Matching of Meta-Expressions with Recursive Bindings

David Sabel
Goethe-University Frankfurt am Main, Germany

UNIF 2017, Oxford, UK

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

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

\[
\text{letrec } x_1 = s_1 ; \ldots ; x_n = s_n \text{ in } t
\]

- extended call-by-need lambda calculi with letrec that model core languages of **lazy functional programming languages** like Haskell
Program transformation $T$ is **correct** iff

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

where $\downarrow = \text{successful evaluation w.r.t. standard reduction}$
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 (requires unification, see [SSS16, PPDP])
- Join the overlaps $\Rightarrow$ forking and commuting diagrams
- Induction using the diagrams (automatable, see [RSSS12, IJCAR])

\[
\begin{align*}
\sigma(\ell') &= \sigma(\ell) \quad \text{program transformation} \quad \sigma(r) \\
\text{standard reduction} \downarrow \\
\sigma(r') &\rightarrow \quad \star
\end{align*}
\]

\[
\begin{align*}
\sigma(\ell') &= \sigma(r) \quad \text{program transformation} \quad \sigma(\ell) \\
\text{standard reduction} \downarrow \\
\sigma(r') &\leftarrow \quad \star
\end{align*}
\]
Requirements on the Meta-Language

Operational semantics of typical call-by-need calculi (excerpt)

Reduction contexts:
\[ A ::= \cdot | (A \ e) \]
\[ R ::= A | \text{letrec Env in } A | \text{letrec } \{ x_i = A_i[x_{i+1}] \}_{i=1}^{n-1}, x_n = A_n, \text{Env in } A[x_1] \]

Standard-reduction rules and some program transformations:
\[ (\text{SR}, \text{lbeta}) \quad R[(\lambda x. e_1) \ e_2] \to R[\text{letrec } x = e_2 \text{ in } e_1] \]
\[ (\text{SR}, \text{llet}) \quad \text{letrec Env}_1 \text{ in letrec Env}_2 \text{ in } e \to \text{letrec Env}_1, \text{Env}_2 \text{ in } e \]
\[ (\text{T}, \text{cpx}) \quad T[\text{letrec } x = y, \text{Env in } C[x]] \to T[\text{letrec } x = y, \text{Env in } C[y]] \]
\[ (\text{T}, \text{gc}) \quad T[\text{letrec Env in } e] \to T[e] \quad \text{if } \text{LetVars} \text{(Env)} \cap \text{FV}(e) = \emptyset \]
Requirements on the Meta-Language

Operational semantics of typical call-by-need calculi (excerpt)

Reduction contexts:

\[ A ::= [\_] \mid (A \; e) \]

\[ R ::= A \mid \text{letrec } Env \; \text{in } A \mid \text{letrec } \{x_i = A_i[x_{i+1}]\}_{i=1}^{n-1}, x_n = A_n, Env \; \text{in } A[x_1] \]

Standard-reduction rules and some program transformations:

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

\((\text{SR,lllet})\) \[ \text{letrec } Env_1 \; \text{in letrec } Env_2 \; \text{in } e \rightarrow \text{letrec } Env_1, Env_2 \; \text{in } e \]

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

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

Meta-syntax must be capable to represent:

- contexts of different classes
- environments \(Env_i\)
- environment chains \(\{x_i = A_i[x_{i+1}]\}_{i=1}^{n-1}\)
Requirements on the Meta-Language

Operational semantics of typical call-by-need calculi (excerpt)

Reduction contexts:

\[
A ::= [\cdot] \mid (A \ e)
\]

\[
R ::= A \mid \text{letrec } Env \ \text{in } A \mid \text{letrec } \{x_i=A_i[x_{i+1}]\}_{i=1}^{n-1},x_n=A_n,Env \ \text{in } A[x_1]
\]

Standard-reduction rules and some program transformations:

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

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

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

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

Meta-syntax must be capable to represent:

- contexts of different classes
- environments \( Env_i \)
- environment chains \( \{x_i=A_i[x_{i+1}]\}_{i=1}^{n-1} \)
Requirements on the Meta-Language

Operational semantics of typical call-by-need calculi (excerpt)

