Institut für Informatik
Fachbereich Informatik und Mathematik

Rewriting of Higher-Order-Meta-Expressions with Recursive Bindings

David Sabel

2017

Frankfurter Informatik-Berichte
Institut für Informatik • Robert-Mayer-Str. 11-15 • D-60325 Frankfurt am Main, Germany

ISSN 1868-8330

Rewriting of Higher-Order-Meta-Expressions with Recursive Bindings

David Sabel*

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

Abstract. We introduce rewriting of meta-expressions which stem from a meta-language that uses higher-order abstract syntax augmented by meta-notation for recursive let, contexts, sets of bindings, and chain variables. Additionally, three kinds of constraints can be added to meta-expressions to express usual constraints on evaluation rules and program transformations. Rewriting of meta-expressions is required for automated reasoning on programs and their properties. A concrete application is a procedure to automatically prove correctness of program transformations in higher-order program calculi which may permit recursive let-bindings as they occur in functional programming languages. Rewriting on meta-expressions can be performed by solving the so-called letrec matching problem which we introduce. We provide a matching algorithm to solve it. We show that the letrec matching problem is NP-complete, that our matching algorithm is sound and complete, and that it runs in non-deterministic polynomial time.

1 Introduction

We are interested in meta-languages which are capable to represent the syntax and semantics of program calculi in form of a reduction semantics with evaluation contexts (see e.g. [21]). We are particularly interested in extended lambda-calculi with call-by-need evaluation which model the (untyped) core languages of lazy functional programming languages like Haskell (see [2,1,18]). A common construct are cyclic let-expressions representing an unordered set of recursive bindings and a body which can reference the bindings. With those *letrec*-expressions recursive functions and sharing can easily be expressed.

To represent those program calculi, we introduced the meta-language `LRSX` in [16]. It uses higher-order abstract syntax [11] extended with a letrec-construct `letr` and further meta-constructs which stem from modeling small-step reduction rules. For example, the following reduction rule

$$\begin{array}{l}\texttt{letr}\ x_1{=}A_1[x_2],\ldots,x_{n-1}{=}A_n[x_n],x_n{=}(\lambda y.s_0)\ s_1\ \texttt{in}\ A'[x_1]\\ \to \texttt{letr}\ x_1{=}A_1[x_2],\ldots,x_{n-1}{=}A_n[x_n],x_n{=}(\texttt{letr}\ y{=}s_1\ \texttt{in}\ s_0)\ \texttt{in}\ A'[x_1]\end{array}$$

performs a (sharing-variant) of β-reduction at a needed position (assuming that A,A_i represent evaluation contexts). However, the search for the reduction position is modeled by the informal notion $x_1{=}A_1[x_2],\ldots,x_{n-1}{=}A_n[x_n]$ for a chain of bindings (of arbitrary length). Our meta-language provides so-called chain-variables to represent the chains on the meta level. The example also shows that the meta-syntax requires a notion of contexts for different context classes. The rule $\texttt{letr}\ \mathit{Env}_1\ \texttt{in}\ \texttt{letr}\ \mathit{Env}_2\ \texttt{in}\ s \to \texttt{letr}\ \mathit{Env}_1, \mathit{Env}_2\ \texttt{in}\ s$ where $\mathit{Env}_1, \mathit{Env}_2$ represent arbitrary letrec-environments joins two nested letrec-environments. Our meta-language supports this representation by providing meta-variables for (parts of) `letr`-environments. The rule also requires that scoping is respected, i.e. let-bindings of Env_2 must not capture variables in Env_1. That is why we use so-called *constrained expressions*, which are meta-expressions augmented by constraints which restrict the ground instances of the expression. Hence our meta-language is capable to model higher-order program calculi with recursive bindings, e.g., the calculus L_{need} [17] which is a call-by-need lambda calculus with `letr`, as well as the calculus LR [18] which extends L_{need} by constructs of core languages for Haskell.

We focus on automated proofs of the correctness of program transformations for program calculi which are representable using `LRSX`. In this paper we are concerned with applying rewrite rules to constrained expressions. One application of this rewriting is the diagram method (see [18,14] and also [20,9,8]) which is a syntactic method to prove the correctness of program transformations. One step of the method is to compute joins for overlaps which are pairs (s,t) of constrained expressions that stem from overlapping standard reductions of the calculus with program transformation steps. To compute joins, rewriting of s and t w.r.t. a set of rules consisting of standard reductions of the calculus and also program transformations has to be performed to find a common successor of s and t.

