**Lemma 3.3.** Reduction does not change the type of expressions.

**Example 3.4.** As an example we reduce an expression including the polymorphic combinator $K := \Lambda a. \Lambda b. \lambda x : a. \lambda y : b.x$. Applying it to the type Bool, the constant True of type Bool and a type variable $a'$ is as follows: $(\Lambda a. \Lambda b. \lambda x : a. \lambda y : b.x) \text{Bool } a' \text{ True}$ results after three normal-order reductions in $(\text{letrec } x : \text{Bool } = \text{True in } \lambda y : a'.x)$. 

<table>
<thead>
<tr>
<th>Rule</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(e_1, e_2)_{\text{sub} \lor \text{top}}$</td>
<td>$\rightarrow (e_1^{\text{sub}}, e_2^{\text{sub}})$, $e_1 \neq \Lambda a.e'$</td>
</tr>
<tr>
<td>$(\text{letrec } Env \ in \ e)^{\text{top}}$</td>
<td>$\rightarrow (\text{letrec } Env \ in \ e^{\text{sub}})$</td>
</tr>
<tr>
<td>$(\text{letrec } x = e, Env \ in \ C[x^{\text{sub}}])$</td>
<td>$\rightarrow (\text{letrec } x = e^{\text{sub}}, Env \ in \ C[x^{\text{sub}}])$</td>
</tr>
<tr>
<td>$(\text{letrec } x = e_1, y = C[x^{\text{sub}}], Env \ in \ e_2)$</td>
<td>$\rightarrow (\text{letrec } x = e_1^{\text{sub}}, y = C[x^{\text{sub}}], Env \ in \ e_2)$</td>
</tr>
<tr>
<td>$(\text{seq } e_1, e_2)^{\text{sub} \lor \text{top}}$</td>
<td>$\rightarrow (\text{seq } e_1^{\text{sub}}, e_2^{\text{sub}})$</td>
</tr>
<tr>
<td>$(\text{case } e \ of \ alts)^{\text{sub} \lor \text{top}}$</td>
<td>$\rightarrow (\text{case } e^{\text{sub}} \ of \ alts)^{\text{sub}}$</td>
</tr>
<tr>
<td>$(\Lambda a. u)^{\text{sub} \lor \text{top}}$</td>
<td>$\rightarrow ((\Lambda a. e^{\text{sub}}) \ u)^{\text{sub}}$; then stop with success</td>
</tr>
<tr>
<td>$(\Lambda a. \Lambda x. a)^{\text{sub} \lor \text{top}}$</td>
<td>$\rightarrow (\Lambda a. x^{\text{sub}})$</td>
</tr>
</tbody>
</table>

$\text{sub} \lor \text{top}$ means label sub or top.
In this section we define 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 3.6. Let \( e \in L_F \). A normal order reduction sequence of \( e \) is called a (normal-order) evaluation if the last term is a WHNF. We write \( e \leadsto e' \) (\( e \) converges) iff there is an evaluation starting from \( e \) ending in WHNF \( e' \) (we omit \( e' \), if it is of no interest). Otherwise, if there is no evaluation of \( e \), we write \( e \uparrow \).

Definition 3.7 (Contextual Preorder and Equivalence). Contextual preorder \( \leq_F \) and contextual equivalence \( \sim_F \) are defined for equally typed expressions. For \( e_1, e_2 \) of type \( \rho \):

\[
\begin{align*}
\text{if } e_1 \leq_F e_2 & \iff \forall C[: \rho] : \tau . \text{ If } C[e_1] \text{ and } C[e_2] \text{ are closed, then } (C[e_1] \downarrow \Rightarrow C[e_2] \downarrow) \\
\text{if } e_1 \sim_F e_2 & \iff e_1 \leq_F e_2 \land e_2 \leq_F e_1
\end{align*}
\]

Note that we are only interested in top-expressions of closed type. It is standard to show that \( \leq_F \) is a precongruence, i.e. a compatible partial order, and that \( \sim_F \) is a congruence, i.e. a compatible equivalence relation. Also, a progress lemma holds for closed expressions \( e \): If no reduction is possible, then \( e \) is a WHNF, or the search for a redex fails.

