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

*Manfred Schmidt-Schauss* and *David Sabel*

Goethe-Universität Frankfurt, Institut für Informatik, Germany

WPTE 14
Contextual equivalence: higher order calculi

The Issue

Develop tools and methods for detecting/proving equivalence of two program expressions $s, t$:

\[ s \sim t? \]
The Issue

Develop tools and methods for detecting/proving equivalence of two program expressions $s, t$:

$$s \sim t$$

Untyped and monomorphically typed; contextual equivalence

$$s \sim t \text{ iff } \forall C[.] : \ C[s] \Downarrow \iff C[t] \Downarrow$$

$C[.]$: a program context

$s \Downarrow$: $s \rightarrow^* \text{ WHNF}$

$\rightarrow^*$ is the reduction of the calculus (with a strategy)
Work Done and Results: untyped

**Known**

Well-analyzed: **Calculus LR**: lambda-calculus with case, constructors, seq, let(rec) and call-by-need strategy

(see e.g. [S., Sabel, Machkasova, Simulation..., RTA, 2010; S., Sabel, Machkasova, 2012, TechRep])

Methods: Context Lemmata, diagram method, applicative similarity, ...
Known

Well-analyzed: Calculus LR: lambda-calculus with case, constructors, seq, let(rec) and call-by-need strategy
(see e.g. [S., Sabel, Machkasova, Simulation..., RTA, 2010; S., Sabel, Machkasova, 2012, TechRep])

Methods: Context Lemmata, diagram method, applicative similarity, ...

Desired Extension

Polymorphically Typed Variant of LR
Methods to be adapted to polymorphic types.

Would provide a better semantic model for Haskell and more expressive tools for proving correctness of program transformations.
Lifting from Untyped

Lifted equivalence from untyped covers a lot of equivalences, but is insufficient:

\[ \lambda x.\text{case } x \text{ of } \text{Nil} \rightarrow \text{Nil}; y:ys \rightarrow y:ys \sim \lambda x::(\text{List } a).x \]

- Holds in polymorphic calculi
- Not valid in an untyped calculus, for example for \((\lambda y.y)/x\)
- Equivalence cannot be expressed in a monomorphic calculus.
types as labels of expressions

Too complex:
- complex scoping conditions of type variables which are in labels. (see [Sabel, S., Harwath, ATPS, 2009])
- a (nontrivial) type-modification during reduction.
- complex extra conditions in the context lemmas.

Monomorphising

Failed:
Does not work in the presence of letrec-bindings. It appears to require infinitely many differently typed copies.
Approach

Use a system-F like modelling of polymorphic types as extension of LR

(Haskell-modelling:
Vytiniotis and S. Peyton Jones. . . . System FC. RTA 2013)

This make types a part of the language and permits also beta-reduction for types
Syntax of $L_F$

<table>
<thead>
<tr>
<th>Type variables:</th>
<th>$a \in \mathcal{A}$ where $\mathcal{A}$ is the set of type variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Term variables:</td>
<td>$x, x_i \in \mathcal{X}$ where $\mathcal{X}$ is the set of term variables</td>
</tr>
<tr>
<td>Types:</td>
<td>$\tau \in \text{Typ} := a \mid (\tau_1 \rightarrow \tau_2) \mid (K \tau_1 \ldots \tau_{\text{ar}(K)})$</td>
</tr>
<tr>
<td>Polymorphic types:</td>
<td>$\rho \in \text{PTyp} := \tau \mid \lambda a.\rho$</td>
</tr>
<tr>
<td>Expressions:</td>
<td>$e \in \text{Expr}<em>F := x : \rho \mid u \mid (e \tau) \mid (e_1 e_2) \mid (c : \tau e_1 \ldots e</em>{\text{ar}(c)})$</td>
</tr>
<tr>
<td></td>
<td>$\mid (\text{seq } e_1 e_2) \mid \text{letrec } x_1 : \rho_1 = e_1, \ldots, x_n : \rho_n = e_n \text{ in } e$</td>
</tr>
<tr>
<td></td>
<td>$\mid \text{case}_K e \text{ of } (p_1 \rightarrow e_1) \ldots (p_n \rightarrow e_n)$</td>
</tr>
<tr>
<td></td>
<td>where there is a pattern $p$ for every constructor in $D_K$</td>
</tr>
</tbody>
</table>

