Matching of Meta-Expressions with Recursive Bindings

David Sabel

Goethe-University, Frankfurt am Main, Germany
sabel@ki.informatik.uni-frankfurt.de

1 Motivation and Problem Description

We focus automated reasoning on program calculi with reduction semantics (see e.g. [8]), in
particular, lambda-calculi with call-by-need evaluation and letrec-expressions (consisting of a set
of recursive bindings and a body to reference the bindings) modelling the core of lazy functional
programming languages like Haskell (see [1, 6]). The meta-language LRSX [5] is designed for
this purpose. It uses higher-order function symbols (as they also occur in approaches using
higher-order abstract syntax [3]), has a letrec-construct letr, meta-variables for environments,
expressions, variables and contexts. Its syntax (see Fig. 1) is parametrized over context classes
K and (higher-order) function symbols F. A ground LRSX-expression (an LRS-expression) does
not contain any meta-variable. Contexts are expressions with a hole [:] : HEexpr.

The semantics of meta-variables is straightforward except for chain-variables Ch where
Ch[x, s] of class K stands for x.d[s] or chains x.d1([var x1]); x1.d2([var x2]); ... : xn.dn[s] with
fresh variables x1, and contexts d1, d2, from the class K. As a motivation for chain-variables,
consider the reduction rule letr x1=A1[x2],..., xn−1=An[xn],xn=(λy.s0) s1 in A'[x1] →
letr x1=A1[x2],..., xn−1=An[xn],xn=(letr y=s1 in s0) in A'[x1]. It performs β-reduction
with sharing at a needed position (where A, A1 are evaluation contexts) which is expressed by the
informal notion x1=A1[x2],..., xn−1=An[xn] for a chain of bindings of arbitrary length. In LRSX
the left hand side of the rule can be represented as letr E; Ch[X1,app λY.S] S1 in A[var X1]
where Ch is a chain-variable of class A. The example also shows that the eta-syntax requires a
notion of contexts and context classes. The rule letr E1 in letr E2 in s → letr E1, E2 in s
joins two nested letrec-environments. The rule requires that scoping is respected, i.e. let-bindings
of E2 must not capture variables in E1. That is why we use so-called constrained expressions:

Definition 1. In a constrained expression (s, Δ) s is an LRSX-expression and Δ = (Δ1, Δ2, Δ3)
is a constraint tuple s.t. Δ1 is a set of context variables, called non-empty context constraints;
Δ2 is a set of environment variables, called non-empty environment constraints; and Δ3 is
a finite set of pairs (t, d) where t is an LRSX-expression and d is an LRSX-context, called non-capture constraints (NCCs). A ground substitution\(^2\) ρ satisfies Δ iff ρ(D) \≠ [\] for all D ∈ Δ1;
ρ(E) \≠ 0 for all E ∈ Δ2; and Var(ρ(t)) \cap CV(ρ(d)) = \emptyset for all (t, d) ∈ Δ3. The concretizations
of (s, Δ) are γ(s, Δ) := {ρ(s) | ρ is a ground substitution, ρ(s) fulfills the LVC\(^3\), ρ satisfies Δ}.

In this paper we treat matching of constrained expressions against constrained expressions.
An application is to apply rewrite rules to constrained expressions which is necessary to

---

\(^1\)A context class K ∈ K is a set of contexts. To ease reading, we use K = {Triv, A, T, C}, where Triv consists of
the empty context only, in C-contexts the hole is allowed everywhere, T contexts have the hole not below a
higher-order binder x.s, and the path into the hole in A-contexts always uses strict positions of function symbols.
We use the ordering Triv < A < T < C, since C ⊇ T ⊇ A ⊇ Triv.

\(^2\)A substitution ρ is ground iff it maps all variables in dom(ρ) to LRS-expressions.