Reduction contexts:
\[ A ::= [\cdot] \mid (A \ e) \]
\[ R ::= A \mid \text{letrec } Env \text{ in } A \mid \text{letrec } \{x_i = A_i[x_{i+1}]\}_{i=1}^{n-1}, x_n = A_n, \text{ Env in } A[x_1] \]

Standard-reduction rules and some program transformations:
\[(\text{SR,}\ellbeta) \quad R[(\lambda x. e_1) e_2] \to R[\text{letrec } x = e_2 \text{ in } e_1] \]
\[(\text{SR,}\elllet) \quad \text{letrec } Env_1 \text{ in letrec } Env_2 \text{ in } e \to \text{letrec } Env_1, Env_2 \text{ in } e \]
\[ (\text{T,}\text{cpx}) \quad T[\text{letrec } x = y, \text{ Env in } C[x]] \to T[\text{letrec } x = y, \text{ Env in } C[y]] \]
\[ (\text{T,}\text{gc}) \quad T[\text{letrec } Env \text{ in } e] \to T[e] \quad \text{if } \text{LetVars}(Env) \cap \text{FV}(e) = \emptyset \]

Meta-syntax must be capable to represent:
- contexts of different classes
- environments $Env_i$
- environment chains $\{x_i = A_i[x_{i+1}]\}_{i=1}^{n-1}$
Requirements on the Meta-Language

**Operational semantics of typical call-by-need calculi (excerpt)**

Reduction contexts:

\[
A ::= [\cdot] \mid (A \ e)
\]

\[
R ::= A \mid \text{letrec } Env \ \text{in } A \mid \text{letrec } \{x_i = A_i[x_{i+1}]\}_{i=1}^{n-1}, x_n = A_n, Env \ \text{in } A[x_1]
\]

Standard-reduction rules and some program transformations:

\((\text{SR,}l\beta)\) \quad R[(\lambda x.e_1) \ e_2] \to R[\text{letrec } x = e_2 \ \text{in } e_1]

\((\text{SR,}l\text{let})\) \quad \text{letrec } Env_1 \ \text{in } \text{letrec } Env_2 \ \text{in } e \to \text{letrec } Env_1, Env_2 \ \text{in } e

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

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

Meta-syntax must be capable to represent:

- contexts of different classes
- environments \(Env_i\)
- environment chains \(\{x_i = A_i[x_{i+1}]\}_{i=1}^{n-1}\)
Syntax of the Meta-Language LRSX

Variables
\[ x \in \text{Var} ::= X \quad \text{(variable meta-variable)} \]
\[ \quad | \quad x \quad \text{(concrete variable)} \]

Expressions
\[ s \in \text{Expr} ::= S \quad \text{(expression meta-variable)} \]
\[ \quad | \quad \text{var } x \quad \text{(variable)} \]
\[ \quad | \quad (f \ r_1 \ldots r_{ar}(f)) \quad \text{(function application)} \]
\[ \quad \text{where } r_i \text{ is } o_i, s_i, \text{ or } x_i \text{ specified by } f \]
\[ \quad | \quad D[s] \quad \text{(context meta-variable)} \]
\[ \quad | \quad \text{letrec } env \text{ in } s \quad \text{(letrec-expression)} \]

Higher-Order Expressions
\[ o \in \text{HExpr}^n ::= x_1 \ldots x_n.s \quad \text{(higher-order expression)} \]

Environments
\[ env \in \text{Env} ::= \emptyset \quad \text{(empty environment)} \]
\[ \quad | \quad E; env \quad \text{(environment meta-variable)} \]
\[ \quad | \quad Ch[x, s]; env \quad \text{(chain meta-variable)} \]
\[ \quad | \quad x=s; env \quad \text{(binding)} \]

- Context variables \( D \) and \( Ch \)-variables have a context class \( cl(D) \)
- instances of \( Ch[x, s] \): chains \( x=D_1[\text{var } x_1]; x_1=D_2[\text{var } x_2]; \ldots; x_n=D_n[s] \)
  where \( D_i \) are contexts of class \( cl(Ch) \).
Operational semantics of typical call-by-need calculi (excerpt)

