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:	$\tau \in \mathit{Typ}$	$:= a \mid (\tau_1 \to \tau_2) \mid (K\ \tau_1 \ldots \tau_{ar(K)})$
Polymorphic types:	$\rho \in \mathit{PTyp}$	$:= \tau \mid \boldsymbol{\lambda} a.\rho$
Expressions:	$e \in \mathit{Expr}_F$	$:= x : \rho \mid u \mid (e\ \tau) \mid (e_1\ e_2) \mid (c : \tau\ e_1\ \ldots\ e_{ar(c)})$
		$\mid$ (`seq` $e_1\ e_2$) $\mid$ `letrec` $x_1 : \rho_1 = e_1, \ldots, x_n : \rho_n = e_n$ `in` e
		$\mid$ `case`$_K$ e `of` (p_1 `->` e_1) $\ldots$ (p_n `->` e_n)
		where there is a pattern p for every constructor in D_K
Polymorphic expressions:	$u \in \mathit{PExpr}_F$	$:= x : \boldsymbol{\lambda} a.\rho \mid \lambda x : \tau.e \mid (u\ \tau) \mid \Lambda a.u$
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.

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]).

$$\begin{tikzcd}
\text{typed:} & L_F \arrow[r, "T"] \arrow[d, "\varepsilon"] & L_P \arrow[d, "\varepsilon_P"] \\
\text{untyped:} & L_{LR} \text{ in [20]} & L_{lcc} \text{ in [20]}
\end{tikzcd}$$

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 $\preccurlyeq_P$ 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 $\preccurlyeq_P$ are continuous, and so $\preccurlyeq_P = \preccurlyeq_{P,\omega}$ where the latter is defined inductively (see Theorem 4.10). Using the fully abstract typed translation T, 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 $\tau \in \mathit{Typ}$ are like extended monomorphic types, where type variables are allowed, but are treated more or less as constants. Polymorphic types $\rho \in \mathit{PTyp}$ allow to explicitly quantify type variables by $\boldsymbol{\lambda}$. Expressions are built from variables (which always occur with their type), abstractions, applications, type abstractions and applications, `seq`-expressions to

$$\frac{e:\tau'}{(\lambda x:\tau.e):\tau\to\tau'} \qquad \frac{e:\rho}{\Lambda a.e:\boldsymbol{\lambda} a.\rho} \qquad \frac{e:\boldsymbol{\lambda} a.\rho}{(e\ \tau):\rho[\tau/a]} \qquad \frac{e:\tau_1\to\tau_2 \quad e':\tau_1}{(e\ e'):\tau_2}$$

$$\frac{e_1:\rho_1 \quad \ldots \quad e_n:\rho_n \quad e:\rho}{(\texttt{letrec}\ x_1:\rho_1=e_1,\ldots,x_n:\rho_n=e_n\ \texttt{in}\ e):\rho} \qquad \frac{e_1:\rho \quad e_2:\rho'}{(\texttt{seq}\ e_1\ e_2):\rho'}$$

$$\frac{\begin{array}{l} e_1:\tau_1,\ldots,e_n:\tau_n \quad \tau=\tau_1\to\ldots\to\tau_n\to\tau_{n+1} \\ \quad c:\boldsymbol{\lambda} a_1,\ldots,a_m.\tau'' \text{ is the general type of } c \\ \text{there are } \tau_1',\ldots,\tau_m':\tau''[\tau_1'/a_1,\ldots,\tau_m'/a_m]=\tau \end{array}}{(c:\tau\ e_1,\ldots e_n):\tau_{n+1}} \qquad \frac{e:\tau \quad p_i:\tau \quad e_i:\rho}{(\texttt{case}_K\ e\ \texttt{of}\ (p_1\ \texttt{->}\ e_1)\ldots(p_n\ \texttt{->}\ e_n)):\rho}$$

Figure 2 Typing Rules for L_F.

model strict evaluation, recursive `letrec`, constructor applications and `case`-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 [9]). We assume that the constant type constructors `Bool`, `Nat` and the unary `List` are already defined. For every type-constructor K of arity $ar(K)$, 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 c of K comes with a type $\boldsymbol{\lambda} a_1,\ldots,a_{ar(K)}.\tau_1 \to \ldots \to \tau_{ar(c)} \to K\ a_1 \ldots a_{ar(K)}$. We assume that the following is available: $D_{\texttt{Bool}} = \{\texttt{True},\texttt{False}\}$, with $\texttt{True}:\texttt{Bool}$, $\texttt{False}:\texttt{Bool}$, $D_{\texttt{List}} = \{\texttt{Nil},\texttt{Cons}\}$, with $\texttt{Nil}:\boldsymbol{\lambda} a.\texttt{List}\ a$, $\texttt{Cons}:\boldsymbol{\lambda} a.a\to\texttt{List}\ a\to\texttt{List}\ a$, and $D_{\texttt{Nat}} = \{0,\texttt{Succ}\}$, where $0:\texttt{Nat}$ and $\texttt{Succ}:\texttt{Nat}\to\texttt{Nat}$.