4 Correctness of Program Transformations and Translations

A typed program transformation \( P \) is a binary relation on \( L_F \)-expressions, such that \((e_1, e_2) \in P\) is only valid for well-formed \( e_1, e_2 \) of equal type. The restriction of \( P \) to a type \( \rho \) is denoted with \( P_\rho \). A program transformation \( P \) is called correct iff for all \( \rho \) and all \((e_1, e_2) \in P_\rho \), the contextual equivalence relation \( e_1 \sim_F e_2 \) holds. Analogously, for untyped programs, a program transformation \( P \) is a binary relation on untyped expressions and it is correct if \( P \subseteq \sim \). Disproving the correctness of a (typed or untyped) program transformation is often easy, since a counter example consisting of a program context which distinguishes two related expressions by their convergence behavior is sufficient.

Definition 4.1. Let \( L_1, L_2 \) be two (typed or untyped) calculi with a notion of expressions, contexts, may-convergence \( \downarrow_i \), and contextual preorder \( \leq_i \). A translation \( \psi \) from calculus \( L_1 \) in \( L_2 \) (with equal set of types) mapping expressions to expressions and contexts into contexts where types are retained and \( \psi([\cdot]) = [\cdot] \) is 1. convergence equivalent if \( e_1 \downarrow_1 \iff \psi(e_1) \downarrow_2 \); 2. compositional if \( \psi(C[e_1]) = C[\psi(e_1)] \); 3. adequate if \( \psi(e_1) \leq_2 \psi(e_2) \Rightarrow e_1 \leq_1 e_2 \); and 4. fully abstract if \( \psi(e_1) \leq_2 \psi(e_2) \iff e_1 \leq_1 e_2 \).

4.1 Importing Results from Untyped Calculi

The untyped language \( L_{LR} \) has the same syntax as \( L_F \) except that variables have no type, and that type abstractions \( \lambda \alpha.e \) and type applications \( e \tau \) are not permitted. The normal order reduction for \( L_{LR} \) is defined as for \( L_F \) where the rule \( (L\beta) \) is not used. The semantics of \( L_{LR} \) was investigated e.g. in [20, 21].

Definition 4.2. The translation \( \varepsilon \) translates \( L_F \)-expressions into untyped expressions \( L_{LR} \) where we assume that the data types and the constructors are the same. It is defined as \( \varepsilon(\lambda \alpha.e) := \varepsilon(e) \), \( \varepsilon(e \tau) := \varepsilon(e) \), and on the other constructs \( \varepsilon \) acts homomorphically, removing type labels and types.

Since the untyped reduction is the same as the typed reduction if the types and \((L\beta)\)-reductions are ignored, and since the untyped WHNFs are exactly the typed WHNFs with the types removed, \( \varepsilon \) is convergence equivalent. Since it is also independent of the surrounding context, it is also compositional:

Lemma 4.3. The translation \( \varepsilon \) is convergence equivalent and compositional.
This implies that it is also adequate (see for example [17]).

\[ \text{Corollary 4.4. The translation } \varepsilon \text{ is adequate, which means that equivalences from } L_{LR} \text{ also hold in } L_{F}. \]

Proving correctness of program transformations is in general a hard task, since all contexts need to be taken into account. In e.g. [8, 21, 12] methods to prove correctness of program transformations for untyped letrec calculi were developed. As a first step we will use the result of [21] to lift untyped program equivalences into the typed calculus. In the calculus introduced in [21] the normal order reduction 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 in [15], see also [20]). This implies that all reduction rules of Fig. 4 and the optimizations garbage collection (gc) and general copying (gcp) (see Fig. 5) are correct program transformations for \( L_{F} \) (for gcp see [16]).

\[ \text{Corollary 4.5. The reductions rules from Figs. 4 and 5 are correct program transformations in } L_{F}, \text{ and can be used in any context.} \]