Reduction contexts:

\[ A ::= \cdot \mid (A \ e) \]

\[ R ::= A \mid \text{letrec } \text{Env in } A \mid \text{letrec } \{x_i=A_i[x_{i+1}]\}_{i=1}^{n-1}, x_n=A_n, \text{Env in } A[x_1] \]

Standard-reduction rules and some program transformations:

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

(SR,\text{\textit{llet}}) \quad \text{letrec } \text{Env}_1 \ in \ \text{letrec } \text{Env}_2 \ in \ e \rightarrow \text{letrec } \text{Env}_1, \text{Env}_2 \ in \ e

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

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

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

- (gc): \text{Env must not be empty; side condition on variables}
- (llet): \text{FV(Env}_1) \cap \text{LetVars(Env}_2) = \emptyset
- (cpx): x, y are not captured by C in C[x]
Operational semantics of typical call-by-need calculi (excerpt)

Reduction contexts:

\[ A ::= [\cdot] \mid (A \, e) \]

\[ R ::= A \mid \text{letrec } Env \, \text{in } A \mid \text{letrec } \{ x_i = A_i[x_{i+1}] \}_{i=1}^{n-1}, x_n = A_n, Env \, \text{in } A[x_1] \]

Standard-reduction rules and some program transformations:

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

(SR,\lambda let) \quad \text{letrec } Env_1 \, \text{in } \text{letrec } Env_2 \, \text{in } e \rightarrow \text{letrec } Env_1, Env_2 \, \text{in } e

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

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

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

- (gc): \textit{Env} must not be empty; side condition on variables

- (\lambda let): \textit{FV}(Env_1) \cap \text{LetVars}(Env_2) = \emptyset

- (cpx): \textit{x}, \textit{y} are not captured by \textit{C} in \textit{C}[x]
Operational semantics of typical call-by-need calculi (excerpt)

Reduction contexts:
\[ A ::= [\cdot] \mid (A \ e) \]
\[ R ::= A \mid \text{letrec } Env \text{ in } A \mid \text{letrec } \{ x_i = A_i[x_{i+1}] \}_{i=1}^{n-1}, x_n = A_n, Env \text{ in } A[x_1] \]

Standard-reduction rules and some program transformations:
\[(\text{SR,}\ell\beta) \ R[(\lambda x. e_1) \ e_2] \rightarrow R[\text{letrec } x = e_2 \text{ in } e_1] \]
\[(\text{SR,}\ell\ell) \ \text{letrec } Env_1 \text{ in } \text{letrec } Env_2 \text{ in } e \rightarrow \text{letrec } Env_1, Env_2 \text{ in } e \]
\[(\text{T,}\ell\text{px}) \ T[\text{letrec } x = y, Env \text{ in } C[x]] \rightarrow T[\text{letrec } x = y, Env \text{ in } C[y]] \]
\[(\text{T,}\ell\text{g}) \ T[\text{letrec } Env \text{ in } e] \rightarrow T[e] \quad \text{if } \text{LetVars}(Env) \cap FV(e) = \emptyset \]

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

- \((\text{gc})\): \(Env\) must not be empty; side condition on variables
- \((\ell\text{let})\): \(FV(Env_1) \cap \text{LetVars}(Env_2) = \emptyset\)
- \((\text{cpx})\): \(x, y\) are not captured by \(C\) in \(C[x]\)
Operational semantics of typical call-by-need calculi (excerpt)

Reduction contexts:
- \( A ::= [\cdot] \mid (A \; e) \)
- \( R ::= A \mid \text{letrec } Env \; \text{in } A \mid \text{letrec } \{x_i = A_i[x_{i+1}]\}_{i=1}^{n-1}, x_n = A_n, \; \text{Env in } A[x_1] \)

Standard-reduction rules and some program transformations:
- (SR,\( \lambda \beta \)) \( R[(\lambda x.e_1) \; e_2] \to R[\text{letrec } x = e_2 \; \text{in } e_1] \)
- (SR,\( \lambda \text{let} \)) \( \text{letrec } Env_1 \; \text{in } \text{letrec } Env_2 \; \text{in } e \to \text{letrec } Env_1, \; Env_2 \; \text{in } e \)
- (T,\( \text{cpx} \)) \( T[\text{letrec } x = y, \; \text{Env in } C[x]] \to T[\text{letrec } x = y, \; \text{Env in } C[y]] \)
- (T,\( \text{gc} \)) \( T[\text{letrec } \text{Env in } e] \to T[e] \) if \( \text{LetVars}(\text{Env}) \cap FV(e) = \emptyset \)