*This research is supported by the Deutsche Forschungsgemeinschaft (DFG) under grant SA2908/3-1.

$$
\begin{aligned}
x, y, z \in \mathbf{Var} &::= X \mid \mathsf{x} \\
s, t \in \mathbf{HExpr}^0 &::= S \mid D[s] \mid \mathtt{letr}\ env\ \mathtt{in}\ s \mid (f\, r_1 \ldots r_{ar(f)}) \\
&\quad \text{where } r_i \in \mathbf{HExpr}^k \text{ if } oar(f)(i) = k \geq 0, \text{ and } r_i \in \mathbf{Var}, \text{ if } oar(f)(i) = \mathbf{Var}. \\
s \in \mathbf{HExpr}^n &::= x.s_1 \quad \text{if } s_1 \in \mathbf{HExpr}^{n-1} \text{ and } n \geq 1 \\
b \in \mathbf{Bind} &::= x.s \quad \text{where } s \in \mathbf{HExpr}^0 \\
env \in \mathbf{Env} &::= \emptyset \mid E; env \mid Ch[x, s]; env \mid b; env
\end{aligned}
$$

Fig. 1. Syntax of LRSX, the constructs X, S, D, E, Ch are meta-variables.

Results. We focus on solving the so-called letrec matching problem. Instances of the problem can be used to perform rewriting of constrained expressions. We sketch the matching algorithm MatchLRS which and argue that it is sound and complete w.r.t. the letrec matching problem (Theorem 4.13), and that it runs in non-deterministic polynomial time. Furthermore, we show that the letrec matching problem is NP-complete (Theorem 4.15). Also an implementation of the matching algorithm exists, it is used in the LRSX Tool[1] – a tool to automatically prove correctness of program transformations.

Related and Previous Work. Higher-order abstract syntax was introduced in [11] for implementing higher-order unification and matching. Since our approach should be applicable to descriptions of program calculi, we have to combine several techniques and thus require a matching algorithm which can treat meta-variables representing environments, chains, contexts, and expression variables. An approach for syntactic reasoning on higher-order expressions and binders are nominal techniques [12] which reason w.r.t. α-equivalence, including nominal unification [19,4,7], nominal matching [3], and nominal rewriting [6]. However, our focused functional languages contain letrec (see e.g. [5] for a discussion on reasoning with more general name binders) and require more sophisticated constructs which are not available for nominal reasoning. A recent approach is [15] where a nominal unification algorithm including recursive bindings is given. However, it cannot deal with environment, context and chain variables. Thus we use a syntactic approach excluding alpha-equivalence.

In [16] a unification algorithm for LRSX-expressions was developed which also removed several restrictions on the input problem which were present in an earlier attempt [13] to unify expressions with letrec. Usually unification is more complicated than matching (since matching is a unification problem where variables occur only on one side of the equations), but for our meta-expressions the situation is different, since i) the occurrence restrictions on meta-variables occurring in the unification problems in [16] are too strong for symbolic rewriting and our application (computing joins as part of the diagram method), ii) since we want to rewrite meta-expressions, meta-variables occur on both sides of matching equations, however with a different meaning: on one side the variables can be instantiated by the matcher, while on the other side they represent sets of expressions, environments, contexts, variables or chains which are fixed, iii) the matching problem is defined on constrained expressions and a matcher has to ensure that the given constraints imply the needed constraints, while the unification algorithm in [16] is not designed to handle this.

Outline. In Sect. 2 we recall our meta-language LRSX. In Sect. 3 we introduce constrained expressions, the notion of meta letrec rewrite rules, and the letrec matching problem. Sect. 4 contains the matching algorithm MatchLRS, and we prove soundness and completeness of MatchLRS, and show NP-completeness of the letrec matching problem. We conclude in Sect. 5.

2 The Meta-Language

We recall the syntax of the meta-language LRSX (see also [16]) which covers several extended lambda calculi (e.g. [18,10,2]). It is parametrized over a set $\mathcal{F}$ of function symbols and a finite set $\overline{K}$ of context classes. However, to avoid complex definitions, in this paper we work with four context classes only, and thus assume $\overline{K} = \{\mathit{Triv}, \mathcal{A}, \mathcal{T}, \mathcal{C}\}$ where *Triv* only contains the empty context, $\mathcal{A}$ are applicative contexts, $\mathcal{T}$ are top-contexts, and $\mathcal{C}$ are arbitrary contexts.