4.2 Context Lemma

For the context lemma we first define the \( \leq \)-relation for reduction contexts. A context lemma for a similar polymorphic calculus with a (more complex) type labeling but without explicit type abstractions and applications is in [15].

\[ \text{Definition 4.6. For a polymorphic type } \rho \text{ and } e_{1}, e_{2} \text{ of type } \rho, \text{ let } e_{1} \leq_{F,R} e_{2} \text{ hold if for all reduction contexts } R[\cdot : \rho] \text{ such that } R[e_{1}], R[e_{2}] \text{ are closed: } R[e_{1}] \Downarrow \implies R[e_{2}] \Downarrow. \]

\[ \text{Proposition 4.7 (Context Lemma for } L_{F}). \text{ Let } \rho \text{ be a polymorphic type and } e_{1}, e_{2} \text{ be of type } \rho. \text{ Then } e_{1} \leq_{F,R} e_{2} \iff e_{1} \leq_{F} e_{2}. \]

The proof is standard, e.g. it follows the technique explained in [18]. However, the proof technique relies on the proper use of sharing. For instance, the proof technique breaks down in a call-by-name calculus (like \( L_{F} \) in Sect. 5).

The context lemma 4.7 immediately implies:

\[ \text{Corollary 4.8. If } e_{1}, e_{2} \text{ are closed expressions of equal type with } e_{1} \Downarrow, e_{2} \Downarrow, \text{ then } e_{1} \sim_{F} e_{2}. \]

4.3 Inductive Similarity For \( L_{F} \)

As a further proof tool for showing contextual equivalences, we define an improved similarity definition in \( L_{F} \), which is often superior to the context lemma.
Definition 4.9. We define $\equiv_\omega := \bigcap_{n \geq 0} \equiv_n$ where for $n \geq 0$, $\equiv_n$ is defined on closed $L_F$-expressions $e_1, e_2$ of the same type as follows:

1. $e_1 \equiv_0 e_2$ is always true.
2. $e_1 \equiv_n e_2$ for $n > 0$ holds if the following conditions hold:
   a. if $e_1 \downarrow \Lambda a. e_1'$, then $e_2 \downarrow \Lambda a. e_2'$, and for all $\tau$: $e_1'[\tau/a] \equiv_{n-1} e_2'[\tau/a]$.
   b. if $e_1 \downarrow W[\lambda x : \tau. e_1']$, then $e_2 \downarrow W[\lambda x : \tau. e_2']$ and for all closed $e : \tau$: $W[\lambda x. e_1'] e \equiv_{n-1} W[\lambda x. e_2'] e$.
   c. if $e_1 \downarrow W[c e_1' \ldots e_m']$, then $e_2 \downarrow W[c e_1'' \ldots e_m'']$ and for all $i$: $W[e_i'] \equiv_{n-1} W[e_i'']$.

The proof of soundness (and completeness) of inductive similarity w.r.t. contextual preorder can be constructed similar to an analogous proof in the untyped case (see [20]). We sketch the proof: The language $L_F$ is a polymorphically typed call-by-name lambda-calculus with fixpoint combinators, but no letrec (see Sect. 5). In $L_F$ coincidence of contextual preorder and inductive similarity can be shown by proving soundness and completeness of an applicative similarity using Howe’s method (Theorem 5.6). Using some further arguments which rely on the continuity of the fixpoint combinators show the coincidence of applicative and inductive similarity (Theorem 5.8). Then a translation $T : L_P \rightarrow L_F$ is defined (below in Sect. 5.2) which replaces letrec-expressions by fixpoint combinators. Since the translation $T$ is fully abstract and surjective on the equivalence classes of contextual equivalence, Theorem 5.8 can be lifted into $L_F$ which shows the following theorem:

Theorem 4.10. $\equiv_\omega = \leq_F$

As a corollary we show that $\varepsilon$ is not fully abstract.

Proposition 4.11. The translation $\varepsilon$ is not fully abstract.