\(^3\)s satisfies the let variable convention (LVC) iff no binder occurs twice in the same letr-environment.
compute joins for critical pairs that occur in the diagram method (see [6] and also [7, 2]) – a syntactic method to prove the correctness of program transformations. Here critical pairs stem from overlapping left hand sides of calculus reduction steps with left or right hand sides of transformation rules. The overlaps are computed on constrained meta-expressions using the unification algorithm for LR SX-expressions from [5]. For computing joins, the transformations and reduction steps have to be applied to the constrained expressions where a requirement is that all steps must be applicable to each instance of the constrained expressions. This leads to the following matching problem with two kinds of meta-variables: Usually matching means to solve directed equations of the form $s \leq t$ where $s$ contains meta-variables and $t$ is a ground expression. However, in our equations $s$ is a meta-expression with instantiable meta-variables and $t$ contains meta-variables which are treated like “meta-constants” (called fixed meta-variables). To distinguish the meta-variables, we use blue font for instantiable meta-variables and red font and underlining for fixed meta-variables. With $MV_I(\cdot)$ and $MV_P(\cdot)$ we compute the sets.

**Definition 2.** A letrec matching problem (LMP) is a tuple $P = (\Gamma, \Delta, \nabla)$ where $\Gamma$ is a set of matching equations $s \leq t$ s.t. $MV_I(t) = 0$; $\Delta = (\Delta_1, \Delta_2, \Delta_3)$ is a constraint tuple (needed constraints); $\nabla = (\nabla_1, \nabla_2, \nabla_3)$ is a constraint tuple (given constraints), s.t. $MV_I(\nabla_i) = 0$ for $i = 1, 2, 3$. $\nabla$ is satisfiable, for all expressions in $\Gamma$, the LVC holds, and every instantiable variable of kind $S$ (E, Ch, D, resp.) occurs at most twice (once, resp.) in $\Gamma$. A matcher of $P$ is a substitution $\sigma$ where $\text{Dom}(\sigma) = MV_I(\Gamma)$, $MV_I(\sigma(s)) = 0$ and $MV_P(\sigma(s)) \subseteq MV_P(\Gamma)$ for all $s \leq t \in \Gamma$, s.t. for any ground substitution $\rho$ with $\text{Dom}(\rho) = MV_P(P)$ which satisfies $\nabla$, $\rho(\sigma(s)), \rho(t)$ fulfill the LVC for all $s \leq t \in \Gamma$, we have $\rho(\sigma(s)) \sim_{\text{let}} \rho(t)$ for all $s \leq t \in \Gamma$ and there exists a ground substitution $\rho_0$ with $\text{Dom}(\rho_0) = MV_I(\rho(\sigma(\Delta)))$ s.t. $\rho_0(\rho(\sigma(\Delta)))$ is satisfied.

**Example 3.** The LMP $\{s \leq t, \Delta, \nabla\}$ with $s = \text{letrec } E_1 \text{ in } S_1$, $t = \text{letrec } E_2 \text{ in } S_2$, $\Delta = (\emptyset, \{E_1\}, \{S_1, \text{letrec } E_1 \text{ in } [\ ]\})$, and $\nabla = (\emptyset, \{E_2\}, \emptyset)$ has no matcher: The substitution $\sigma = \{E_1 \mapsto E_2, S_1 \mapsto S_2\}$ is not a matcher, since for instance, for $\rho = \{E_2 \mapsto x.\text{var x}, S_2 \mapsto \text{var x}\}$ the NCC $\rho(\sigma((S_1, \text{letrec } E_1 \text{ in } [\ ]))) = (\text{var x.letrec x.\text{var x in [\ ]}})$ is violated. However, the substitution $\sigma$ is a matcher of the LMP $\{s \leq t, \Delta, \nabla'\}$ with $\nabla' = (\emptyset, \{E_2\}, \{S_2, \text{letrec } E_2 \text{ in } [\ ]\})$.

The unification algorithm in [5] cannot be reused for matching, since its occurrence restrictions
are too strong and it cannot infer whether the given constraints imply the needed constraints.