The *syntax of the language* LRSX$(\overline{K}, \mathcal{F})$ is defined in Fig. 1 with four syntactic categories of objects (called types): **Var** is a countably-infinite set of variables, **HExpr** are higher-order expressions, **Env** are letrec-environments, and **Bind** are letrec-bindings. Elements o of **HExpr** have an *order* $order(o) \in \mathbb{N}_0$, where $\mathbf{HExpr}^n$ denotes the elements of **HExpr** of order n. We set $\mathbf{Expr} = \mathbf{HExpr}^0$. Every $f \in \mathcal{F}$ has a syntactic type of the form $f : \tau_1 \to \ldots \to \tau_n \to \mathbf{Expr}$, where τ_i may be **Var**, or $\mathbf{HExpr}^{k_i}$; n is called the *arity* of f, denoted $ar(f)$; and the *order arity* $oar(f)$ is the n-tuple $\langle \delta_1, \ldots, \delta_n \rangle$, where

[1] http://goethe.link/LRSXTOOL

$\delta_i = k_i \in \mathbb{N}_0$, or $\delta_i = \mathbf{Var}$, depending on the type of f. We write $oar(f)(i)$ to extract δ_i. For $f \in F$, $sp(f) \subseteq \{i \mid 1 \leq i \leq ar(f), oar(f)(i) = 0\}$ denotes the *set of strict positions of* f. In $\mathcal{F}$ there is at least a unary operator `var` of type $\mathbf{Var} \to \mathbf{Expr}$ which lifts variables to expressions, where $ar(\mathtt{var}) = 1$, $oar(\mathtt{var}) = \langle \mathbf{Var} \rangle$, and $sp(\mathtt{var}) = \emptyset$, and the operator λ with $ar(\lambda)$=1, $oar(\lambda)$=$\langle 1 \rangle$, and $sp(\lambda)$=$\emptyset$.

To distinguish concrete term variables, meta-variables, and meta-symbols, we use different fonts and lower- or upper-case letters: concrete term-variables of type **Var** are denoted by x, y, and x, y are used as meta-symbols to denote a concrete term variable or a meta-variable. Similarly, lower-case letters s, t denote expressions, *env* denotes environments, and b denotes bindings. Meta-variables are written in upper-case letters, where X, Y are of type **Var**, S is of type **Expr**, E is of type **Env**, D is a context variable, and *Ch* is a two-hole environment-context variable (chain variable, for short) which must be of the type $\mathbf{Var} \to \mathbf{Expr} \to \mathbf{Env}$, and occurs with a **Var**-argument x, and an **Expr**-argument s. Each context variable has a class $cl(D) \in \{\mathcal{A}, \mathcal{T}, \mathcal{C}\}$ and each *Ch*-variable has a class $cl(Ch) \in \{Triv, \mathcal{A}\}$. An LRSX-expression s is *ground* (an LRS-*expression*, often written as s) iff it does not contain any meta-variable.

Contexts are expressions, where the symbol $[\cdot] : \mathbf{Expr}$ (the hole) is permitted to occur instead of one subexpression. With D we denote meta-variables for contexts, d represents a concrete context (i.e. an LRS-expression with a hole), and d denotes LRSX-contexts, i.e. contexts, that may contain meta-variables. Filling the hole of d with s is written as $d[s]$. Multi-contexts with $k > 1$ holes are written with several hole symbols $[\cdot_1], \ldots, [\cdot_k]$. A *context class* $\mathcal{K} \in \overline{K}$ is a set of contexts. Classes $\mathcal{A}$, $\mathcal{T}$, $\mathcal{C}$ are defined by the following grammar where D_A, D_T, D_C are context-variables s.t. $cl(D_C) \in \{\mathcal{A}, \mathcal{T}, \mathcal{C}\}$, $cl(D_T) \in \{\mathcal{A}, \mathcal{T}\}$, and $cl(D_A) = \mathcal{A}$; and where $f, g \in \mathcal{F}$ s.t. $oar(f)(i) = 0$ and $oar(g)(i) = m$:

$$d_A \in \mathcal{A} ::= D_A \mid [\cdot] \mid f\, s_1 \ldots s_{i-1}\, d_A\, s_{i+1} \ldots s_n \text{ where } i \in sp(f)$$
$$d_T \in \mathcal{T} ::= D_T \mid [\cdot] \mid \mathtt{letr}\, x.d_T; env\, \mathtt{in}\, s \mid \mathtt{letr}\, env\, \mathtt{in}\, d_T \mid f\, s_1 \ldots s_{i-1}\, d_T\, s_{i+1} \ldots s_n \text{ if } oar(f)(i) = 0$$
$$d_C \in \mathcal{C} ::= D_C \mid [\cdot] \mid \mathtt{letr}\, x.d_C; env\, \mathtt{in}\, s \mid \mathtt{letr}\, env\, \mathtt{in}\, d_C \mid g\, s_1 \ldots s_{i-1}\, x_1. \ldots x_m.d_C\, s_{i+1} \ldots s_n$$

The class *Triv* contains only the empty context $[\cdot]$ and there are no context variables for this class. We use the ordering $Triv < \mathcal{A} < \mathcal{T} < \mathcal{C}$, since $\mathcal{C} \supseteq \mathcal{T} \supseteq \mathcal{A} \supseteq Triv$.

Example 2.1. Since $oar(\lambda)$=$\langle 1 \rangle$, λ must be applied to a higher-order expression of order 1. The identity function is represented by applying λ to x.(`var` x) written as $\lambda\mathsf{x}$.(`var` x). Applications can be represented by a function symbol `app` with $ar(\mathtt{app})$=2, $oar(\mathtt{app})$=$\langle 0, 0 \rangle$, and $sp(\mathtt{app})$= $\{1\}$. The context $\lambda x.[\cdot]$ is a $\mathcal{C}$-context but neither a top- nor an application context, the context (`app` S $[\cdot]$) is a $\mathcal{C}$- and $\mathcal{T}$-context, but not an application context (since $2 \notin sp(\mathtt{app})$), while the context (`app` (`app` $[\cdot]$ S_1) S_2 is an $\mathcal{A}$-context.

Example 2.2. If $oar(f)(i)$=**Var**, then the i^{th} argument of function symbol f must be a variable. For instance, with the definition $\mathtt{appx} \in \mathcal{F}$, $oar(\mathtt{appx}) = \langle 0, \mathbf{Var} \rangle$ we introduce an application where the second argument is restricted to variables (e.g. in [10] such applications occur). Constructors can be represented by a function symbol f where $sp(f)$=$\emptyset$ and $oar(f)$ is a tuple of only 0-s. For example, the list constructors are `nil` and `cons` with $ar(\mathtt{nil})$=0, $oar(\mathtt{nil})$=$\langle\rangle$, $ar(\mathtt{cons})$=2, $oar(\mathtt{cons})$=$\langle 0, 0 \rangle$, and $sp(\mathtt{cons})$=$sp(\mathtt{nil})$=$\emptyset$.

Definition 2.3. *For any syntactic object r, let $MV(r)$ be the* set of meta-variables occurring in r. *In a higher-order expression $x.r$, the* scope *of x is r. The* scope *of x in* `letr` $x.s; env$ `in` s' *or* `letr` $Ch[x, s]; env$ `in` s' *is s, env, Ch and s'. With $FV(r)$ we denote the set of variables x that are not bound by some higher-order binder, a let-binding, or the x in $Ch[x, s]$, and with $BV(r)$ we denote the set of bound variables. We write $Var(r)$ for $FV(r) \cup BV(r)$. For environment env, $LV(env)$ denotes the let-bound variables in env, i.e. all x s.t. $env = env'; x.s$ or $env = env'; Ch[x, s]$. For a ground context* d, $CV(\mathsf{d})$ *(the* captured variables*) denotes the set of variables* x *which become bound if plugged into the hole of* d. *Every context class except for Triv must contain a non-empty context* d, *s.t.* $CV(\mathsf{d}) = \emptyset$, *and for every variable* y *it contains a context* d_y *s.t.* $Var(\mathsf{d}_\mathsf{y}) = \{\mathsf{y}\}$[2]. *Let $\sim_{let}$ be the reflexive-transitive closure of permuting bindings in a* `letr`*-environment, and $\sim_\alpha$ (*extended α-equivalence*) be the reflexive-transitive closure of combining $\sim_{let}$ and α-equivalence.*