Proof. In $L_F$ the expressions $e_1 = \lambda x : \text{Bool}. \text{case} x \text{ of} (\text{True} \rightarrow x) (\text{False} \rightarrow x)$ and $e_2 = \lambda x : \text{Bool}. x$ are contextually equivalent. This follows by Theorem 4.10 and since the possible arguments can be classified as equivalent to $\bot$ (a nonterminating expression of type Bool), True or False, and the result is equivalent or equal in all cases. However, $\varepsilon(e_1)$ and $\varepsilon(e_2)$ are different in $L$, which can be seen by applying them to $\lambda x. x$.

As an example, we show the equivalence of other polymorphic expressions.

Proposition 4.12. In the language $L_F$ the two expressions $e_1 = \Lambda a. \lambda x : (\text{List} a). x$ and $e_2 = \Lambda a. \lambda x : (\text{List} a). \text{case} x \text{ of} (\text{Nil} \rightarrow \text{Nil}) (\text{Cons} y_1 y_2 \rightarrow \text{Cons} y_1 y_2)$ of polymorphic type $\Lambda a. \text{List} a \rightarrow \text{List} a$ are contextually equivalent.

Proof. We use inductive similarity. Application to $\tau$ yields two monomorphic abstractions. Further application can only be to arguments without WHNF, or with WHNF $W[\text{Nil}]$ or $W[\text{Cons} e_1' e_2']$. In all cases, the results are obviously contextually equivalent and thus applicative similar.

5 A Call-by-Name Polymorphic Lambda Calculus

We present a call-by-name polymorphic lambda calculus $L_P$ as a second calculus, with built-in multi-fixpoint constructions $\Psi$ for representing mutual recursive functions.

We argue that there is a fully abstract translation $T$ from $L_F$ into the calculus $L_P$. To demonstrate the power, we show that there is a polymorphic applicative simulation in $L_P$ that is useful for recognizing equivalences. The calculus $L_P$ is related to the lazy lambda calculus [1], however $L_P$ is more expressive and typed.
Polymorphic expressions: \( u \in P\text{Expr}_P \) \( \iff \) \( x : \Lambda a.\rho \mid \lambda x : \tau.e \mid (u \tau) \mid \Lambda a.u \)

Expressions:
\( e, e_i \in \text{Expr}_P \) \( \iff \) \( x : \rho \mid u \mid (e \tau) \mid (e_1 e_2) \mid (c : \tau e_1 \ldots e_{ar(c)}) \mid (\text{case}_K e \text{ of } \text{alts}) \mid (\text{seq } e_1 e_2) \mid \Psi_{i,n}\tau/x.p.e \)

Reduction contexts:
\( R_P \in \mathcal{R}_P \) \( \iff \) \( [\cdot] \mid (R_P e) \mid \text{case}_K R_P \text{ of } \text{alts} \mid \text{seq } R_P e \)

Normal order reduction:
\begin{align*}
\text{rbeta} & : R_P[((\lambda x.e_1)e_2)] \xrightarrow{P} R_P[e_1[e_2/x]] \\
\text{rcase} & : R_P[\text{case}_K (c e_1 \ldots e_{ar(c)}) \text{ of } \ldots ((c x_1 \ldots x_{ar(c)}) \rightarrow e) \ldots] \xrightarrow{P} e[c/x_1, \ldots, x_{ar(c)}] \\
\text{rnseq} & : R_P[\text{seq } v e] \xrightarrow{P} R_P[e], \text{ if } v \text{ is an } L_P-\text{WHNF.} \\
\text{rtype} & : R_P[(\Lambda a.e) \tau] \xrightarrow{P} R_P[e[\tau/a]] \\
\text{rnfix} & : R_P[\Psi_{i,n}\tau/x.e] \xrightarrow{P} R_P[(e_1[\Psi_{i,n}\tau/x_1, \ldots, \Psi_{n,n}\tau/x_n])] \\
\end{align*}

\[\text{Figure 6} \] Syntax and normal order reduction \( \xrightarrow{P} \) of \( L_P. \)

### 5.1 The Calculus \( L_P \)

**Definition 5.1.** The calculus \( L_P \) is defined as follows. The set \( \text{Expr}_P \) of \( L_P \)-expressions is that of \( L_F \), where \text{letrec} is removed, see Fig. 6. \( (\Psi_{i,n}\tau/x.p.e) \) is a family of multi-fixpoint-operators, where \( 1 \leq i \leq n \) and where \( \tau \) means \( x_1, \ldots, x_n \), similarly for \( e \).

