

Alpha-Renaming of Higher-Order Meta-Expressions

David Sabel[†]

Goethe-University Frankfurt am Main, Germany

PPDP 2017, Namur, Belgium

[†]Research supported by the Deutsche Forschungsgemeinschaft (DFG) under grant SA 2908/3-1.

- **automated reasoning on programs and program transformations** w.r.t. operational semantics
- for program calculi with higher-order constructs and **recursive bindings**, e.g.

$$\text{letrec } x_1 = s_1; \dots; x_n = s_n \text{ in } t$$

- extended call-by-need lambda calculi with letrec that model core languages of **lazy functional programming languages** like Haskell

Call-by-need standard reductions & program transformations (excerpt)

(SR,lbeta) $R[(\lambda x.e_1) e_2] \rightarrow R[\text{letrec } x = e_2 \text{ in } e_1]$

(SR,llet) $\text{letrec } Env_1 \text{ in letrec } Env_2 \text{ in } e \rightarrow \text{letrec } Env_1, Env_2 \text{ in } e$

(T,cpx) $T[\text{letrec } x = y, Env \text{ in } C[x]] \rightarrow T[\text{letrec } x = y, Env \text{ in } C[y]]$

(T,gc) $T[\text{letrec } Env \text{ in } e] \rightarrow T[e] \quad \text{if } LetVars(Env) \cap FV(e) = \emptyset$

Meta-syntax must be capable to represent:

- contexts of different classes
- environments Env_i ,

Variables

$x \in \mathbf{Var} ::= X$	(variable meta-variable)
x	(concrete variable)

Expressions

$s \in \mathbf{Expr} ::= S$	(expression meta-variable)
$\mathbf{var} \ x$	(variable)
$(f \ r_1 \ \dots \ r_{ar(f)})$	(function application)
where r_i is o_i, s_i , or x_i specified by f	
$D[s]$	(context meta-variable of class $cl(D)$)
$\mathbf{letrec} \ env \ \mathbf{in} \ s$	(letrec-expression)
$o \in \mathbf{HExpr}^n ::= x_1 \ \dots \ x_n \cdot s$	(higher-order expression)

Environments:

$env \in \mathbf{Env} ::= \emptyset$	(empty environment)
$E; env$	(environment meta-variable)
$x=s; env$	(binding)

Call-by-need standard reductions & program transformations (excerpt)

(SR,lbeta) $R[(\lambda x.e_1) e_2] \rightarrow R[\text{letrec } x = e_2 \text{ in } e_1]$

(SR,llet) $\text{letrec } Env_1 \text{ in letrec } Env_2 \text{ in } e \rightarrow \text{letrec } Env_1, Env_2 \text{ in } e$

(T,cpx) $T[\text{letrec } x = y, Env \text{ in } C[x]] \rightarrow T[\text{letrec } x = y, Env \text{ in } C[y]]$

(T,gc) $T[\text{letrec } Env \text{ in } e] \rightarrow T[e] \quad \text{if } LetVars(Env) \cap FV(e) = \emptyset$

restrictions on scoping and emptiness have to be respected, e.g.:

- (gc): Env must not be empty; side condition on variables
- (llet): $FV(Env_1) \cap LetVars(Env_2) = \emptyset$
- (cpx): x, y are not captured by C in $C[x]$

- A **constraint tuple** $\Delta = (\Delta_1, \Delta_2, \Delta_3)$ consists of
 - Δ_1 : set of context variables (non-empty context constraint)
 - Δ_2 : set of environment variables (non-empty environment constraint)
 - Δ_3 : set of pairs (s, d) (non-capture constraint, NCC)
(s an expression, d a context)
- Ground substitution ρ **satisfies** $(\Delta_1, \Delta_2, \Delta_3)$ iff
 - $\rho(D) \neq [\cdot]$ for all $D \in \Delta_1$
 - $\rho(E) \neq \emptyset$ for all $E \in \Delta_2$
 - hole of $\rho(d)$ does not capture variables of $\rho(s)$, for all $(s, d) \in \Delta_3$

- A **constraint tuple** $\Delta = (\Delta_1, \Delta_2, \Delta_3)$ consists of
 - Δ_1 : set of context variables (non-empty context constraint)
 - Δ_2 : set of environment variables (non-empty environment constraint)
 - Δ_3 : set of pairs (s, d) (non-capture constraint, NCC)
(s an expression, d a context)
- Ground substitution ρ **satisfies** $(\Delta_1, \Delta_2, \Delta_3)$ iff
 - $\rho(D) \neq [\cdot]$ for all $D \in \Delta_1$
 - $\rho(E) \neq \emptyset$ for all $E \in \Delta_2$
 - hole of $\rho(d)$ does not capture variables of $\rho(s)$, for all $(s, d) \in \Delta_3$
- A pair (s, Δ) is called a **constrained expression**

$$\text{sem}(s, \Delta) = \{\rho(s) \mid \rho(s) \text{ fulfills LVC and } \rho \text{ satisfies } \Delta\}$$