The additional substitution $\rho_0$ in the definition of a matcher is needed for the case that a transformation introduces “fresh” variables. E.g., the rewrite rule $\text{let}r X. c S_1 \in S_2 \rightarrow \text{let}r X. c (\text{var} Y); Y S_1 \in S_2$ requires NCCs $\{\text{var} X, \text{var} Y, [;]\}, \{(S_1, \text{var} Y, [;])\}$ to ensure that $Y$ is fresh. Matching the left hand side of the rule against $\text{let}r u. c (\text{var} y) \text{in} \text{var} u$, will not instantiate the variable $Y$. After instantiation, the NCCs become $\{\text{var} u, \text{var} y, [;]\}, \{(S_2, \text{var} Y, [;])\}$. Validity depends on the instantiation of $Y$. The definition of a matcher allows us to choose an instance that satisfies the constraints (e.g. $\rho_0 = \{Y \mapsto w\}$). Any instantiation which satisfies the NCCs is valid, and thus to use matching for symbolic reduction, we can also keep the constraints (instead of using a ground instance) and add them to the given constraints on the result.

2 Solving the Letrec Matching Problem

We present the algorithm MATCHLRS to solve LMPs\(^4\). A state is a tuple $(Sol, \Gamma, \Delta, \nabla)$ where $Sol$ is a computed substitution and $(\Gamma, \Delta, \nabla)$ is a LMP s.t. $\Gamma$ consists of expression-, environment-, binding-, and variable-equations. For $(\Gamma, \Delta, \nabla)$, the state is initialized as $(Id, \Gamma, \Delta, \nabla)$. The output is either a final state $(Sol, \emptyset, \Delta, \nabla)$ or Fail. In Figs. 2 and 3 are the rules of MATCHLRS where only necessary components of the states are shown. They are inference rules $\frac{\text{Sol}\{X \leq x, \Delta\}}{Sol\{X \rightarrow x, \Gamma[\leq \text{env}], \Delta[\leq \text{env}]\}}$ s.t. for state $S$, the algorithm (don’t know) non-deterministically branches into $S_1, \ldots, S_n$.

\(^4\)MATCHLRS is implemented as a part of the LRSX Tool (goethe.link/LRSXTOOL) – a tool to automatically prove the correctness of program transformations.
Application of rules is don’t care non-determinism. Rules (SolveX) and (SolveS) solve, and (EIX) and (EIS) eliminate an expression equation. Rules (DecF), (DecH), (Decl), and (Decd) decompose function symbols, higher-order binders, bindings, letrec-expressions, and contexts. Other rules on expressions treat equations of the form $D[s] \leq t$, where (CxPx) covers the case that $t$ is $D'[t']$ and $D'$ is a prefix of $D$ where $D$ must be at least as general as $D'$. If $D'$ is non-empty, but $D$ may be empty, then rule (CxGuess) is applicable. If the class of $D'$ is strictly more general than the class of $D$, $D$ must be instantiated by the empty context (rule (CxCG)). Rules (CxF) and (CxL) match the context variable against a function symbol or a let-rec expression. Rules (EnvEm) and (EIE) eliminate, and (SolveE) solves an environment equation. Rule (EnvAE) solves a set of environment variables by instantiating them with $\emptyset$, where env is non-empty if $\text{env} = \text{env}'$, $\text{env} = \text{Ch}[y,s]; \text{env}$, or $\text{env} = E'; \text{env}$ with $E' \in \nabla_2$. Rule (EnvB) is applicable if the right hand side of the equation contains a binding which may be matched against a binding, a part of a non-empty environment variable, or a part of a chain-variable, where four cases are possible: the binding coincides with, the binding is a prefix, a proper infix, or a suffix of the chain. Rule (EnvE) applies if the right hand side of an equation contains a fixed environment variable which has to be matched with a part of an instantiated variable. Rule (EnvC) covers the cases that a fixed chain-variable on the right hand side must be matched against the same variable on the left hand side, an instantiable environment variable, or an instantiation chain-variable.