The typing rules are according to Fig. 2 with the additional rule for the \( \Psi \)-operator:

\[
\frac{
\text{for } i = 1, \ldots, n: e_i : \rho_i, x_i : \rho_i
}{
(\Psi_{i,n}\tau/x.p.e) : \rho_i
}\]

WHNFs in \( L_P \) are (polymorphic) abstractions \( \Lambda a_1, \ldots, \Lambda a_n. \lambda x : \rho.e \) and constructor applications. In Fig. 6 the reduction rules and the normal order reduction \( \xrightarrow{P} \) for \( L_P \) using reduction contexts \( R_P \) are given. The contextual preorder \( \leq_P \) and contextual equality \( \sim_P \) are defined as above for the calculus \( L_F \), where convergence to \( L_P \)-WHNFs and where holes in contexts are permitted to have polymorphic type \( \rho \), but the context itself must have plain type.

Note that our syntax permits a “polymorphic bot”: \( \Psi_{1,1} x.(\Lambda a.a).x \). An example is polymorphic length of lists (type-labels in the notation are partially omitted):

\[
\text{length} \quad := \quad (\Psi_{1,1}\text{len}\text{:List } a \rightarrow \text{Nat}.\Lambda a.\Lambda x\text{:List } a.
\text{case } x s \text{ of } (\text{Nil} \rightarrow 0) (\text{Cons } y s \rightarrow \text{Succ } (\text{len } a y s))
\]

Our formulation is a bit more general than that in [10] for system F and ML, which corresponds to Milner type checking, whereas our formulation permits differently typed occurrences of a recursive polymorphic functions in its defining body, and so corresponds to iterative (polymorphic) type checking.

### 5.2 On the Translation \( T : L_F \rightarrow L_P \)

In order to define the typed translation \( T : L_F \rightarrow L_P \), we adapt the combined translation from the untyped variant in [20].

**Definition 5.2.** The translation \( T : L_F \rightarrow L_P \) is defined as:

\( T(\text{letrec } x_1 = e_1; \ldots; x_n = e_n \text{ in } e') := T(e')[(\Psi_{1,n}\tau.f)/x_1, \ldots, (\Psi_{n,n}\tau.f)/x_n] \) and where \( f_i = \lambda x_1, \ldots, x_n. T(e_i) \) for \( i = 1, \ldots, n \), and it is homomorphically on other constructs.
Proposition 5.3. The translation $T : L_F \rightarrow L_P$ is fully abstract and surjective on contextual-equivalence classes.

Proof. This can be derived from the untyped version in [20]. ▶

For inheriting correct translations we again use a translation $\varepsilon_P$ into an untyped call-by-name calculus $L_{icc}$ in [20] by forgetting the types (and allowing also untyped expressions), and with $\varepsilon_P(\Psi_{1,1}.x.s) = Y (\varepsilon_P(\lambda x.s))$, where $Y$ is the untyped fixpoint combinator in $L_{icc}$. Similarly, we can translate $\Psi_{i,n}(\ldots)$ using the multi fixpoint combinators in [20].

Proposition 5.4. The translation $\varepsilon_P : L_P \rightarrow L_{icc}$ is convergence equivalent and compositional, hence adequate, but it is not fully abstract. The reduction rules of $L_P$ are correct.

Proof. Only the case of a type abstraction requires an extra argument. It is sufficient that the contexts used for testing have the same type of the hole as the expressions. Then adequacy of the translation $\varepsilon_P$ can be used. ◀

5.3 Applicative Simulation in $L_P$

In the following we use binary relations $\eta$ on closed expressions (of the same type). We need closing substitutions $\sigma$ which are defined as mapping free variables (of plain type) to closed expressions of the same type, and all type variables to plain types. This extension to type variables is the key to apply applicative simulation also to polymorphic functions. The open extension $\eta^o$ of $\eta$ is the relation on open expressions, where $e_1 \eta^o e_2$ is valid iff for all substitutions $\sigma$ where $\sigma(e_1), \sigma(e_2)$ are closed, the relation $\sigma(e_1) \eta_\sigma \sigma(e_2)$ holds. We will also use the restriction of a binary relation $\eta$ to closed expressions which is denoted as $\eta^c$.