The scoping is as expected: For expressions, λx, Λa, $(p \to \ldots)$ open a scope, and `letrec` opens a recursive scope. For types, $\boldsymbol{\lambda} a$ opens a scope. Types of the expressions are (generalized) monomorphic and the polymorphic aspect is the type computation via type abstractions and beta-reduction for type applications. For the reduction, the idea is that types could be omitted from reduction without any change in the operational reduction sequence (see Sect. 4.1). The body u of a type abstractions $\Lambda a.u$ are syntactically restricted (see Fig. 1), which implies that reduction cannot generate a `letrec`-expression as its body. We will explain the reason for this restriction in Remark 3.5 below. Note that the syntax permits a polymorphic non-termination expression (i.e. a "bot").

▶ **Example 2.1.** An example is the polymorphic combinator $\mathbf{K} := \Lambda a.\Lambda b.\lambda x:a.\lambda y:b.x$. 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 dags 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 $\boldsymbol{\lambda}$ is also permitted in the syntax of types. The typing rules are in Fig. 2. Typing the polymorphic combinator $\mathbf{K} := \Lambda a.\Lambda b.\lambda x:a.\lambda y:b.x.$ results in $\boldsymbol{\lambda} a.\boldsymbol{\lambda} b.a \to (b \to a)$.

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

This represents contexts, where the hole maybe at polymorphic expressions.

$(e_1\ e_2)^{\mathsf{sub}\vee\mathsf{top}}$	$\to (e_1^{\mathsf{sub}}\ e_2)^{\mathsf{sub}}$	$e_1 \neq \Lambda a.e'$
$(\texttt{letrec}\ Env\ \texttt{in}\ e)^{\mathsf{top}}$	$\to (\texttt{letrec}\ Env\ \texttt{in}\ e^{\mathsf{sub}})^{\mathsf{sub}}$	
$(\texttt{letrec}\ x = e, Env\ \texttt{in}\ \mathbb{C}[x^{\mathsf{sub}}])$	$\to (\texttt{letrec}\ x = e^{\mathsf{sub}}, Env\ \texttt{in}\ \mathbb{C}[x^{\mathsf{sub}}])$	
$(\texttt{letrec}\ x = e_1, y = \mathbb{C}[x^{\mathsf{sub}}], Env\ \texttt{in}\ e_2)$	$\to (\texttt{letrec}\ x = e_1^{\mathsf{sub}}, y = \mathbb{C}[x^{\mathsf{sub}}], Env\ \texttt{in}\ e_2)$	
$(\texttt{seq}\ e_1\ e_2)^{\mathsf{sub}\vee\mathsf{top}}$	$\to (\texttt{seq}\ e_1^{\mathsf{sub}}\ e_2)^{\mathsf{sub}}$	
$(\texttt{case}\ e\ \texttt{of}\ alts)^{\mathsf{sub}\vee\mathsf{top}}$	$\to (\texttt{case}\ e^{\mathsf{sub}}\ \texttt{of}\ alts)^{\mathsf{sub}}$	
$((\Lambda a.u)\ \tau)^{\mathsf{sub}\vee\mathsf{top}}$	$\to ((\Lambda a.e^{\mathsf{sub}})\ \tau)^{\mathsf{sub}}$; then stop with success	
$(\Lambda a.u)^{\mathsf{sub}\vee\mathsf{top}}$	$\to (\Lambda a.u^{\mathsf{sub}})^{\mathsf{sub}}$	
$\mathsf{sub} \vee \mathsf{top}$ means label sub or top.		

Figure 3 Searching the normal-order redex using labels.