(LVC = let variable convention, binders of the same environment are different)

Program transformation T is **correct** iff

$$\forall \ell \rightarrow r \in T: \forall \text{ contexts } C: C[\ell] \downarrow \iff C[r] \downarrow$$

where $\downarrow =$ successful evaluation w.r.t. standard reduction

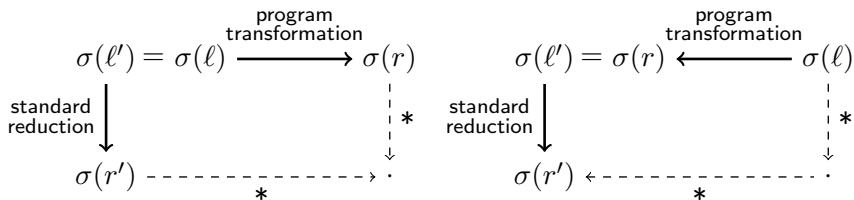
Program transformation T is **correct** iff

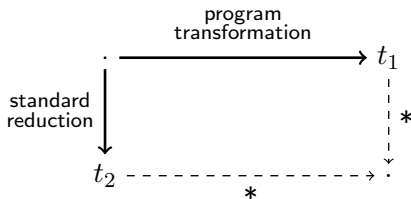
$$\forall \ell \rightarrow r \in T: \forall \text{ contexts } C: C[\ell] \downarrow \iff C[r] \downarrow$$

where $\downarrow =$ successful evaluation w.r.t. standard reduction

Diagram method to show correctness of transformations:

- Compute overlaps between standard reductions and program transformations (unification, see [SSS16, PPDP])
- Join the overlaps \Rightarrow forking and commuting diagrams (Meta-rewriting and matching [Sab17, UNIF])
- Induction using the diagrams (automatable, see [RSSS12, IJCAR])





- t_1, t_2 are **meta-expressions** restricted by **constraints** ∇
- computing joins $\xrightarrow{*}$ requires **abstract rewriting** by rewrite rules $\ell \rightarrow_{\Delta} r$ with Δ restricting ℓ and r
- applying rewrite rules: match the rule and show that the given constraints imply the needed constraints

$$(t, \nabla) \rightarrow (\sigma(r), \nabla \cup \sigma(\Delta)) \quad \text{if } \ell \rightarrow_{\Delta} r, t = \sigma(\ell), \text{ and } \nabla \implies \sigma(\Delta)$$

Example: Overlap (SR,llet) with (T,llet)

(T,llet) $T[\text{letrec } E \text{ in letrec } E' \text{ in } S] \rightarrow T[\text{letrec } E; E' \text{ in } S]$
where an NCC must hold s.t. $\text{LetVars}(E') \cap \text{Vars}(E) = \emptyset$

letrec E_1 in
 letrec E_2 in
 letrec E_3 in S

SR,llet



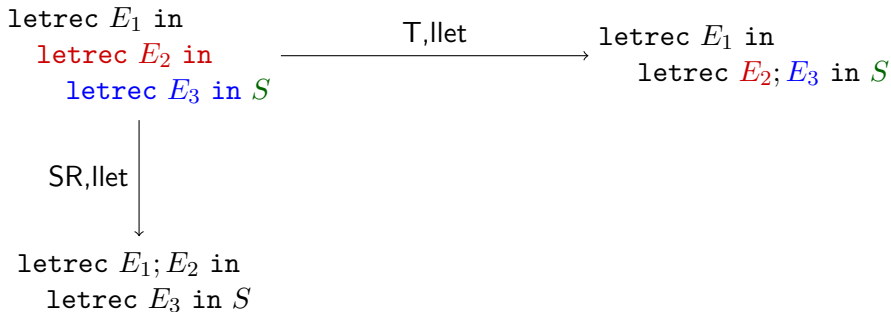
letrec $E_1; E_2$ in
 letrec E_3 in S

Given constraints:

- $\text{LetVars}(E_2) \cap \text{Vars}(E_1) = \emptyset$

Example: Overlap (SR,llet) with (T,llet)

(T,llet) $T[\text{letrec } E \text{ in letrec } E' \text{ in } S] \rightarrow T[\text{letrec } E; E' \text{ in } S]$
where an NCC must hold s.t. $\text{LetVars}(E') \cap \text{Vars}(E) = \emptyset$