Restrictions on scoping and emptiness have to be respected, e.g.:
- (\( \text{gc} \)): \( \text{Env} \) must not be empty; side condition on variables
- (\( \lambda \text{let} \)): \( FV(Env_1) \cap \text{LetVars}(Env_2) = \emptyset \)
- (\( \text{cpx} \)): \( x, y \) are not captured by \( C \) in \( C[x] \)
Constrained Expressions

- A **constraint tuple** $\Delta = (\Delta_1, \Delta_2, \Delta_3)$ consists of:
  - $\Delta_1$: set of context variables (non-empty context constraint)
  - $\Delta_2$: set of environment variables (non-empty environment constraint)
  - $\Delta_3$: set of pairs $(s, d)$ ($s$ an expression, $d$ a context) (non-capture constraint)

- Ground substitution $\rho$ **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
$\Delta_1$: set of context variables (non-empty context constraint)
$\Delta_2$: set of environment variables (non-empty environment constraint)
$\Delta_3$: set of pairs $(s, d)$ ($s$ an expression, $d$ a context) (non-capture constraint)

Ground substitution $\rho$ satisfies $(\Delta_1, \Delta_2, \Delta_3)$ iff
- $\rho(D) \neq []$ 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 $(s, \Delta)$ 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)

Example:

$s = \text{letrec } E_1 \text{ in letrec } E_2 \text{ in } S$
$\Delta = (\emptyset, \{E_1, E_2\}, \{(\text{letrec } E_2 \text{ in } S, \text{letrec } E_1 \text{ in } [])\})$
$\text{sem}(s, \Delta) = \text{nested letrec-expressions with unused outer environment}$
- $t_1, t_2$ are meta-expressions restricted by constraints $\nabla$
- computing joins $\rightarrow^*$ requires abstract rewriting by rewrite rules $\ell \rightarrow_\Delta r$ with $\Delta$ restricting $\ell$ and $r$
- matching equations $\ell \sqsubseteq t$ together with constraint tuples $\nabla, \Delta$
- a matcher $\sigma$ may instantiate $\ell$ but not $t$, i.e. $\sigma(\ell) = t$
- $l$ contains instantiable meta-variables and $t$ contains fixed meta-variables, denoted by $MV_I(\cdot)$ and $MV_F(\cdot)$
A **letrec matching problem** is a tuple $P=(\Gamma, \Delta, \nabla)$ where
- $\Gamma$ is a set of matching equations $s \sqsubseteq t$ s.t. $MV_I(t) = \emptyset$
- $\Delta=(\Delta_1, \Delta_2, \Delta_3)$ is a constraint tuple (needed constraints);
- $\nabla=(\nabla_1, \nabla_2, \nabla_3)$ is a constraint tuple (given constraints), s.t. $MV_I(\nabla)=\emptyset$ and $\nabla$ is satisfiable.

Occurrence restrictions for **instantiable meta variables**:
- Each instantiable $S$-variable occurs at most twice in $\Gamma$
- Each $E$, $Ch$, $D$-variable occurs at most once in $\Gamma$
Matcher of $P = (\Gamma, \Delta, \nabla)$

A substitution $\sigma$ is a **matcher of** $P = (\Gamma, \Delta, \nabla)$ iff

- $\sigma$ instantiates the instantiable variables and does not introduce new instantiable or fixed variables
- for any ground substitution $\rho$ on $MV_F(P)$ that satisfies $\nabla$ and where $\rho(\sigma(s))$ and $\rho(t)$ for $s \subseteq t \in \Gamma$ fulfill the LVC:
  - $\rho(\sigma(s)) \sim_{let} \rho(t)$ for all $s \subseteq t \in \Gamma$
  - the $\Delta$-constraints hold
    - ($\exists \rho_0$ with $\text{Dom}(\rho_0) = MV_I(\rho(\sigma(\Delta)))$ s.t. $\rho_0(\rho(\sigma(\Delta)))$ is satisfied.)