3 Small-Step Operational Semantics of L_F

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 notationally 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 $\mathsf{sub}, \mathsf{top}$, 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 $\mathbb{R}$. 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 $(c\ x_1 \dots x_n)$ where c is a constructor and x_i are variables. A *polymorphic abstraction* is an expression of the form $\Lambda a_1, \dots, \Lambda a_n.u$, where $n \geq 0$ and u is an abstraction. Let $\mathbb{W} \in \mathit{WCtxt}$ denote contexts according to the grammar $\mathbb{W} \in \mathit{WCtxt} ::= [\cdot] \mid (\texttt{letrec}\ Env\ \texttt{in}\ [\cdot])$. A *weak head normal form* (WHNF) is a polymorphic abstraction, or of the form $W[w]$, 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 $\mathbf{K} := \Lambda a.\Lambda b.\lambda x : a.\lambda y : b.x$. Applying it to the type `Bool`, the constant `True` of type `Bool` and a type variable a' is as follows: $(\Lambda a.\Lambda b.\lambda x : a, \lambda y : b.x)$ `Bool` a' `True` results after three normal-order reductions in $(\texttt{letrec}\ x : \texttt{Bool} = \texttt{True}\ \texttt{in}\ \lambda y : a'.x)$.

(lbeta)	$((\lambda x:\tau.e_1)^{\mathsf{sub}}\ e_2) \to \mathtt{letrec}\ x:\tau = e_1\ \mathtt{in}\ e_2$
(Lbeta)	$((\Lambda a.u)^{\mathsf{sub}}\ \tau) \to u[\tau/a]$
(cp-in)	$\mathtt{letrec}\ x = v^{\mathsf{sub}}, \mathit{Env}\ \mathtt{in}\ \mathbb{C}[x^{\mathsf{sub}}] \to \mathtt{letrec}\ x = v, \mathit{Env}\ \mathtt{in}\ \mathbb{C}[v]$ where v is a polymorphic abstraction, or a cv-expression
(cp-e)	$\mathtt{letrec}\ x = v^{\mathsf{sub}}, y = \mathbb{C}[x^{\mathsf{sub}}], \mathit{Env}\ \mathtt{in}\ e$ $\to \mathtt{letrec}\ x = v, y = \mathbb{C}[v], \mathit{Env}\ \mathtt{in}\ e$ where v is a polymorphic abstraction, or a cv-expression
(cpcx-in)	$\mathtt{letrec}\ x = (c:\tau\ e_1 \dots e_n)^{\mathsf{sub}}, \mathit{Env}\ \mathtt{in}\ \mathbb{C}[x^{\mathsf{sub}}]$ $\to \mathtt{letrec}\ x = (c:\tau\ x_1 \dots x_n), x_1:\tau_1 = e_1, \dots, x_n:\tau_n = e_n, \mathit{Env}$ $\mathtt{in}\ \mathbb{C}[(c\ x_1 \dots x_n)]$ where the types τ_i are computed as the type of e_i
(cpcx-e)	$\mathtt{letrec}\ x = (c:\tau\ e_1 \dots e_n)^{\mathsf{sub}}, y = \mathbb{C}[x^{\mathsf{sub}}], \mathit{Env}\ \mathtt{in}\ e$ $\to \mathtt{letrec}\ x = (c:\tau\ e_1 \dots e_n), x_1:\tau_1 = e_1, \dots, x_n:\tau_n = e_n, y = \mathbb{C}[(c\ x_1 \dots x_n)],$ $\mathit{Env}\ \mathtt{in}\ e$ where the types τ_i are computed as the type of e_i
(case)	$(\mathtt{case}\ (c\ e_1 \dots e_n)^{\mathsf{sub}}\ \mathtt{of}\ \dots ((c\ y_1:\tau_1 \dots y_n:\tau_n)\ \mathtt{->}\ e) \dots)$ $\to \mathtt{letrec}\ y_1:\tau_1 = e_1, \dots, y_n:\tau_n = e_n\ \mathtt{in}\ e$
(case)	$(\mathtt{case}\ c^{\mathsf{sub}}\ \mathtt{of}\ \dots (c\ \mathtt{->}\ e) \dots) \to e$
(seq)	$(\mathtt{seq}\ v^{\mathsf{sub}}\ e) \to e$ if v is a constructor application or a polymorphic abstraction
(llet-e)	$\mathtt{letrec}\ \mathit{Env}_1, x = (\mathtt{letrec}\ \mathit{Env}_2\ \mathtt{in}\ e_1)^{\mathsf{sub}}\ \mathtt{in}\ e_2$ $\to \mathtt{letrec}\ \mathit{Env}_1, \mathit{Env}_2, x = e_1\ \mathtt{in}\ e_2$
(llet-in)	$\mathtt{letrec}\ \mathit{Env}_1\ \mathtt{in}\ (\mathtt{letrec}\ \mathit{Env}_2\ \mathtt{in}\ e)^{\mathsf{sub}} \to \mathtt{letrec}\ \mathit{Env}_1, \mathit{Env}_2\ \mathtt{in}\ e$
(lapp)	$((\mathtt{letrec}\ \mathit{Env}\ \mathtt{in}\ e_1)^{\mathsf{sub}}\ e_2) \to \mathtt{letrec}\ \mathit{Env}\ \mathtt{in}\ (e_1\ e_2)$
(lseq)	$(\mathtt{seq}\ (\mathtt{letrec}\ \mathit{Env}\ \mathtt{in}\ e_1)^{\mathsf{sub}}\ e_2) \to \mathtt{letrec}\ \mathit{Env}\ \mathtt{in}\ (\mathtt{seq}\ e_1\ e_2)$
(lcase)	$(\mathtt{case}\ (\mathtt{letrec}\ \mathit{Env}\ \mathtt{in}\ e)^{\mathsf{sub}}\ \mathtt{of}\ \mathit{alts}) \to \mathtt{letrec}\ \mathit{Env}\ \mathtt{in}\ (\mathtt{case}\ e\ \mathtt{of}\ \mathit{alts})$