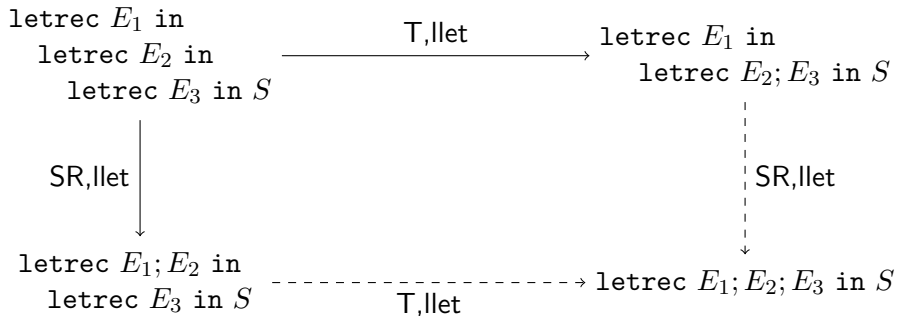


Given constraints:

- $\text{LetVars}(E_2) \cap \text{Vars}(E_1) = \emptyset$
- $\text{LetVars}(E_3) \cap \text{Vars}(E_2) = \emptyset$

Example: Overlap (SR,llet) with (T,llet)

(T,llet) $T[\text{letrec } E \text{ in letrec } E' \text{ in } S] \rightarrow T[\text{letrec } E; E' \text{ in } S]$
where an NCC must hold s.t. $\text{LetVars}(E') \cap \text{Vars}(E) = \emptyset$

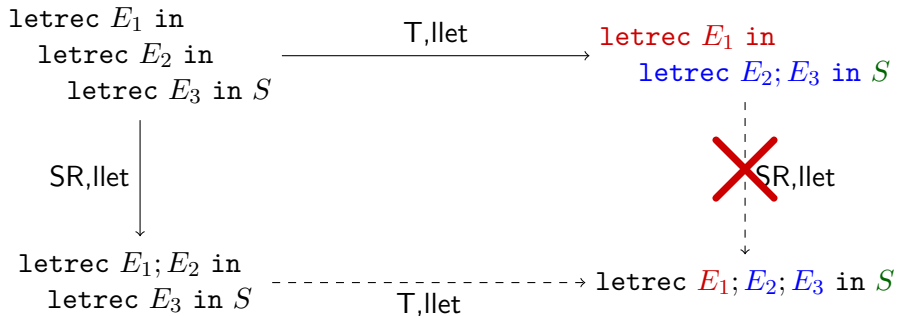


Given constraints:

- $\text{LetVars}(E_2) \cap \text{Vars}(E_1) = \emptyset$
- $\text{LetVars}(E_3) \cap \text{Vars}(E_2) = \emptyset$

Example: Overlap (SR,llet) with (T,llet)

(T,llet) $T[\text{letrec } E \text{ in letrec } E' \text{ in } S] \rightarrow T[\text{letrec } E; E' \text{ in } S]$
where an NCC must hold s.t. $\text{LetVars}(E') \cap \text{Vars}(E) = \emptyset$



Given constraints:

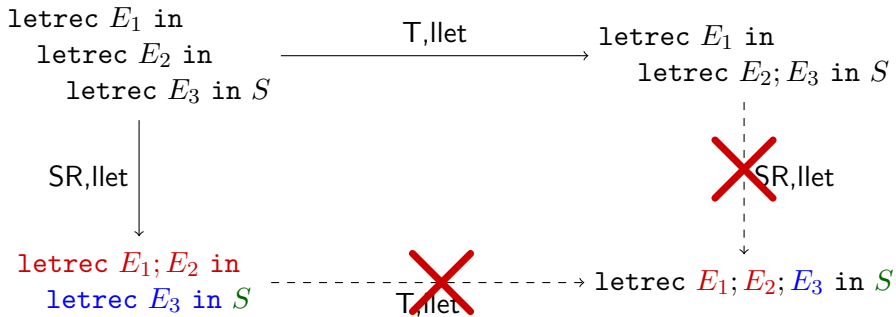
- $\text{LetVars}(E_2) \cap \text{Vars}(E_1) = \emptyset$
- $\text{LetVars}(E_3) \cap \text{Vars}(E_2) = \emptyset$

Needed constraints:

- $\text{LetVars}(E_2; E_3) \cap \text{Vars}(E_1) = \emptyset$

Example: Overlap (SR,llet) with (T,llet)

$(T, \text{llet}) \ T[\text{letrec } E \text{ in letrec } E' \text{ in } S] \rightarrow T[\text{letrec } E; E' \text{ in } S]$
where an NCC must hold s.t. $\text{LetVars}(E') \cap \text{Vars}(E) = \emptyset$



Given constraints:

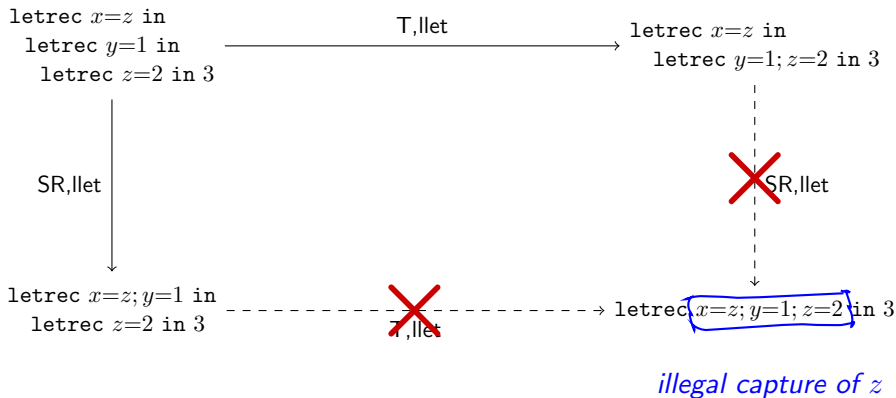
- $\text{LetVars}(E_2) \cap \text{Vars}(E_1) = \emptyset$
- $\text{LetVars}(E_3) \cap \text{Vars}(E_2) = \emptyset$

Needed constraints:

- $\text{LetVars}(E_2; E_3) \cap \text{Vars}(E_1) = \emptyset$
- $\text{LetVars}(E_3) \cap \text{Vars}(E_1; E_2) = \emptyset$



Instance: $E_1 \mapsto x=z$, $E_2 \mapsto y=1$, $E_3 \mapsto z=2$, $S \mapsto 3$



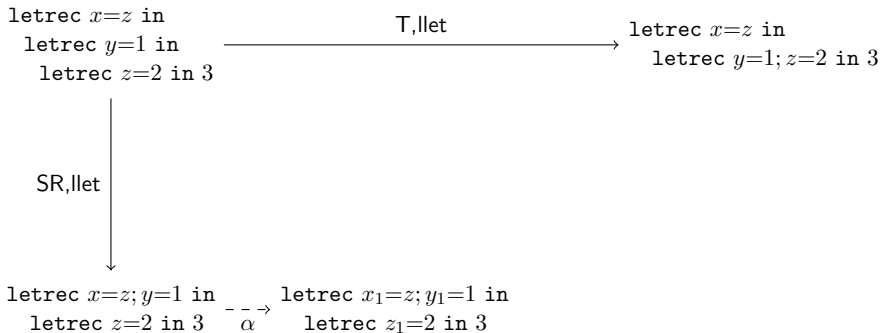
Given constraints:

- $\text{LetVars}(y=1) \cap \text{Vars}(x=z) = \emptyset$
- $\text{LetVars}(z=2) \cap \text{Vars}(y=1) = \emptyset$

Needed constraints:

- $\text{LetVars}(y=1; z=2) \cap \text{Vars}(x=z) = \emptyset$
- $\text{LetVars}(z=2) \cap \text{Vars}(x=z; y=1) = \emptyset$

Instance: $E_1 \mapsto x=z$, $E_2 \mapsto y=1$, $E_3 \mapsto z=2$, $S \mapsto 3$



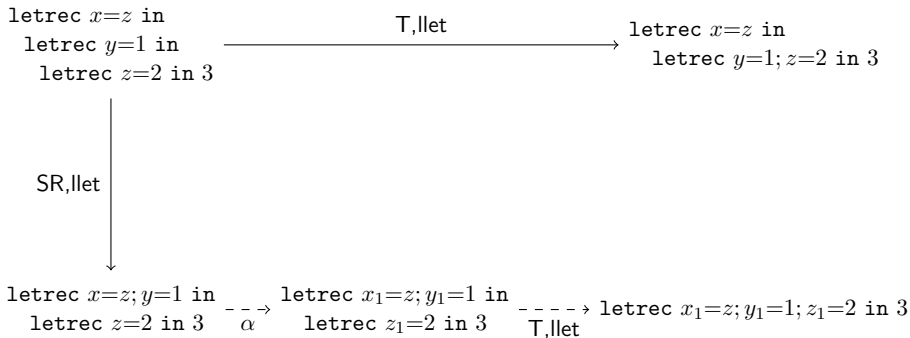
solution: use fresh α -renamings

Given constraints:

- $\text{LetVars}(y=1) \cap \text{Vars}(x=z) = \emptyset$
- $\text{LetVars}(z=2) \cap \text{Vars}(y=1) = \emptyset$

Needed constraints:

Instance: $E_1 \mapsto x=z$, $E_2 \mapsto y=1$, $E_3 \mapsto z=2$, $S \mapsto 3$



solution: use fresh α -renamings

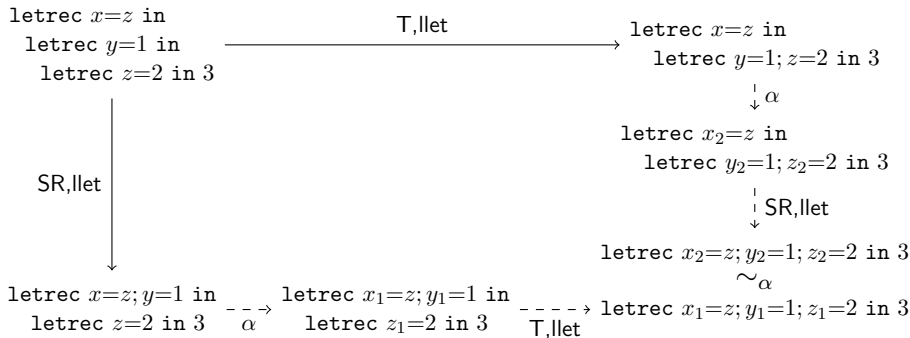
Given constraints:

- $\text{LetVars}(y=1) \cap \text{Vars}(x=z) = \emptyset$
- $\text{LetVars}(z=2) \cap \text{Vars}(y=1) = \emptyset$

Needed constraints:

- $\text{LetVars}(y_1=1; z_1=2) \cap \text{Vars}(x_1=z) = \emptyset$

Instance: $E_1 \mapsto x=z$, $E_2 \mapsto y=1$, $E_3 \mapsto z=2$, $S \mapsto 3$



solution: use fresh α -renamings

Given constraints:

- $\text{LetVars}(y=1) \cap \text{Vars}(x=z) = \emptyset$
- $\text{LetVars}(z=2) \cap \text{Vars}(y=1) = \emptyset$

Needed constraints:

- $\text{LetVars}(y_1=1; z_1=2) \cap \text{Vars}(x_1=z) = \emptyset$
- $\text{LetVars}(z_2=2) \cap \text{Vars}(x_2=z; y_2=1) = \emptyset$

- α -renaming **on the meta-level**
- Instances must fulfill the **distinct variable convention (DVC)**:

Distinct variable convention DVC

A ground LRSX-expression fulfills the DVC iff

- the bound variables are disjoint from the free variables
 - variables on binders are pairwise disjoint
- How to rename meta-variables X, S, E, D ?
⇒ Requires meta-notations for symbolic α -renamings

Variables

$$x \in \mathbf{Var} ::= \langle rc_1, \dots, rc_n \rangle \cdot X \quad \text{(variable meta-variable)}$$

$$| \langle rc_1, \dots, rc_n \rangle \cdot x \quad \text{(concrete variable)}$$

Expressions

$$s \in \mathbf{Expr} ::= \langle \alpha_{S,i}, rc_1, \dots, rc_n \rangle \cdot S \quad \text{(expression meta-variable)}$$

$$| \langle \alpha_{D,i}, rc_1, \dots, rc_n \rangle \cdot D[s] \quad \text{(context meta-variable)}$$

$$| \dots$$

Environments

$$env \in \mathbf{Env} ::= \langle \alpha_{E,i}, rc_1, \dots, rc_n \rangle \cdot E; env \quad \text{(environment meta-variable)}$$

$$| \dots$$

a component $\alpha_{U,i}$ α -renames instances of U

Atomic renaming components

$$rc \in \mathbf{ARC} ::= \alpha_{x,i} \quad \text{(fresh renaming of variable } x)$$

$$| LV(\alpha_{E,i}) \quad \text{(restriction of } \alpha_{E,i} \text{ on } LetVars(E))$$

$$| CV(\alpha_{D,i}) \quad \text{(restriction of } \alpha_{D,i} \text{ on } CapVars(D))$$

- For expression s , an **interpretation** is given by $(\tau \circ \rho)(s)$ where
 - **ground instantiation** ρ instantiates the X -, S -, E -, D -meta variables
 - τ instantiates the renamings $\alpha_{U,i}$ as fresh α -renamings of $\rho(U)$ and:
 - $\tau(\alpha_{x,i})$ is the substitution $\{x \mapsto y_i\}$ where y_i is fresh
 - $\tau(LV(\alpha_{E,i}))$ is the restriction of $\alpha_{E,i}$ to the let-variables of $\rho(E)$
 - $\tau(CV(\alpha_{D,i}))$ is the restriction of $\alpha_{D,i}$ to the capture variables of $\rho(D)$
 - sequences are interpreted as **compositions**
- $sem(s, \Delta) = \{(\tau \circ \rho)(s) \mid s \text{ fulfills the LVC and } \tau \circ \rho \text{ satisfies } \Delta\}$