| Polymorphic expressions: | $u \in \text{PExpr}_F := x : \lambda a.\rho \mid \lambda x : \tau.e \mid (u \tau) \mid \Lambda a.u$ |
| Patterns:                | $p := (c : \tau x_1 : \tau_1 \ldots x_{\text{ar}(c)} : \tau_{\text{ar}(c)})$ |
|                          | where $x_i$ are different term variables |
Example for $L_F$

**map**

\[
(letrec \ map = \lambda f.\lambda xs.\text{case } xs \text{ of } \\
\quad \text{Nil } \rightarrow \text{ Nil; } \\
\quad (y : ys) \rightarrow f \ y : map \ f \ ys \\
\text{in } map)
\]

**polymorphic map in $L_F$**

\[
(letrec \ map :: \lambda a.\lambda b. (a \rightarrow b) \rightarrow (List \ a) \rightarrow (List \ b) \\
\quad = \Lambda a.\Lambda b. \lambda f :: (a \rightarrow b).\lambda xs :: (List \ a).\text{case } xs \text{ of } \\
\quad \quad \text{Nil :: (List } a) \rightarrow \text{ Nil :: (List } b); \\
\quad \quad \text{Cons :: (List } a \ (y :: a) (ys :: List } a) \\
\quad \quad \rightarrow \text{ Cons :: (List } b \ (f \ y) (map \ a \ b \ f \ ys) \\
\quad \text{in } map)
\]
Normal-Order Reductions of $L_F$

\[(\text{Lbeta}) \quad ((\lambda x : \tau. e_1) \ e_2) \rightarrow \text{letrec } x : \tau = e_2 \ \text{in} \ e_1\]

\[(\text{Lbeta}) \quad ((\Lambda a. u) \ \tau) \rightarrow u[\tau/a]\]

\[(\text{cp-in}) \quad \text{letrec } x = v, Env \ \text{in} \ C[x] \rightarrow \text{letrec } x = v, Env \ \text{in} \ C[v]\]
where $v$ is a polymorphic abstraction, or a cv-expression

\[(\text{cp-e}) \quad \ldots\]

\[(\text{cpcx-in}) \quad \text{letrec } x = (c : \tau \ e_1 \ldots e_n), Env \ \text{in} \ C[x]\]
\[\rightarrow \text{letrec } x = (c : \tau \ x_1 \ldots x_n), x_1 : \tau_1 = e_1, \ldots, x_n : \tau_n = e_n, Env\]
\[\ \text{in} \ C[(c \ x_1 \ldots x_n)]\]
where the types $\tau_i$ are computed as the type of $e_i$

\[(\text{cpcx-e}) \quad \ldots\]

\[(\text{case}) \quad \text{(case} \ (c \ e_1 \ldots e_n) \ \text{of} \ \ldots ((c \ y_1 : \tau_1 \ldots y_n : \tau_n) \rightarrow e) \ldots)\]
\[\rightarrow \text{letrec } y_1 : \tau_1 = e_1, \ldots, y_n : \tau_n = e_n \ \text{in} \ e\]
\[\ldots\]
**Contextual Equality:** \( L_F \)

*Contextual preorder* \( \leq_F \) and *contextual equivalence* \( \sim_F \)