Figure 4 Normal-order rules.

▶ **Remark 3.5.** On typed and untyped sharing: There is a constellation that has to be excluded (by syntax): expressions of the form $e = \Lambda a_1. \dots a_n.(\mathtt{letrec}\ \mathit{Env}\ \mathtt{in}\ t)$. The technical problem is that the (cp)-rules want to copy these expressions. However, in the untyped case, this is not possible, but instead a let-shifting can be done. In the typed case the scoping of the types prevents this let-shifting, and so the expression is stuck: it cannot be further reduced. Analyzing the usage at runtime of the expressions in the environment Env, it turns out that it does not make sense to share them among differently typed copies, since it is not type-safe. So the intention can only be to copy Env together with the abstraction. But then the elements in Env are not really useful, since they must have all types. Due to this conflict with the untyped reduction, the body e in $\Lambda a.e$ is syntactically restricted to expressions $u \in \mathit{PExpr}_F$. I.e., `letrec`-expressions and also expressions which may reduce to a `letrec`-expression (e.g. an application) are forbidden for e. If e is a variable, then it must have a type of the form $\boldsymbol{\lambda}.\rho$ which again ensures that the variable cannot be replaced by arbitrary expressions. For the same reasons, we also forbid constructors at the position for e in $\Lambda a.e$. Thus allowed expressions for e are type applications, abstractions and polymorphic variables in Λ-bodies, which also enforces potential reductions in the body. We permit only $\Lambda a_1. \dots a_n.\lambda x.e$ as proper polymorphic WHNFs.

3.1 Contextual Equivalence

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 $e{\downarrow}e'$ (*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 ρ:

$$\begin{array}{lll} e_1 \leq_F e_2 & \text{iff} & \forall \mathbb{C}[\cdot : \rho] : \tau : \text{ If } \mathbb{C}[e_1] \text{ and } \mathbb{C}[e_2] \text{ are closed, then } (\mathbb{C}[e_1]{\downarrow} \implies \mathbb{C}[e_2]{\downarrow}) \\ e_1 \sim_F e_2 & \text{iff} & e_1 \leq_F e_2 \;\; \wedge \;\; e_2 \leq_F e_1 \end{array}$$

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 ρ is denoted with P_ρ. A program transformation P is called *correct* iff for all ρ 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 ψ 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 $e{\downarrow}_1 \iff \psi(e){\downarrow}_2$; **2. compositional** if $\psi(\mathbb{C}[e]) = \psi(\mathbb{C})[\psi(e)]$; **3. adequate** if $\psi(e_1) \leq_2 \psi(e_2) \implies e_1 \leq_1 e_2$; 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 $\Lambda a.e$ and type applications $(e\ \tau)$ are not permitted. The normal order reduction for L_{LR} is defined as for L_F where the rule (Lbeta) is not used. The semantics of L_{LR} was investigated e.g. in [20, 21].

▶ **Definition 4.2.** The *translation* ε 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 $\varepsilon(\Lambda a.e) := \varepsilon(e)$, $\varepsilon(e\ \tau) := \varepsilon(e)$, and on the other constructs ε acts homomorphically, removing type labels and types.