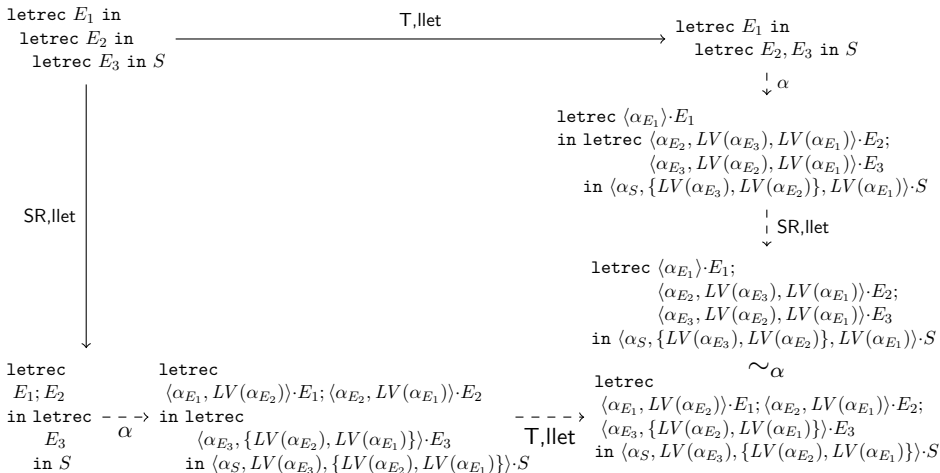
Example: $s = \text{letrec } \langle \alpha_{E,1} \rangle \cdot E \text{ in } \langle \alpha_S, LV(\alpha_{E,1}) \rangle \cdot S$

- Let $\rho = \{E \mapsto x = f(\text{var } x) \lambda x. \text{var } x, S \mapsto \lambda z. \text{var } x\}$
- $(\tau \circ \rho)(s)$

$$= \text{letrec } \tau(\alpha_{E,1})\rho(E) \text{ in } \tau(LV(\alpha_{E,1}))(\tau(\alpha_S)(\rho(S)))$$

$$= \text{letrec } \tau(\alpha_{E,1})(x = f(\text{var } x) \lambda x. \text{var } x) \text{ in } \tau(LV(\alpha_{E,1}))(\tau(\alpha_S)(\lambda z. \text{var } x))$$

$$= \text{letrec } x_1 = f(\text{var } x_1) \lambda x_2. \text{var } x_2 \text{ in } \lambda z_1. \text{var } x_1$$



Tasks: • Perform renaming

letrec E_1 in
letrec E_2 in
letrec E_3 in S

SR, llet

T, llet

letrec E_1 in
letrec E_2, E_3 in S

$\downarrow \alpha$

letrec $\langle \alpha_{E_1} \rangle \cdot E_1$
in letrec $\langle \alpha_{E_2}, LV(\alpha_{E_3}), LV(\alpha_{E_1}) \rangle \cdot E_2;$
 $\langle \alpha_{E_3}, LV(\alpha_{E_2}), LV(\alpha_{E_1}) \rangle \cdot E_3$
in $\langle \alpha_S, \{LV(\alpha_{E_3}), LV(\alpha_{E_2})\}, LV(\alpha_{E_1}) \rangle \cdot S$

\downarrow SR, llet

letrec $\langle \alpha_{E_1} \rangle \cdot E_1;$
 $\langle \alpha_{E_2}, LV(\alpha_{E_3}), LV(\alpha_{E_1}) \rangle \cdot E_2;$
 $\langle \alpha_{E_3}, LV(\alpha_{E_2}), LV(\alpha_{E_1}) \rangle \cdot E_3$
in $\langle \alpha_S, \{LV(\alpha_{E_3}), LV(\alpha_{E_2})\}, LV(\alpha_{E_1}) \rangle \cdot S$

$\sim \alpha$

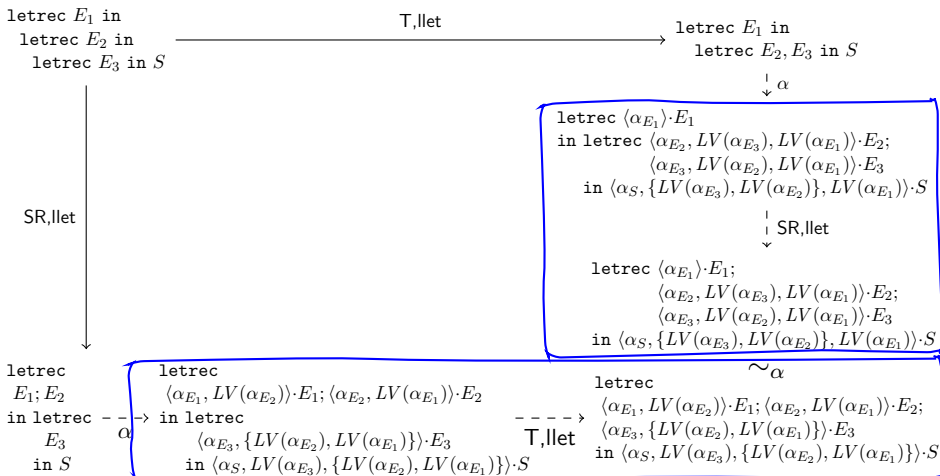
letrec
 $\langle \alpha_{E_1}, LV(\alpha_{E_2}) \rangle \cdot E_1; \langle \alpha_{E_2}, LV(\alpha_{E_1}) \rangle \cdot E_2;$
 $\langle \alpha_{E_3}, \{LV(\alpha_{E_2}), LV(\alpha_{E_1})\} \rangle \cdot E_3$
in $\langle \alpha_S, LV(\alpha_{E_3}), \{LV(\alpha_{E_2}), LV(\alpha_{E_1})\} \rangle \cdot S$