For \( e_1, e_2 \) of (polymorphic) type \( \rho \):

\[
e_1 \leq_F e_2 \quad \text{iff} \quad \forall C[\cdot : \rho] : \tau : \text{If } C[e_1] \text{ and } C[e_2] \text{ are closed, then } (C[e_1]\downarrow \implies C[e_2]\downarrow)
\]

\[
e_1 \sim_F e_2 \quad \text{iff} \quad e_1 \leq_F e_2 \land e_2 \leq_F e_1
\]

where weak head normal forms are:

\[
\Lambda a_1\ldots.\Lambda a_n.\lambda x.e, \quad (c \ e_1\ldots e_n),
\]

\[(\text{letrec } \ldots \text{ in } \Lambda a_1\ldots.\Lambda a_n.\lambda x.e), (\text{letrec } \ldots \text{ in } (c \ e_1\ldots e_n))\]

for constructors \( c \).
### Contextual Equality: $L_F$

#### Properties of $\leq_F, \sim_F$

1. $\leq_F$ is a precongruence.
2. $\sim_F$ is a congruence:
   - This is the **equality** on $L_F$.
   - I.e. the notion of correctness of transformations
## Contextual Equality: $L_F$

<table>
<thead>
<tr>
<th>Properties of $\leq_F, \sim_F$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $\leq_F$ is a precongruence.</td>
</tr>
<tr>
<td>2. $\sim_F$ is a congruence:</td>
</tr>
<tr>
<td>This is the <strong>equality</strong> on $L_F$.</td>
</tr>
<tr>
<td>I.e. the notion of correctness of transformations</td>
</tr>
</tbody>
</table>

## Issue

Tools and sufficient conditions for $e \sim_F e'$
Inheriting Equations from Untyped

Translating Lifting from Untyped

\[ \begin{align*}
\text{typed:} & \quad L_F \\
\epsilon & \downarrow \\
\text{untyped:} & \quad L_{LR}
\end{align*} \]

The translation \( \epsilon \) removes types, is adequate, hence transports equations from \( L_{LR} \) into \( L_F \)

already known: correctness of a set of program transformations in the (untyped) call-by-need calculus \( L_{LR} \)

([ S., Sabel, Machkasova, Simulation... , RTA, 2010; S., Sabel, Machkasova, 2012, TechRep])
Inheriting Equations from Untyped

Implications

All reduction rules in $L_F$ are correct. Also several other transformations, like garbage collection (gc), and copying (gcp):

\[
\begin{align*}
(gc) \quad & (\text{letrec } x_1 = e_1, \ldots, x_n = e_n \text{ in } e) \rightarrow e \quad \text{if no } x_i \text{ occurs free in } e \\
(gc) \quad & (\text{letrec } x_1 = e_1, \ldots, x_n = e_n, y_1 = e'_1, \ldots, y_m = e'_m \text{ in } e) \\
& \rightarrow (\text{letrec } y_1 = e'_1, \ldots, y_m = e'_m \text{ in } e) \\
& \quad \text{if no } x_i \text{ occurs free in } e \text{ nor in any } e'_j \\
(gcp) \quad & (\text{letrec } x = e, Env \text{ in } C[x]) \rightarrow (\text{letrec } x = e, Env \text{ in } C[e]) \\
(gcp) \quad & (\text{letrec } x = e_1, y = C[x], Env \text{ in } e_2) \\
& \rightarrow (\text{letrec } x = e_1, y = C[e_1], Env \text{ in } e_2) \\
(gcp) \quad & (\text{letrec } x = C[x], Env \text{ in } e) \rightarrow (\text{letrec } x = C[C[x]], Env \text{ in } e)
\end{align*}
\]
A Context-Lemma in $L_F$

Let $e_1, e_2$ of type $\rho$.
If for all reduction contexts $R[\cdot : \rho]$ such that $R[e_1], R[e_2]$ are closed: $R[e_1] \downarrow \Rightarrow R[e_2] \downarrow$;
Then $e_1 \leq_F e_2$

where reduction contexts $R[\cdot : \rho]$ are the contexts that indicate the call-by-need reduction position.
Applicative Simulation is in general more powerful than a context lemma.

However, in $\mathcal{L}_F$ there is a letrec:
there is no known direct proof for soundness of an applicative simulation in a calculus with letrec like $\mathcal{L}_{LR}$ (or even $\mathcal{L}_F$).