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

▶ **Lemma 4.3.** *The translation ε is convergence equivalent and compositional.*

(gc)	$(\texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e) \to e$ if no x_i occurs free in e
(gc)	$(\texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n, y_1 = e'_1, \ldots, y_m = e'_m\ \texttt{in}\ e)$ $\to (\texttt{letrec}\ y_1 = e'_1, \ldots, y_m = e'_m\ \texttt{in}\ e)$ if no x_i occurs free in e nor in any e'_j
(gcp)	$(\texttt{letrec}\ x = e, \mathit{Env}\ \texttt{in}\ \mathbb{C}[x]) \to (\texttt{letrec}\ x = e, \mathit{Env}\ \texttt{in}\ \mathbb{C}[e])$
(gcp)	$(\texttt{letrec}\ x = e_1, y = \mathbb{C}[x], \mathit{Env}\ \texttt{in}\ e_2) \to (\texttt{letrec}\ x = e_1, y = \mathbb{C}[e_1], \mathit{Env}\ \texttt{in}\ e_2)$
(gcp)	$(\texttt{letrec}\ x = \mathbb{C}[x], \mathit{Env}\ \texttt{in}\ e) \to (\texttt{letrec}\ x = \mathbb{C}[\mathbb{C}[x]], \mathit{Env}\ \texttt{in}\ e)$

Figure 5 Further program transformations.

This implies that it is also adequate (see for example [17]).

▶ **Corollary 4.4.** *The translation ε is adequate, which means that equivalences from L_{LR} 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]).

▶ **Corollary 4.5.** *The reductions rules from Figs. 4 and 5 are correct program transformations in L_F, 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].

▶ **Definition 4.6.** For a polymorphic type ρ and e_1, e_2 of type ρ, let $e_1 \leq_{F,R} e_2$ hold if for all reduction contexts $\mathbb{R}[\cdot : \rho]$ such that $\mathbb{R}[e_1], \mathbb{R}[e_2]$ are closed: $\mathbb{R}[e_1]\downarrow \implies \mathbb{R}[e_2]\downarrow$.

▶ **Proposition 4.7** (Context Lemma for L_F). *Let ρ be a polymorphic type and e_1, e_2 be of type ρ. 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_P in Sect. 5).

The context lemma 4.7 immediately implies:

▶ **Corollary 4.8.** *If e_1, e_2 are closed expressions of equal type with $e_1\Uparrow, e_2\Uparrow$, 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 $\preccurlyeq_\omega := \bigcap_{n \geq 0} \preccurlyeq_n$ where for $n \geq 0$, $\preccurlyeq_n$ is defined on closed L_F-expressions e_1, e_2 of the same type as follows:

1. $e_1 \preccurlyeq_0 e_2$ is always true.
2. $e_1 \preccurlyeq_n e_2$ 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 τ: $e_1'[\tau/a] \preccurlyeq_{n-1} e_2'[\tau/a]$.
   b. if $e_1{\downarrow}\mathbb{W}[\lambda x : \tau.e_1']$, then
   $$e_2{\downarrow}\mathbb{W}'[\lambda x : \tau.e_2'] \text{ and for all closed } e : \tau\text{: } \mathbb{W}[\lambda x.e_1']\ e \preccurlyeq_{n-1} \mathbb{W}'[\lambda x.e_2']\ e.$$
   c. if $e_1{\downarrow}\mathbb{W}[c\ e_1' \dots e_m']$, then $e_2{\downarrow}\mathbb{W}'[c\ e_1'' \dots e_m'']$ and for all i: $\mathbb{W}[e_i'] \preccurlyeq_{n-1} \mathbb{W}'[e_i'']$.

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_P is a polymorphically typed *call-by-name* lambda-calculus with fixpoint combinators, but no `letrec` (see Sect. 5). In L_P conincidence 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 $T : L_F \to L_P$ 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.** $\preccurlyeq_\omega^o = \leq_F$

As a corollary we show that ε is not fully abstract.

▶ **Proposition 4.11.** *The translation ε is not fully abstract.*

Proof. In L_F the expressions $e_1 = \lambda x : \texttt{Bool}.\texttt{case}_{\texttt{Bool}}\ x\ \texttt{of}\ (\texttt{True} \texttt{->} x)\ (\texttt{False} \texttt{->} x)$ and $e_2 = \lambda x : \texttt{Bool}.x$ 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 $e_1 = \Lambda a.\lambda x : (\texttt{List}\ a).x$ and $e_2 = \Lambda a.\lambda x : (\texttt{List}\ a).\texttt{case}\ x$ of $(\texttt{Nil} \texttt{->} \texttt{Nil})\ (\texttt{Cons}\ y_1\ y_2 \texttt{->} \texttt{Cons}\ y_1\ y_2)$ of polymorphic type $\boldsymbol{\lambda} a.\texttt{List}\ a \to \texttt{List}\ a$ are contextually equivalent.*

Proof. We use inductive similarity. Application to τ yields two monomorphic abstractions. Further application can only be to arguments without WNHF, or with WHNF $\mathbb{W}[\texttt{Nil}]$ or $\mathbb{W}[\texttt{Cons}\ e_1'\ e_2']$. 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 Ψ 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.