Definition 5.5 (Applicative Similarity in $L_P$). Let $\eta$ be a binary relation on closed $L_P$-expressions, where only expression of equal syntactic type can be related. Let $F_P$ be the operator on relations on closed $L_P$-expressions s.t. $e_1 F_P(\eta) e_2$ holds iff

$$
equiv_1 F_P(\lambda x.e')_1 \implies (\nequiv_2 F_P(\lambda x.e')_2 \land \equiv_1 \eta^c e'_2)$$

$$\equiv_1 F_P(c e'_1 \ldots e'_n) \implies (\equiv_2 F_P(c e''_1 \ldots e''_n) \land \text{the relation } e'_i \eta e''_i \text{ holds for all } i)$$

$$\equiv_1 F_P(\eta a.e')_1 \implies (\equiv_2 F_P(\eta a.e'_2) \land \equiv_1 \eta^c e'_2)$$

Applicative similarity $\equiv_P$ is defined as the greatest fixpoint of the operator $F_P$. Mutual similarity $\equiv_P$ is defined as $e_1 \equiv_1 e_2$ iff $e_1 \equiv_P e_2 \land e_2 \equiv_P e_1$.

Note that the operator $F_P$ is monotone, hence the greatest fixpoint $\equiv_P$ exists.

Howe’s method [5, 6] to show that $\equiv_P$ is a pre-congruence and equal to $\equiv_P$ can be applied without unexpected changes, see also [11] and [20, Sect. 4.2] where the only extra feature is the typing. For completeness, we show $\equiv_P P \subseteq \equiv_P$ by proving that contextual equivalence in $L_P$ satisfies the fixpoint conditions of $F_P$ and then we use coinduction. By the properties of $\equiv_P$ this implies that $\equiv_P \subseteq \equiv_P$ also holds for open expressions.

Theorem 5.6. $\leq_P = \equiv_P$, and $\leq_P = \equiv_P$.

Proof. Only the completeness part is missing. We have to analyze the three conditions of Definition 5.5, where we use Proposition 5.4 several times.

1. If $e_1 \leq_P e_2$, and $e_1 \equiv_P \lambda x : \tau.e'_1$, then clearly $e_2 \equiv P \lambda x : \tau.e'_2$. Since beta-reduction is correct, also for all closed expressions $e : \tau e_1 e \leq_P e_2 e$, since $\leq_P$ is a precongruence, and since reduction sequences are correct w.r.t. $\sim_P$; thus $e'_1[e/x] \leq_P e'_2[e/x]$. 
2. If \( e_1 \leq_P e_2 \) and \( e_1 \downarrow_P (e' \ldots e'_m) \), then \( e_2 \downarrow_P e_2. \) Since \( \leq_P \) is a pre-congruence, reduction is correct, and using simple contexts like \( \text{case} \left[ \ldots \right] \) of \(( (c_1 \ldots c_n) \rightarrow \text{True}) \ldots (p' \rightarrow \bot) \) and \( \text{case} \left[ \ldots \right] \) of \(( (c_1 \ldots c_n) \rightarrow x_1) \ldots (p' \rightarrow \bot) \), we see that \( v_2 \) is of the form \(( c'_{i_1} \ldots c'_{i_m}) \), where \( c'_{i} \leq_P e'_{i} \) for all \( i \).

3. The last case is the type abstraction and type substitution. The same arguments as above can be used, by plugging the expressions \( e_1, e_2 \) into a context \( \left[ \cdot \right] \tau \).