Idea: use the transfer technique for untyped calculi in
[S., Sabel, Machkasova, 2012, TechRep]
also for polymorphic calculi. This requires a fully abstract translation into a call-by-name calculus.
Transferring Equations by Translation

**Translations Lifting from Untyped**

\[ L_F \xrightarrow{T} L_P \]
\[ L_{LR} \xrightarrow{\epsilon} \text{Inftrees} \]
\[ L_{lcc} \xrightarrow{\epsilon_P} \text{Inftrees} \]

- \( T \) translates \( L_F \) into the call-by-name calculus \( L_P \),
- \( \text{letrec} \) is replaced using fixpoint operators
- calculus with infinite trees used in proofs
Transferring Equations by Translation

**Translations Lifting from Untyped**

- **Polym. typed:**
  - $L_F$ → $L_P$
  - $\varepsilon \downarrow$ → $\varepsilon_P$
  - $T$ → Inftrees → $L_{lcc}$

- **Untyped:**
  - $L_{LR}$ → $L_{lcc}$
  - $T_{untyped}$ → Inftrees

- **Proofs in the polymorphic part are analogous to the ones in the untyped scenario.**

Manfred Schmidt-Schauss and David Sabel: Contextual Equivalences in Polymorphically Typed Calculi
The polymorphic call-by-name calculus $L_P$ is like $L_F$, without \texttt{letrec}, but with

- **fixpoint operators** $(\Psi_{i,n} \overline{x} : \rho . \overline{e})$ where $1 \leq i \leq n$ and where $\overline{x}$ means $x_1, \ldots, x_n$, similarly for $\overline{e}$.
- reduction is call-by-name, i.e. copying beta-reduction.
- WHNFs in $L_P$ are (polymorphic) abstractions
  \[ \Lambda a_1 \ldots \Lambda a_n . \lambda x : \rho . e \] and \[ (c \ e_1 \ldots e_n) \]
The $\Psi$-operator(s) model mutually recursive functions: $\Psi_{i,n}$ corresponds to the $i$th function among $n$ mutually recursive functions.

**Reducing the $\Psi$-operator:**

\[
(rnfix) \quad R_P[\Psi_{i,n,x.e}] \xrightarrow{P} R_P[(e_i[\Psi_{1,n,x.e}/x_1, \ldots, \Psi_{n,n,x.e}/x_n])] 
\]
Example: map in $L_P$

**polymorphic map in** $L_P$

$$\Psi_{1,1}(map:: \lambda a.\lambda b.(a \rightarrow b) \rightarrow (List\ a) \rightarrow (List\ b).\Lambda a.\Lambda b.\lambda f:: (a \rightarrow b).\lambda xs:: (List\ a).case\ xs\ of$$

- $Nil:: (List\ a) \rightarrow Nil:: (List\ b)$;
- $Cons:: (List\ a) (y:: a) (ys:: List\ a)$

$$\rightarrow Cons:: (List\ b) (f\ y) (map\ a\ b\ f\ ys))$$
Example: map in $L_P$

**polymorphic map in $L_P$**

$$\Psi_{1,1}(map :: \lambda a. \lambda b. (a \to b) \to (\text{List } a) \to (\text{List } b). \Lambda a. \Lambda b. \lambda f :: (a \to b). \lambda xs :: (\text{List } a). \text{case } xs \text{ of }$$

- Nil :: (List a) $\to$ Nil :: (List b);
- Cons :: (List a) (y :: a) (ys :: List a) $\to$ Cons :: (List b) (f y) (map a b f ys))

Our syntax permits a “polymorphic bot”:

$$\Psi_{1,1} (x_1 :: (\lambda a.a). x_1)$$
Example: map in $L_P$

**polymorphic map in $L_P$**

\[ \Psi_{1,1}(\text{map} :: \lambda a. \lambda b.(a \rightarrow b) \rightarrow (\text{List } a) \rightarrow (\text{List } b). \lambda a. \lambda b. \lambda f :: (a \rightarrow b). \lambda xs :: (\text{List } a). \text{case } xs \text{ of } \]
\[ \quad \text{Nil} :: (\text{List } a) \rightarrow \text{Nil} :: (\text{List } b); \]
\[ \quad \text{Cons} :: (\text{List } a) \ (y :: a) \ (ys :: \text{List } a) \]
\[ \quad \rightarrow \text{Cons} :: (\text{List } b) \ (f \ y) \ (\text{map } a \ b \ f \ ys) \]