letrec $E_1; E_2$ letrec
 $\langle \alpha_{E_1}, LV(\alpha_{E_2}) \rangle \cdot E_1; \langle \alpha_{E_2}, LV(\alpha_{E_1}) \rangle \cdot E_2$
in letrec E_3 in letrec
 $\langle \alpha_{E_3}, \{LV(\alpha_{E_2}), LV(\alpha_{E_1})\} \rangle \cdot E_3$
in S in $\langle \alpha_S, LV(\alpha_{E_3}), \{LV(\alpha_{E_2}), LV(\alpha_{E_1})\} \rangle \cdot S$

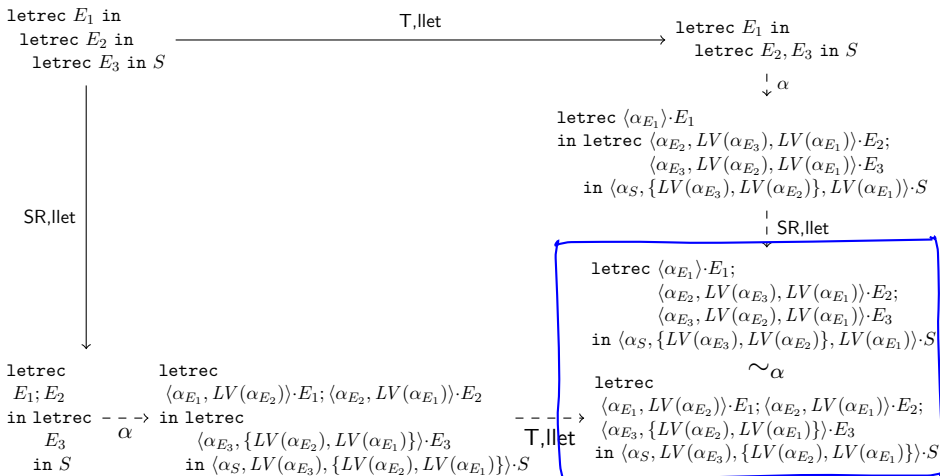
α

T, llet

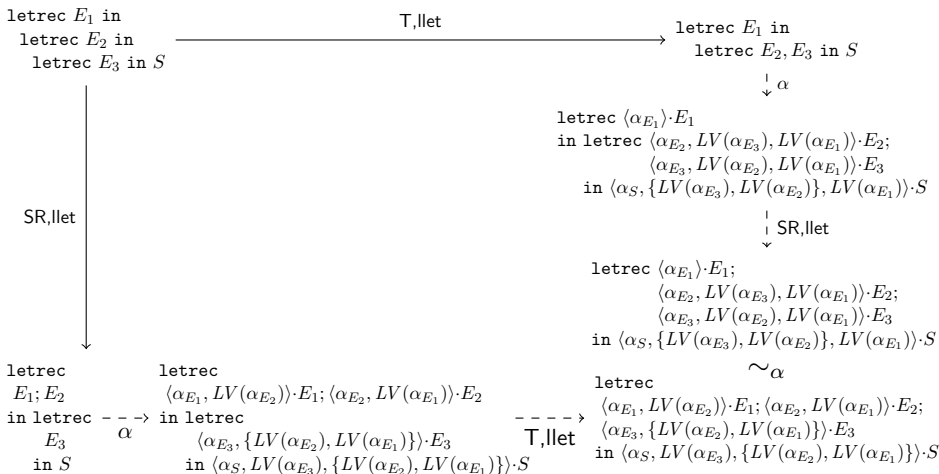
- Tasks:**
- Perform renaming
 - Rewriting for renamed expressions



- Tasks:**
- Perform renaming
 - Rewriting for renamed expressions
 - Check alpha-equivalence



- Tasks:**
- Perform renaming
 - Rewriting for renamed expressions
 - Check alpha-equivalence
 - Simplify renamings
 - Refresh renamings



In the paper we provide:

- An **algorithm to α -rename** $s \in \text{LRSX}$ into $AR(s) \in \text{LRSX}_\alpha$

Proposition (AR preserves the semantics w.r.t. α -equivalence classes)

- For each $t \in \text{sem}(s)$, there exists $t' \in \text{sem}(AR(s))$ such that $t \sim_\alpha t'$.
- For each $t' \in \text{sem}(AR(s))$, there exists $t \in \text{sem}(s)$ such that $t \sim_\alpha t'$.
- All $t' \in \text{sem}(AR(s))$ fulfill the DVC.