$$\begin{array}{lll}
\text{Polymorphic expressions: } u \in \mathit{PExpr}_P & ::= & x : \boldsymbol{\lambda} a.\rho \mid \lambda x : \tau.e \mid (u\ \tau) \mid \Lambda a.u \\
\text{Expressions: } e, e_i \in \mathit{Expr}_P & ::= & x : \rho \mid u \mid (e\ \tau) \mid (e_1\ e_2) \mid (c : \tau\ e_1 \ldots e_{ar(c)}) \\
 & & \mid (\mathtt{case}_K\ e\ \mathtt{of}\ \mathit{alts}) \mid (\mathtt{seq}\ e_1\ e_2) \mid \Psi_{i,n}\overline{x : \rho}.\overline{e} \\
\text{Reduction contexts: } \mathbb{R}_P \in \mathcal{R}_P & ::= & [\cdot] \mid (\mathbb{R}_P\ e) \mid \mathtt{case}_K\ \mathbb{R}_P\ \mathtt{of}\ \mathit{alts} \mid \mathtt{seq}\ \mathbb{R}_P\ e
\end{array}$$

Normal order reduction:

$$\begin{array}{ll}
\text{(rnbeta)} & \mathbb{R}_P[((\lambda x.e_1)\ e_2)] \xrightarrow{P} \mathbb{R}_P[e_1[e_2/x]] \\
\text{(rncase)} & \mathbb{R}_P[\mathtt{case}_K\ (c\ e_1 \ldots e_{ar(c)})\ \mathtt{of}\ \ldots ((c\ x_1 \ldots x_{ar(c)})\ \texttt{->}\ e) \ldots)] \\
 & \quad \xrightarrow{P} e[e_1/x_1, \ldots, e_{ar(c)}/x_{ar(c)}] \\
\text{(rnseq)} & \mathbb{R}_P[\mathtt{seq}\ v\ e] \xrightarrow{P} \mathbb{R}_P[e],\ \text{if } v \text{ is an } L_P\text{- WHNF.} \\
\text{(rntype)} & \mathbb{R}_P[(\Lambda a.e)\ \tau] \xrightarrow{P} \mathbb{R}_P[e[\tau/a]] \\
\text{(rnfix)} & \mathbb{R}_P[\Psi_{i,n}\overline{x}.\overline{e}] \xrightarrow{P} \mathbb{R}_P[(e_i[\Psi_{1,n}\overline{x}.\overline{e}/x_1, \ldots, \Psi_{n,n}\overline{x}.\overline{e}/x_n])]
\end{array}$$

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 Expr_P of L_P-expressions is that of L_F, where `letrec` is removed, see Fig.6. $(\Psi_{i,n}\overline{x : \rho}.\overline{e})$ is a family of multi-fixpoint-operators, where $1 \leq i \leq n$ and where $\overline{x}$ means $x_1, \ldots, x_n$, similarly for e.

The typing rules are according to Fig. 2 with the additional rule for the Ψ-operator:

$$\frac{\text{for } i = 1, \ldots, n\text{: } e_i : \rho_i, x_i : \rho_i}{(\Psi_{i,n}\overline{x : \rho}.\overline{e}) : \rho_i}$$

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 $\mathbb{R}_P$ 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 ρ, but the context itself must have plain type.

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

$$\begin{array}{lll}
\mathtt{length} & := & (\Psi_{1,1}\mathit{len}{:}\mathtt{List}\ a \to \mathtt{Nat}.\Lambda a.\lambda \mathit{xs}{:}\mathtt{List}\ a. \\
 & & \quad \mathtt{case}\ \mathit{xs}\ \mathtt{of}\ (\mathtt{Nil}\ \texttt{->}\ 0)\ (\mathtt{Cons}\ y\ \mathit{ys}\ \texttt{->}\ \mathtt{Succ}\ (\mathit{len}\ a\ \mathit{ys})))
\end{array}$$

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 $T : L_F \to L_P$

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

▶ **Definition 5.2.** The translation $T : L_F \to L_P$ is defined as:
$T(\mathtt{letrec}\ x_1 = e_1; \ldots, x_n = e_n\ \mathtt{in}\ e') := T(e')[(\Psi_{1,n}\overline{x}.\overline{f})/x_1, \ldots, (\Psi_{n,n}\overline{x}.\overline{f})/x_n]$ 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 $T : L_F \to L_P$ 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 ε_P into an untyped call-by-name calculus L_{lcc} in [20] by forgetting the types (and allowing also untyped expressions), and with $\varepsilon_P(\Psi_{1,1}x.s) = \mathbf{Y}\ (\varepsilon_P(\lambda x.s))$, where $\mathbf{Y}$ is the untyped fixpoint combinator in L_{lcc}. Similarly, we can translate $\Psi_{i,n}(\ldots)$ using the multi fixpoint combinators in [20].