For input $(\Gamma_1, \Delta_1, \nabla_1)$ and state $(\text{Sol}, \Gamma, \Delta, \nabla)$, MATCHLRS delivers Fail if $\Gamma \neq \emptyset$ and no rule is applicable, or if $\Gamma = \emptyset$ and i) for $s \leq t \in \Gamma_1$, Sol(s) violates the LVC, or ii) the NCC-implication check (Def. 4) is invalid. For this check, we split NCCs into atomic NCCs $(u,v)$ s.t. $u,v$ are variables or meta-variables as $\text{split}_{ncc}(S) := \bigcup \{ (s,d) \in \nabla \mid \text{Var}_{M}(s) \times CV_{M}(d) \}$ where $\text{Var}_{M}(s) \subseteq MV_{1}(s) \cup MV_{2}(s) \cup \text{Var}(s)$, and $CV_{M}$ collects all concrete variables that capture variables of the context hole, and all meta-variables which may have concretizations that introduce capture variables.

**Definition 4.** Let $(\text{Sol}, \emptyset, \Delta, \nabla)$ be a final state of MATCHLRS for input $(\Gamma_1, \Delta_1, \nabla_1)$. The NCC-implication check is valid iff for all $(u,v) \in \text{split}_{ncc}(\Delta_1)$ one of the following cases holds

1. $(u,v) \in \text{split}_{ncc}(\nabla_3) \cup \text{NCC}_{lvc}$ or $(u,v) = (x,y)$ where $x \neq y$.
2. $\emptyset \neq v$ and $u = \text{Ch}$ or $u = D$ or $u = E$ with $E \notin \Delta_2$.
3. $\emptyset \neq v$ and $u \in \{ \text{Ch}, \text{D}, \text{E}, \text{X} \}$ or $v \in \{ \text{Ch}, \text{D}, \text{E}, \text{X} \}$.
4. $u = \text{E}$ or $u = \text{Ch}$ with $\text{cl}(\text{Ch}) = \text{Triv}$ and $(u,v) \in \text{split}_{ncc}(\nabla_3) \cup \text{NCC}_{lvc}$.
5. $v \in \{ \text{E}, \text{Ch}, \text{D} \}$ and $(v,v) \in \text{split}_{ncc}(\nabla_3) \cup \text{NCC}_{lvc}$.
6. $(u,v) \in \text{split}_{ncc}(\nabla_3) \cup \text{NCC}_{lvc}$ and $(u,v) \in \{ (x,y),(x,y),(x,y),(x,y),(x,y),(x,y),(x,y),(x,y),(x,y),(x,y),(x,y) \} \in \text{split}_{ncc}(\nabla_3) \cup \text{NCC}_{lvc}$.

**Example 5.** We apply MATCHLRS for the LMP $\{(s \leq t), \Delta, \nabla\}$ with $s = \text{letrec} \text{Ch}[X,S_1]$ in $S_2$, $\Delta = (\Delta_1, \Delta_2, \Delta_3) = (\emptyset, \emptyset, \{ (S_1, |X|, \cdot) \})$, $t = \text{letrec} \text{Y.app} S_2 S_3 S_4$ in $S_5$, and $\nabla = (\nabla_1, \nabla_2, \nabla_3) = (\emptyset, \emptyset, \{ (S_3, |X|, \cdot) \})$ where $\text{cl}(\text{Ch}) = \text{A}$. Applying (DecL) and (SolS), yields $\{ S_2 \rightarrow S_3 \}$, $\text{Ch}[X,S_1] \subseteq \text{app} S_3 S_4$, $\Delta$, and (EnvB) branches into four states, where all but the first case result in Fail, since they imply that $\text{Ch}$ contains more than one binding. The remaining state is $\{ S_2 \rightarrow S_3, \text{Ch}[S_2, \cdot] \rightarrow [S_1, \cdot], \text{A}[S_1] \subseteq \text{app} S_3 S_4 \}$. Applying (DecH) and (SolX) results in $\{ S_2 \rightarrow S_3, \text{Ch}[S_2, \cdot] \rightarrow [S_1, \cdot], \text{A}[S_1] \subseteq \text{app} S_3 S_4, \Delta, \nabla \}$. Now (CxGuess) is applied and branches into two cases.