In the paper we provide:

- An **algorithm to α -rename** $s \in \text{LRSX}$ into $AR(s) \in \text{LRSX}\alpha$

Proposition (AR preserves the semantics w.r.t. α -equivalence classes)

- For each $t \in \text{sem}(s)$, there exists $t' \in \text{sem}(AR(s))$ such that $t \sim_\alpha t'$.
 - For each $t' \in \text{sem}(AR(s))$, there exists $t \in \text{sem}(s)$ such that $t \sim_\alpha t'$.
 - All $t' \in \text{sem}(AR(s))$ fulfill the DVC.
- A **matching algorithm** to solve $(s, \nabla) \trianglelefteq (s', \Delta)$ where $s \in \text{LRSX}$, $s' \in \text{LRSX}\alpha$, such that for matcher $\sigma: \sigma(s) = s'$ and $\nabla \implies \sigma(\Delta)$.

Theorem: The matching algorithm for $\text{LRSX}\alpha$ is sound.

In the paper we provide:

- An **algorithm to α -rename** $s \in \text{LRSX}$ into $AR(s) \in \text{LRSX}\alpha$

Proposition (AR preserves the semantics w.r.t. α -equivalence classes)

- For each $t \in \text{sem}(s)$, there exists $t' \in \text{sem}(AR(s))$ such that $t \sim_\alpha t'$.
 - For each $t' \in \text{sem}(AR(s))$, there exists $t \in \text{sem}(s)$ such that $t \sim_\alpha t'$.
 - All $t' \in \text{sem}(AR(s))$ fulfill the DVC.
- A **matching algorithm** to solve $(s, \nabla) \trianglelefteq (s', \Delta)$ where $s \in \text{LRSX}$, $s' \in \text{LRSX}\alpha$, such that for matcher $\sigma: \sigma(s) = s'$ and $\nabla \implies \sigma(\Delta)$.

Theorem: The matching algorithm for $\text{LRSX}\alpha$ is sound.

- A **test for extended α -equivalence** for constrained $\text{LRSX}\alpha$ -expressions

Theorem (Soundness of the extended α -equivalence test)

If (s, Δ) and (s', Δ') pass the extended α -equivalence test, then

- for all $\tau, \rho: \tau \circ \rho$ satisfies Δ iff $\tau \circ \rho$ satisfies Δ'
- for all $\tau, \rho: \tau(\rho(s)) \sim_\alpha \tau(\rho(s'))$.

- Simplification removes renaming components that **have no effect**
- Simplification is required for other tasks, for instance before checking α -equivalence
- Example:
In $\langle \dots LV(\alpha_E) \dots \rangle \cdot S$, component $LV(\alpha_E)$ can be removed, if there is an NCC $(S, \text{letrec } E \text{ in } [\cdot])$.
- In the paper: An **inference system to simplify renamings**

Theorem (Soundness of simplification)

If the inference system simplifies (s, Δ) into (s', Δ) , then $sem(s, \Delta) = sem(s', \Delta)$.

Rewriting with copying rules may destroy the DVC:

$$\begin{aligned} & \text{letrec } \alpha_{X,1} \cdot X = \alpha_{S,1} \cdot S \text{ in var } \alpha_{X,1} \cdot X \\ \rightarrow & \text{letrec } \alpha_{X,1} \cdot X = \alpha_{S,1} \cdot S \text{ in var } \alpha_{S,1} \cdot S \end{aligned}$$

Solution: Refresh the α -renaming:

- **Renumber** the α -renamings (using fresh numbers)
- **Add renaming components** if necessary.

Proposition (Refreshing preserves the semantics w.r.t. α -equivalence)

- $t \in \text{sem}(s, \Delta) \implies \exists t' \in \text{sem}(\text{refresh}(s, \Delta))$ with $t \sim_{\alpha} t'$
- $t' \in \text{sem}(\text{refresh}(s, \Delta)) \implies \exists t \in \text{sem}(s, \Delta)$ with $t \sim_{\alpha} t'$.

- **LRSX Tool** available from <http://goethe.link/LRSXTOOL>
- computes diagrams and performs the automated induction
- two **exemplary call-by-need calculi**:
 - L_{need} : letrec-lambda-calculus, analysis of 16 program transformations
 - LR: extension of L_{need} by case, data constructors, Haskell's seq-operator, analysis of 43 program transformations

	# overlaps	# joins	# joins using α -renaming
Calculus L_{need}			
forking	2 242	5 425	93
commuting	3 001	7 273	1 402
Calculus LR			
forking	87 041	391 264	73 601
commuting	107 333	429 104	93 230

Conclusion

- Formalism for representing **symbolic α -renamings**
- **Sound algorithms** for renaming, matching, equivalence test, refreshing, and simplification
- Experiments show that **in about 20%** of the overlaps, **α -renaming helps** for joining

Further work

- Other **applications / meta-languages** for our approach
- **Completeness** of the algorithms