▶ **Proposition 5.4.** *The translation $\varepsilon_P : L_P \to L_{lcc}$ 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 ε_P can be used. ◀

5.3 Applicative Simulation in L_P

In the following we use binary relations η on closed expressions (of the same type). We need closing substitutions σ 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 η^o of η is the relation on open expressions, where $e_1\ \eta^o\ e_2$ is valid iff for all substitutions σ where $\sigma(e_1), \sigma(e_2)$ are closed, the relation $\sigma(e_1)\ \eta\ \sigma(e_2)$ holds. We will also use the restriction of a binary relation η to closed expressions which is denoted as η^c.

▶ **Definition 5.5** (Applicative Similarity in L_P). Let η 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

- $e_1{\downarrow_P}\lambda x.e'_1 \implies \big(\ e_2{\downarrow_P}\lambda x.e'_2$ and $e'_1\ \eta^o\ e'_2\ \big)$
- $e_1{\downarrow_P}(c\ e'_1 \ldots e'_n) \implies \big(\ e_2{\downarrow_P}(c\ e''_1 \ldots e''_n)$ and the relation $e'_i\ \eta\ e''_i$ holds for all $i\ \big)$
- $e_1{\downarrow_P}\Lambda a.e'_1 \implies \big(\ e_2{\downarrow_P}\Lambda a.e'_2$ and $e'_1\ \eta^o\ e'_2\ \big)$

Applicative similarity $\preccurlyeq_P$ is defined as the greatest fixpoint of the operator F_P. *Mutual similarity* $\simeq_P$ is defined as $e_1 \simeq_P e_2$ iff $e_1 \preccurlyeq_P e_2 \wedge e_2 \preccurlyeq_P e_1$.

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

Howe's method [5, 6] to show that $\preccurlyeq_P$ is a pre-congruence and equal to $\leq_P^c$ 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 $\leq_P^c \subseteq \preccurlyeq_P$ by proving that contextual equivalence in L_P satisfies the fixpoint conditions of F_P and then we use coinduction. By the properties of $\preccurlyeq_P$ this implies that $\leq_P \subseteq \preccurlyeq_P$ also holds for open expressions.

▶ **Theorem 5.6.** $\leq_P^c = \preccurlyeq_P$*, and* $\leq_P = \preccurlyeq_P^o$.

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 $e_1{\downarrow_P}\lambda x : \tau.e'_1$, then clearly $e_2{\downarrow_P}\lambda x : \tau.e'_2$. 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 $e_1{\downarrow_P}(c\ e'_1 \ldots e'_n)$, then $e_2{\downarrow_P} v_2$. Since $\leq_P$ is a precongruence, reduction is correct, and using simple contexts like `case` $[\cdot]$ `of` $((c\ x_1 \ldots x_n)$ `->` `True`$) \ldots (p'$ `->` $\bot)$ and `case` $[\cdot]$ `of` $((c\ x_1 \ldots x_n)$ `->` $x_i) \ldots (p'$ `->` $\bot)$, we see that v_2 is of the form $(c\ e''_1 \ldots e''_n)$, where $e'_i \leq_P e''_i$ 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 $([\cdot]\ \tau)$. ◀

5.4 Inductive Similarity For L_P

We define the inductive version of applicative similarity in L_P:

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

1. $e_1 \preccurlyeq_{P,0} e_2$ is always true.
2. $e_1 \preccurlyeq_{P,n} e_2$ 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 τ: $e'_1[\tau/a] \preccurlyeq_{P,n-1} e'_2[\tau/a]$.
   b. if $e_1{\downarrow}\lambda x : \tau.e'_1$ then $e_2{\downarrow}\lambda x : \tau.e'_2$ and for all closed $e : \tau$: $e'_1[e/x] \preccurlyeq_{P,n-1} e'_2[e/x]$.
   c. if $e_1{\downarrow}(c\ e'_1 \ldots e'_m)$ then $e_2{\downarrow}(c\ e''_1 \ldots e''_m)$ and for all i: $e'_i \preccurlyeq_{P,n-1} e''_i$.