---

$NCC_{lvc} := \bigcup \{ NCC_{lvc}(r) \mid r \in \{ \text{Sol}(s,t) \} \}$ represents atomic NCCs implied by the LVC where $NCC_{lvc}(t) = \{ (x,y) \mid x \in \nabla \cap \text{Ch}[x,s]; \text{env} \in \nabla \cap \text{Ch}[x,s]; \text{env} \in \nabla \cap \text{Ch}[y,s]; \text{env} \in \nabla \cap \text{Ch}[x,y]; \text{env} \in \nabla \cap \text{Ch}[x,s]; \text{env} \in \nabla \cap \text{Ch}[y,s]; \text{env} \in \nabla \cap \text{Ch}[y,s]; \text{env} \in \nabla \cap \text{Ch}[x,y]; \text{env} \in \nabla \cap \text{Ch}[x,s]; \text{env} \in \nabla \cap \text{Ch}[y,s]; \text{env} \in \nabla \cap \text{Ch}[y,s]; \text{env} \in \nabla \cap \text{Ch}[x,y] \}$. $E$ is the set of all let-rec environments in $t$. 

4
Figure 3: Rules of MATCHLRS

**Matching of Meta-Expressions with Recursive Bindings**

Sabel

\[(\text{EnvAE})\]

\[
\frac{(\text{Sol}, \Gamma \cup \{E_1, \ldots, E_n \leq \emptyset\}, \Delta)}{(\text{Sol} \circ \sigma, \Gamma, \Delta, \nabla)} \quad \text{s.t. } \sigma = (E_i \mapsto \emptyset)_{i=1}^n
\]

\[
\Gamma \cup \{E_1 : \emptyset \leq E_2\} \quad \text{if } E_1 \not\in \text{Dom}(E_2)
\]

\[
\Gamma \cup \{\emptyset \leq \emptyset\}
\]

\[(\text{EnvE})\]

\[
\frac{(\text{Sol} \circ \sigma, \Gamma \cup \{E' : \emptyset \leq E'_1\}, \Delta, \nabla)}{(\text{Sol} \circ \sigma, \Gamma \cup \{E' : \emptyset \leq E'_1\}, \Delta, \nabla)} \quad \text{if } E \not\in E'_1, \exists E: E \leq E'_1
\]

\[
\forall E': E = E', E'_1 \text{ and } \emptyset \nabla = \emptyset \nabla
\]

\[(\text{SolveE})\]

\[
\frac{(\text{Sol} \circ \sigma, \Gamma) \cup \{E \leq \emptyset\}, \Delta, \nabla)}{(\text{Sol} \circ \sigma, \Gamma) \cup \{E \leq \emptyset\}, \Delta, \nabla)}\]

**if } E \in \Delta_2 \iff E \in \Delta_2^\prime**

\[(\text{EnvB})\]

\[
\frac{(\text{Sol} \circ \sigma, \Gamma \cup \{y' \leq b, \emptyset \leq \emptyset\}, \Delta, \nabla)}{(\text{Sol} \circ \sigma, \Gamma \cup \{y' \leq b, \emptyset \leq \emptyset\}, \Delta, \nabla)}\]

\[
\forall v: v' = b, \emptyset \leq \emptyset
\]

\[
\text{where } \sigma = (E \mapsto b, E')
\]

\[
\text{if } E \leq E'_1
\]

\[(\text{EnvC})\]

\[
\frac{(\text{Sol} \circ \sigma, \Gamma \cup \{y' \leq y, s' \leq s, \emptyset \leq \emptyset\}, \Delta, \nabla)}{(\text{Sol} \circ \sigma, \Gamma \cup \{y' \leq y, s' \leq s, \emptyset \leq \emptyset\}, \Delta, \nabla)}\]

\[
\forall \text{Ch}: \text{Ch} \leq \text{Ch}[y,s], \Delta, \nabla
\]

\[
\text{where } \sigma = (E \mapsto E', \text{Ch}[y,s])
\]