$\sim_{let} = $ syntactic equality up to permuting bindings in environments
Theorem (NP-Hardness)

The decision problem whether a matcher for a letrec matching problem exists is **NP-hard**.

Proof by a reduction from **Monotone one-in-three-3-SAT**.

Sketch: For each clause $C_i = \{S_{i,1}, S_{i,2}, S_{i,3}\}$, add the matching equation

\[
\begin{align*}
\text{letrec } & Y_{i,1} = S_{i,1}; \ Y_{i,2} = S_{i,2}; \ Y_{i,3} = S_{i,3} \text{ in } c \\
\leq & \text{letrec } y_{i,1} = false; \ y_{i,2} = false; \ y_{i,3} = true \text{ in } c
\end{align*}
\]
Intermediate **data structure** of the algorithm: \((Sol, \Gamma, \Delta, \nabla)\) where
- \(Sol\) is a computed substitution
- \(\Gamma\) is a set of equations
- \((\Delta_1, \Delta_2, \Delta_3)\) are needed constraints
- \((\nabla_1, \nabla_2, \nabla_3)\) are given constraints

**Input:**
For \(P = (\Gamma, \Delta, \nabla)\), MatchLRS starts with \((Id, \Gamma, \Delta, \nabla)\)

**Output** (on each branch):
- **Fail** or final state \((Sol, \emptyset, \Delta, \nabla)\)
Matching Algorithm MatchLRS: Rules

Inference rules of the form

\[
\begin{array}{c|c|c}
\text{State} \\
\hline
\text{State}_1 & \ldots & \text{State}_n \\
\end{array}
\]

Rule application is non-deterministic:
- don’t care non-determinism between the rules
- don’t know non-determinism between \( \text{State}_1 \ | \ \ldots \ | \ \text{State}_n \)
Selection of Rules (1)

Solving an expression-variable:

\[
\begin{align*}
(Sol, \Gamma \cup \{ S \leq s \}, \Delta) \\
\rightarrow
(Sol \circ \{ S \rightarrow s \}, \Gamma[s/S], \Delta[s/S])
\end{align*}
\]

Decomposition of letrec:

\[
\Gamma \cup \{ \text{letrec env in } s \leq \text{letrec env'} in t \}
\]

\[
\rightarrow
\Gamma \cup \{ \text{env } \leq \text{env'}, s \leq t \}
\]

Prefix-rule for contexts: \( D' \) is a prefix of \( D \)

\[
\begin{align*}
(Sol, \Gamma \cup \{ D[s] \leq D'[s'] \}, \Delta, \nabla) & \quad \text{if } D \in \Delta_1 \iff D' \in \nabla_1 \\
(Sol \circ \sigma, \Gamma \cup \{ D''[s] \leq s' \}, \Delta\sigma, \nabla) & \quad \text{and } cl(D') \leq cl(D) \\
\text{s.t. } \sigma=\{ D \rightarrow D'[D''] \}, cl(D'')=cl(D)
\end{align*}
\]
Selection of Rules (2)

\[(Sol, \Gamma \cup \{env \leq b; env'\}, \Delta, \nabla)\]

\[\forall b': env=b'; env''\]

\[\Rightarrow (Sol \circ \sigma, \Gamma \cup \{E'; env'' \leq env'\}, \Delta\sigma, \nabla) \text{ where } \sigma = \{E \mapsto b; E'\}\]

\[\forall E: env=E; env''\]

\[\Rightarrow (Sol \circ \sigma, \Gamma \cup \{y.D[s] \leq b, env'' \leq env'\}, \Delta\sigma, \nabla)\]

\[\text{where } \sigma = \{Ch[\cdot_1, \cdot_2] \mapsto [\cdot_1].D[\cdot_2]\} \text{ and } cl(D) = cl(Ch)\]

\[\forall Ch: env=Ch[y,s]; env''\]

\[\Rightarrow (Sol \circ \sigma, \Gamma \cup \{y.D[X] \leq b, Ch_2[X,s]; env'' \leq env'\}, \Delta\sigma, \nabla)\]