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

▶ **Theorem 5.8.** $\preccurlyeq^o_{P,\omega} = \leq_P$

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 $\mathbb{R}_P$, and all (well-typed) substitutions σ where $\sigma(e_1)$ and $\sigma(e_2)$ are closed: $\mathbb{R}_P[\sigma(e_1)]{\downarrow} \implies \mathbb{R}_P[\sigma(e_2)]{\downarrow}$.

▶ **Theorem 5.9.** $\leq_{ciu} = \leq_P$

Proof. We apply the knowledge about applicative simulation $\preccurlyeq$: If $e_1 \preccurlyeq^o e_2$, then for all σ where $\sigma(e_1), \sigma(e_2)$ are closed: $\sigma(e_1) \preccurlyeq \sigma(e_2)$. Since we already know that $\preccurlyeq$ is a pre-congruence, we also obtain $\mathbb{R}_P[\sigma(e_1)] \preccurlyeq \mathbb{R}_P[\sigma(e_2)]$, and so $\mathbb{R}_P[\sigma(e_1)]{\downarrow} \implies \mathbb{R}_P[\sigma(e_2)]{\downarrow}$.

We show that the ciu-relation implies $\preccurlyeq^o$: 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 $[\cdot]\ e'_1 \ldots\ e'_m$ 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 $e_1 \preccurlyeq e_2$. The ciu-relation for non-closed e_1, e_2 is equivalent to the open extension of $\preccurlyeq$, i.e., it implies $e_1 \preccurlyeq^o e_2$, and thus we have the equality: $\preccurlyeq^o = \leq_{ciu}$. ◀

6 Conclusion

Using a system-F-like extension of untyped extended lambda-calculi with case, constructors, and seq, and call-by-need and call-byname 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 `amb` appears unrealistic, since there are counterexamples for combinations of `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

1 S. Abramsky. The lazy lambda calculus. In D. A. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1990.

2 Z. M. Ariola and J. W. Klop. Lambda calculus with explicit recursion. *Inform. and Comput.*, 139(2):154–233, 1997.

3 J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*. CUP, 1994.

4 Haskell-community. The Haskell Programming Language, 2014. `http://www.haskell.org`.

5 D. Howe. Equality in lazy computation systems. In *LICS'89*, pages 198–203, 1989.

6 D. Howe. Proving congruence of bisimulation in functional programming languages. *Inform. and Comput.*, 124(2):103–112, 1996.

7 J. Launchbury. A natural semantics for lazy evaluation. In *POPL'93*, pages 144–154. ACM, 1993.

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

9 S. Peyton Jones. *Haskell 98 language and libraries: the Revised Report*. CUP, 2003.

10 B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

11 A. M. Pitts. Howe's method for higher-order languages. In *Advanced Topics in Bisimulation and Coinduction*, volume 52 of *Cambridge Tracts in Theoretical Computer Science*, chapter 5, pages 197–232. CUP, November 2011. (chapter 5).

12 D. Sabel and M. 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.

13 D. Sabel and M. Schmidt-Schauß. A contextual semantics for Concurrent Haskell with futures. In *PPDP'11*, pages 101–112, New York, NY, USA, July 2011. ACM.

14 D. Sabel and M. Schmidt-Schauß. Conservative concurrency in Haskell. In *LICS'12*, pages 561–570. IEEE, 2012.

15 D. Sabel, M. Schmidt-Schauß, and F. Harwath. Reasoning about contextual equivalence: From untyped to polymorphically typed calculi. In *INFORMATIK 2009 (ATPS'09)*, volume 154 of *LNI*, pages 369; 2931–45, 2009.

16 M. Schmidt-Schauß. Correctness of copy in calculi with letrec. In *RTA'08*, volume 4533 of *LNCS*, pages 329–343. Springer, 2007.

17 M. Schmidt-Schauß, E. Machkasova, and D. Sabel. Extending Abramsky's lazy lambda calculus: (non)-conservativity of embeddings. In *RTA'13*, volume 21 of *LIPIcs*, pages 239–254, Dagstuhl, Germany, 2013. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

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

19 M. Schmidt-Schauß, D. Sabel, and E. Machkasova. Counterexamples to applicative simulation and extensionality in non-deterministic call-by-need lambda-calculi with letrec. *Inf. Process. Lett.*, 111(14):711–716, 2011.

20 M. Schmidt-Schauß, D. Sabel, and E. Machkasova. Simulation in the call-by-need lambda-calculus with letrec, case, constructors, and seq. Frank report 49, Goethe-Universität Frankfurt, 2012.

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

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

23 D. Vytiniotis and S. Peyton Jones. Evidence Normalization in System FC (Invited Talk). In *RTA'13*, volume 21 of *LIPIcs*, pages 20–38, Dagstuhl, Germany, 2013. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.