**Figure 3: Rules of MATCHLRS for environment equations. In rule (EnvC) the function split\_L is defined as follows:**

\[
\text{split\_L}(t) = \{([\cdot], t)\}
\]

\[
\text{split\_L}(A) = \{([\cdot], (f s_1 \ldots s_n)) \cap \{([\cdot], (f s_1 \ldots s_{n-1}) ([s_1, \ldots, s_n] t')) \in \text{split\_L}(A) \mid t' \in \text{sp}(f)\}
\]

\[
\text{split\_L}(A[s]) = \{([\cdot], A[s]) \cap \{([\cdot], A[s]) \mid t' \in \text{split\_L}(A[s])\}
\]

\[
\text{split\_L}(t) = \{([\cdot], t)\}, \text{if } t \neq (f s_1 \ldots s_n) \text{ and } t \neq A[s].
\]
If $A$ is guessed as empty, then the next state is $\{S_2 \mapsto S_3, \text{Ch}[1,2] \mapsto [1], A[2], X \mapsto Y, S_1 \leq \text{app } \text{Sol}_S \text{app } \text{Sol}(A) \text{app } \text{Sol}(X), \text{app } \Delta^{'}, \text{app } \Delta \}$, Applying (SolS) yields $\{S_2 \mapsto S_3, \text{Ch}[1,2] \mapsto [1], A[2], X \mapsto Y, S_1 \mapsto \text{app } \text{Sol}_S \text{app } \text{Sol}(X), \text{app } \Delta^{'}, \text{app } \Delta \}$ where $\Delta^{'}, \Delta$ are leads to Fail, and none of the cases of Def. 4 holds. Applying (CxGuess) yields two cases: if $A'$ is guessed to be non-empty, rule (CxFaill) is leads to Fail, and if $A'$ is guessed to be empty, the next state is $\{S_2 \mapsto S_3, \text{Ch}[1,2] \mapsto [1], A[2], X \mapsto Y, A \mapsto \text{app } \{S_2 \mapsto S_3, \text{Ch}[1,2] \mapsto [1], A[2], X \mapsto Y, A \mapsto \text{app } \text{Sol}_S \text{app } \text{Sol}(X), \text{app } \Delta^{'}, \Delta \}$ and (SolS) results in $\{S_2 \mapsto S_3, \text{Ch}[1,2] \mapsto [1], A[2], X \mapsto Y, A \mapsto \text{app } \text{Sol}_S \text{app } \text{Sol}(X), \text{app } \Delta^{'}, \Delta \}$ the NCC-implication check is valid since $\text{split}_{ncc}(\Delta_3) = \{(S_4, Y), (S_4, Y)\}$ and $(S_4, Y) \in \text{split}_{ncc}(\Delta_3)$. Thus the algorithm delivers the matcher $\{S_2 \mapsto S_3, \text{Ch}[1,2] \mapsto [1], A[2], X \mapsto Y, A \mapsto \text{app } \text{Sol}_S \text{app } \text{Sol}(X), S_1 \leq S_2\}$.

In the technical report [4] the following properties for MatchLRS are implemented in the LRSX Tool, and are part of an automated method to prove correctness of program transformations for program calculi expressible in LRSX.

**Theorem 6.** MATCHLRS is sound and complete, i.e. i) (soundness) if MATCHLRS delivers $S = (\text{Sol}, \emptyset, \Delta, \text{app } \lambda \text{app } \Delta, \text{app } \lambda \text{app } \Delta)$ for input $P$, then Sol is a matcher of $P$; and ii) (completeness) if a LMP $P = (\Gamma, \Delta, \text{app } \lambda \text{app } \Delta, \text{app } \lambda \text{app } \Delta)$ has a matcher $\sigma$, then there exists an output $(\sigma, \emptyset, \Delta_S, \text{app } \lambda \text{app } \Delta_S)$ of MATCHLRS for input $P$. MATCHLRS runs in NP-time, and the letrec matching problem is NP-complete.

3 Conclusion

Motivated by symbolically rewriting meta-expressions of the language LRSX, we formulated the letrec matching problem and developed the algorithm MATCHLRS. We obtained soundness and completeness for MATCHLRS and NP-completeness of the letrec matching problem. The presented algorithms are implemented in the LRSX Tool, and are part of an automated method to prove correctness of program transformations for program calculi expressible in LRSX.

Acknowledgments

We thank the reviewers of UNIF 2017 for their valuable comments.

References