Our syntax permits a “polymorphic bot”:
\[ \Psi_{1,1} (x_1 : (\lambda a.a).x_1) \]
\[ \rightarrow x_1[(\Psi_{1,1} (x_1 : (\lambda a.a).x_1))/x_1] = \Psi_{1,1} (x_1 : (\lambda a.a).x_1) \]
Translating $L_F$ into $L_P$ 

**Translation** $T : L_F \rightarrow L_P$

\[
T(\text{letrec } x_1 = e_1; \ldots, x_n = e_n \text{ in } e') := \\
T(e')[(\Psi_{1,n} x. f)/x_1, \ldots, (\Psi_{n,n} x. f)/x_n]
\]

and where $f_i = \lambda x_1, \ldots, x_n. T(e_i)$ for $i = 1, \ldots, n$,

and $T$ is defined homomorphically on other constructs.

**Lemma**

The translation $T$ is fully abstract.

I.e. $e_1 \sim_F e_2$ if and only if $T(e_1) \sim_P T(e_2)$.

(proof is similar as in the untyped case via infinite trees)
Transferring Applicative Similarity

Transfer Steps

1. Soundness and completeness of coinductive similarity in $L_P$.
2. Equivalence of coinductive and inductive applicative similarity.
3. Transfer inductive applicative similarity from $L_P$ into $L_F$ via $T$.
4. Requires full abstractness of the translation $T$.

\[ L_F \xrightarrow{T} \text{Inftrees} \xrightarrow{T} L_P \]
Let $\eta$ be a binary relation on closed $L_P$-expressions, where only expressions 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

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

Applicative similarity $\preceq_P$ is defined as the greatest fixpoint of the operator $F_P$.

Mutual similarity: $e_1 \simeq_P e_2$ iff $e_1 \preceq_P e_2 \land e_2 \preceq_P e_1$.

$e_1 \eta^o e_2$ holds iff $\sigma(e_1) \eta \sigma(e_2)$ for all closing substitutions $\sigma$. 

Manfred Schmidt-Schauss and David Sabel: Contextual Equivalences in Polymorphically Typed Calculi

22/29
Applying Howe's method to $\preceq_P^o$ in $L_P$ shows:

$$\preceq_P^o = \preceq_P.$$

- Howe's method is applicable in $L_P$, since:
- It is call-by-name: reductions substitute instead of sharing expressions.
- There is no letrec:
  (The recursive scope and the reductions rules for letrec would be in conflict with Howe's method.)
We define $\preceq_{P,\omega} := \bigcap_{n \geq 0} \preceq_n$ where for $n \geq 0$, $\preceq_n$ is defined on closed $L_P$-expressions $e_1, e_2$ of the same type as follows:

1. $e_1 \preceq_0 e_2$ is always true.
2. $e_1 \preceq_n e_2$ for $n > 0$ holds if the following conditions hold:
   1. if $e_1 \downarrow \Lambda a. e_1'$, then $e_2 \downarrow \Lambda a. e_2'$, and for all $\tau$: $e_1'[\tau/a] \preceq_{n-1} e_2'[\tau/a]$.
   2. if $e_1 \downarrow \lambda x : \tau. e_1'$, then $e_2 \downarrow \lambda x : \tau. e_2'$ and for all closed $e : \tau$: $(\lambda x. e_1') e \preceq_{n-1} (\lambda x. e_2') e$.
   3. if $e_1 \downarrow c e_1' \ldots e_m'$, then $e_2 \downarrow c e_1'' \ldots e_m''$ and for all $i$: $e_i' \preceq_{n-1} e_i''$. 
Applicative Similarity in $L_P$

**Continuity**

$\preceq_P$ is continuous, since the reduction strategy is deterministic.