\[\text{where } \sigma = \{Ch[\cdot_1, \cdot_2] \mapsto [\cdot_1].D[X]; Ch_2[X, \cdot_2]\}, cl(D) = cl(Ch_2) = cl(Ch)\]

\[\forall Ch: env=Ch[y,s]; env''\]

\[\Rightarrow (Sol \circ \sigma, \Gamma \cup \{Y = D_1[X] \leq b, Ch_1[y, D_2[Y]]; Ch_2[X,s]; env'' \leq env'\}, \Delta\sigma, \nabla)\]

\[\text{where } \sigma = \{Ch[\cdot_1, \cdot_2] \mapsto Ch_1[\cdot_1, D_2[Y]]; Y = D_1[X]; Ch_2[X, \cdot_2]\}, cl(D_i) = cl(Ch_i) = cl(Ch)\]

\[\forall Ch: env=Ch[y,s]; env''\]

\[\Rightarrow (Sol \circ \sigma, \Gamma \cup \{X_1 = D[s] \leq b, Ch_1[y, D'[X_1]]; env'' \leq env'\}, \Delta\sigma, \nabla) \text{ where}\]

\[\sigma = \{Ch[\cdot_1, \cdot_2] \mapsto Ch_1[\cdot_1, D'[X_1]]; X_1.D[\cdot_2]\}, cl(D) = cl(D') = cl(Ch_1) = cl(Ch)\]

\[\forall Ch: env=Ch[y,s]; env''\]

environment with at least one binding \(b\) on the rhs of the equation
Selection of Rules (2)

\[(\text{Sol}, \Gamma \cup \{\text{env} \leq b; \text{env}'\}, \Delta, \nabla)\]

- \(b\) equals a binding \(b'\) on the lhs
- \(b\) is part of an environment variable \(E\) on the lhs
- \(b\) is part of a chain variable \(Ch\) on the lhs

4 cases:
- Chain consists of the single binding \(b\)
- \(b\) is a prefix of the chain
- \(b\) is an infix of the chain
- \(b\) is a suffix of the chain

Environment with at least one binding \(b\) on the rhs of the equation
Failure Rules

Usual cases:
- $\Gamma$ not empty, but no matching rule applicable

Examples:
- $f \ s_1 \ldots \ s_n \trianglelefteq g \ t_1 \ldots \ t_m$, or
- $D[s] \trianglelefteq D'[t]$ and $cl(D) < cl(D')$.

Extraordinary cases:
- $(Sol, \emptyset, \Delta, \nabla)$ but for some $s$ in an input equation $s \trianglelefteq t$, $Sol(s)$ violates the LVC

- NCC-implication check fails:
  - check that given constraints $\nabla$ imply needed constraints $\Delta$
  - also infers constraints from the LVC for input expressions

Example: letrec $X_1 = S_1; X_2 = S_2$ in ... implies validity of the non-capture constraint ($\text{var } X_1, \lambda X_2.[\_]$)
Theorem

*MatchLRS* is **sound and complete**, i.e.

- **(soundness)** if *MatchLRS* delivers $S = (Sol, \emptyset, \Delta, \nabla)$ for input $P$, then $Sol$ is a matcher of $P$; and
- **(completeness)** if $P = (\Gamma, \Delta, \nabla)$ has a matcher $\sigma$, then there exists an output $(\sigma, \emptyset, \Delta_S, \nabla_S)$ of *MatchLRS* for input $P$.

Theorem

*MatchLRS* runs in **NP-time**.

The letrec matching problem is **NP-complete**.
Conclusion

- Sound and complete matching algorithm for LRSX
- Designed to represent program calculi with recursive bindings
- Letrec matching problem is NP-complete
- Automated computation of overlaps and joins for call-by-need core languages is possible

Implementation: LRSX Tool (http://goethe.link/LRSXTOOL)
Conclusion

- Sound and complete matching algorithm for LRSX
- Designed to represent program calculi with recursive bindings
- Letrec matching problem is NP-complete
- Automated computation of overlaps and joins for call-by-need core languages is possible
- Implementation: LRSX Tool (http://goethe.link/LRSXTOOL)

Further work:

- join more cases by meta alpha-renaming (PPDP 2017, to appear)
- automated correctness of translations between program calculi