### 5.4 Inductive Similarity For \( L_P \)

We define the inductive version of applicative similarity in \( L_P \):

**Definition 5.7.** We define \( \preceq_{P, \omega} := \bigcap_{n \geq 0} \preceq_{P, n} \) where for \( n \geq 0 \), \( \preceq_{P, n} \) is defined on closed \( L_P \)-expressions \( e_1, e_2 \) of the same type as follows:

1. \( e_1 \preceq_{P, 0} e_2 \) is always true.
2. \( e_1 \preceq_{P, n} e_2 \) for \( n > 0 \) holds if the following conditions hold:
   
   a. If \( e_1 \downarrow \Lambda a.e'_1, \) then \( e_2 \downarrow \Lambda a.e'_2, \) and for all \( \tau: e'_1[\tau/a] \preceq_{P, n-1} e'_2[\tau/a], \)
   
   b. if \( e_1 \downarrow \lambda x. \tau.e'_1 \) then \( e_2 \downarrow \lambda x. \tau.e'_2 \) and for all closed \( e : \tau: e'_1[e/x] \preceq_{P, n-1} e'_2[e/x]. \)
   
   c. if \( e_1 \downarrow (e'_{i_1} \ldots e'_{i_m}) \) then \( e_2 \downarrow (e''_{i_1} \ldots e''_{i_m}) \) and for all \( i: e'_{i} \preceq_{P, n-1} e''_{i}. \)

   Proving continuity of the fixpoint operator of \( \preceq_P \) as in (see [20]), we obtain:

**Theorem 5.8.** \( \preceq_{P, \omega}^0 = \leq_P \)

A corollary is a ciu-Theorem: Let \( e_1 \leq_{ciu} e_2 \) for two \( L_P \)-expressions \( e_1, e_2 \) of equal type iff for all closed \( L_P \)-reduction contexts \( R_P \), and all (well-typed) substitutions \( \sigma \) where \( \sigma(e_1) \) and \( \sigma(e_2) \) are closed: \( R_P[\sigma(e_1)] \Rightarrow R_P[\sigma(e_2)] \).

**Theorem 5.9.** \( \leq_{ciu} = \leq_P \)

**Proof.** We apply the knowledge about applicative simulation \( \preceq \): If \( e_1 \preceq e_2 \), then for all \( \sigma \) where \( \sigma(e_1), \sigma(e_2) \) are closed: \( \sigma(e_1) \preceq \sigma(e_2). \) Since we already know that \( \preceq \) is a pre-congruence, we also obtain \( R_P[\sigma(e_1)] \preceq R_P[\sigma(e_2)], \) and so \( R_P[\sigma(e_1)] \Rightarrow R_P[\sigma(e_2)] \).

We show that the ciu-relation implies \( \preceq^0 \): For closed \( e_1, e_2 \) it holds: Since the calculus \( L_P \) is deterministic, it is sufficient to restrict the test to the contexts \( \left[ \cdot \right] \) \( e'_1 \ldots e'_m \) for all \( m \geq 0 \) and all closed \( e'_{i} \) (see Theorem 5.8). So, if the ciu-condition for closed expressions holds, we obtain \( e_1 \leq_P e_2 \) and so \( e_1 \preceq e_2. \) The ciu-relation for non-closed \( e_1, e_2 \) is equivalent to the open extension of \( \preceq \), i.e., it implies \( e_1 \preceq^0 e_2, \) and thus we have the equality: \( \preceq^0 = \leq_{ciu}. \)

### 6 Conclusion

Using a system-F-like extension of untyped extended lambda-calculi with case, constructors, and seq, and call-by-need and call-by-name variants, we present several tools for recognizing correct transformations This could potentially be used in lazy functional programming languages like Haskell.

Further research may be to investigate a polymorphic variant of (the non-deterministic language) Concurrent Haskell with futures (CHF) [14, 13]. A non-deterministic extension of \( L_F \) and \( L_P \) with \( \texttt{amb} \) appears unrealistic, since there are counterexamples for combinations of \( \texttt{letrec} \) [19] and since call-by-name and call-need nondeterminism are very different.

**Acknowledgment.** We thank the anonymous reviewers for their valuable comments.
References