**Corollary of continuity**

$\preceq_P,\omega = \preceq_P$

$\preceq^o_P,\omega = \leq_P$
Introduction  The calculus $L_F$  Methods and Results  Applicative Similarity

Applicative Similarity in $L_P$

**Continuity**

$\lesssim_P$ is continuous, since the reduction strategy is deterministic.

**Corollary of continuity**

$\lesssim_{P,\omega} = \lesssim_P$

$\lesssim^{o}_{P,\omega} = \leq_P$

**Backtranslation**

The relation $\lesssim_{P,\omega}$ can be backtranslated (via $T$) into $L_F$, resulting in $\lesssim_{P,\omega}$ which is sound and complete:

**Theorem**  $\lesssim^{o}_{F,\omega} = \leq_F$
Example for Applicative Similarity in $L_F$

**Example**

\[ e_1 = \lambda x : \text{Bool}. \text{case}_\text{Bool} \ x \ \text{of} \ (\text{True} \rightarrow x) \ (\text{False} \rightarrow x) \]
\[ e_2 = \lambda x : \text{Bool}. x \]

Argue that $e_1 \sim_F e_2$: 

We have only to check $\bot$ and closed arguments that are WHNF and without garbage (for $x$):

- argument $\bot$: both expressions diverge.
- argument reduces to $\text{True}$: the result for $e_1, e_2$ is the same.
- argument reduces to $\text{False}$: the result is the same.
Example for Applicative Similarity in $L_F$

\[ e_1 = \lambda x : \text{Bool}. \text{case}_{\text{Bool}} x \text{ of } (\text{True} \rightarrow x) (\text{False} \rightarrow x) \]
\[ e_2 = \lambda x : \text{Bool}. x \]

Argue that $e_1 \sim_F e_2$: 

We have only to check $\bot$ and closed arguments that are WHNF and without garbage (for $x$):

- argument $\bot$: both expression diverge.
- argument reduces to $\text{True}$: the result for $e_1, e_2$ is the same.
- argument reduces to $\text{False}$: the result is the same
Example for a polymorphic equivalence

\[ \Lambda a. \Lambda b. \lambda x :: a. \lambda y :: b . \text{seq} \ x \ y \sim \Lambda a. \Lambda b. \lambda x :: a. \lambda y :: b . \text{seq} \ y \ (\text{seq} \ x \ y) \]

1. Instantiate the type variables \( a, b \).
2. Test for all closed arguments.
3. Comparison requires also correctness of garbage collection (as program transformation).
Introduction The calculus $L_F$

Methods and Results

Applicative Similarity

Laws from Bird’s book

Map, filter and fold laws (polymorphic)

(Without type, letrec, and lambda-decoration)

$\text{map } f \left( \text{map } g \; \text{xs} \right) \sim \text{map } f \left( \text{map } g \; \text{xs} \right)$

$\text{foldr } f \; a \cdot \text{map } g \sim \text{foldr } (f \cdot g) \; a$

Difference to the foundations of Bird’s proofs

- $\text{seq}$ is available in $L_F$
- fully abstract semantics: (contextual semantics) in $L_F$

R. Bird, Introduction to Functional Programming using Haskell, 1998
Conclusion

Results

- Successful modelling polymorphically typed functional core languages, including well-behaved variants of call-by-need and call-by-name variants (with seq).
- Providing a fully abstract contextual semantics
- Co-inductive and inductive applicative simulation are sound and complete in these calculi

Further Work

- Work out the proof sketches.
- More on: induction schemata and Bird’s laws.
- Try modelling the polymorphic core of GHC (or FC?)
- Modelling polymorphic concurrent Haskell?
Backup slides
We define $\trianglelefteq_{F,\omega} := \bigcap_{n \geq 0} \trianglelefteq_n$ where for $n \geq 0$, $\trianglelefteq_n$ is defined on closed $L_F$-expressions $e_1, e_2$ of the same type as follows:

1. $e_1 \trianglelefteq_0 e_2$ is always true.
2. $e_1 \trianglelefteq_n e_2$ for $n > 0$ holds if the following conditions hold:
   1. if $e_1 \Downarrow \lambda a . e'_1$, then $e_2 \Downarrow \lambda a . e'_2$, and for all $\tau$:
      $e'_1[\tau/a] \trianglelefteq_{n-1} e'_2[\tau/a]$.
   2. if $e_1 \Downarrow W[\lambda x : \tau . e'_1]$, then $e_2 \Downarrow W'[\lambda x : \tau . e'_2]$
      and for all closed $e : \tau$: $W[\lambda x . e'_1] e \trianglelefteq_{n-1} W'[\lambda x . e'_2] e$.
   3. 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 $i$:
      $W[e'_i] \trianglelefteq_{n-1} W'[e''_i]$.

where $W \in WCtxt ::= [\cdot] | (letrec Env in [\cdot]).$
(letrec $y = \Lambda a. (\text{letrec } x = s :: a \text{ in } \lambda z. t) \text{ in } C[y \; \tau])$ 
→ (untyped) 
(\text{letrec } y = \lambda z. t; x = s \text{ in } C[y \; \tau])

(\text{letrec } y = \Lambda a. (\text{letrec } x = s :: a \text{ in } \lambda z. t) \text{ in } C[y \; \tau])
→ 
\text{letrec } y = \Lambda a. \text{letrec } x = s :: a \text{ in } \lambda z. t \text{ in} 

→ 
\text{letrec } y = \Lambda a. \text{letrec } x = s :: a \text{ in } \lambda z. t \text{ in} 
\quad C[(\text{letrec } x = s :: \tau \text{ in } \lambda z. t)]
Sharing in polymorphic functions: a Comment

1. 
\[(\text{letrec } y = \Lambda a.(\text{letrec } x = s :: a \text{ in } \lambda z. t) \text{ in } C[y \tau]) \rightarrow \text{(untyped)}\]
\[(\text{letrec } y = \lambda z.t; x = s \text{ in } C[y \tau])\]
Type scoping violation

2. 
\[(\text{letrec } y = \Lambda a.(\text{letrec } x = s :: a \text{ in } \lambda z. t) \text{ in } C[y \tau]) \rightarrow \text{letrec } y = \Lambda a.\text{letrec } x = s :: a \text{ in } \lambda z. t \text{ in}\]

3. 
\[\rightarrow \text{letrec } y = \Lambda a.\text{letrec } x = s :: a \text{ in } \lambda z. t \text{ in} C[(\text{letrec } x = s :: \tau \text{ in } \lambda z. t)]\]
Unsharing: call-by-name instead of call-by-need.
shared expression independent of type \(\tau\)
Sharing in polymorphic functions: a Comment

1. $(\text{letrec } y = \Lambda a. (\text{letrec } x = s :: a \text{ in } \lambda z. t) \text{ in } C[y \ \tau])$
   $\rightarrow$ (untyped)
   $(\text{letrec } y = \lambda z. t; x = s \text{ in } C[y \ \tau])$
   Type scoping violation

2. $(\text{letrec } y = \Lambda a. (\text{letrec } x = s :: a \text{ in } \lambda z. t) \text{ in } C[y \ \tau])$
   $\rightarrow$
   letrec $y = \Lambda a. \text{letrec } x = s :: a \text{ in } \lambda z. t$ in

3. $\rightarrow$
   letrec $y = \Lambda a. \text{letrec } x = s :: a \text{ in } \lambda z. t$ in
   $C[(\text{letrec } x = s :: \tau \text{ in } \lambda z. t)]$

Unsharing: call-by-name instead of call-by-need.
shared expression independent of type $\tau$

Solution: essentially only $\Lambda a_1 \ldots \Lambda a_n. \lambda xs$ is permitted by syntax