Definition 2.4. *Meta-variables represent ground expressions, environments, and contexts. The semantics of meta-variables X, Y are all concrete variables of type* **Var**, *expression variables S represent any ground expression of type* **Expr**, *and environment variables E represent all ground environments of type* **Env**. *The semantics of a context variable D with $cl(D) = \mathcal{K}$ are all contexts of context class $\mathcal{K}$. The construct*

[2] Note that these assumptions can be satisfied, if $\mathtt{app} \in \mathcal{F}$, since $Var(\mathtt{app}\ [\cdot]\ (\mathtt{var}\ \mathsf{y})) = \{\mathsf{y}\}$ and $CV(\mathtt{app}\ [\cdot]\ (\mathtt{var}\ \mathsf{x})) = \emptyset$.

$Ch[x,s]$ with $cl(Ch) = \mathcal{K}$ stands for $x.\mathtt{d}[s]$ or chains $x.\mathtt{d}_1[(\mathtt{var}\ \mathtt{x}_1)]; \mathtt{x}_1.\mathtt{d}_2[(\mathtt{var}\ \mathtt{x}_2)]; \ldots ; \mathtt{x}_n.\mathtt{d}_n[s]$ with fresh variables $\mathtt{x}_i$, and contexts $\mathtt{d}, \mathtt{d}_i$ from the context class $\mathcal{K}$.

A substitution *ρ maps a finite set of meta-variables to variables, expressions, environments, and contexts respecting their types and classes. With $\mathtt{Dom}(\rho)$ ($\mathtt{Cod}(\rho)$, resp.) we denote the domain (co-domain, resp.) of ρ. Substitutions for chain-variables Ch map two-hole environment-contexts to two-hole environment contexts and they must be of the form $\{Ch[\cdot_1, \cdot_2] \mapsto [\cdot_1].d_1[(\mathtt{var}\ x_1)]; x_1.d_2[(\mathtt{var}\ x_2)]; \ldots ; x_n.d_n[\cdot_2]\}$ where d_i are (meta) contexts of class $cl(Ch)$. A substitution ρ is* ground *iff it maps all variables in $\mathtt{Dom}(\rho)$ to* LRS*-expressions.*

Example 2.5. The rule

$$\begin{array}{l}\mathtt{letr}\ x_1{=}A_1[x_2], \ldots, x_{n-1}{=}A_n[x_n], x_n{=}(\lambda y.s_0)\ s_1\ \mathtt{in}\ A'[x_1] \\ \to \mathtt{letr}\ x_1{=}A_1[x_2], \ldots, x_{n-1}{=}A_n[x_n], x_n{=}(\mathtt{letr}\ y{=}s_1\ \mathtt{in}\ s_0)\ \mathtt{in}\ A'[x_1]\end{array}$$

can be written in LRSX as

$$\mathtt{letr}\, E; Ch[X_1, \mathtt{app}\,(\lambda Y.S_0)\, S_1]\, \mathtt{in}\, A[\mathtt{var}\, X_1] \to \mathtt{letr}\, E; Ch[X_1, \mathtt{letr}\, Y.S_1\, \mathtt{in}\, S_0]\, \mathtt{in}\, A[\mathtt{var}\, X_1]$$

where Ch is a chain-variable of class $\mathcal{A}$.

Definition 2.6. *An* LRSX*-expression s satisfies the* let variable convention (LVC) *iff a let-bound variable does not occur twice as a binder in the same* letr*-environment; and s satisfies the* distinct variable convention (DVC) *iff $BV(s)$ and $FV(s)$ are disjoint and all binders bind different variables.*

We use this definition for concrete and for X-variables. E.g., $s{=}\mathtt{letr}\, X.\mathtt{var}\, X; Y.\mathtt{var}\, Y\ \mathtt{in}\, S$ fulfills the LVC, while for $\rho{=}\{X{\mapsto}\mathtt{x}, Y{\mapsto}\mathtt{x}, S{\mapsto}\mathtt{var}\,\mathtt{x}\}$, the LVC is violated for $\rho(s)$, since there are two let-binders for $\mathtt{x}$ in the same environment.

3 Constrained Meta-Expressions and Letrec Rewrite Rules

Definition 3.1. *A* constrained meta-expression *(s, Δ) consists of an* LRSX*-expression s and a* constraint tuple *$\Delta = (\Delta_1, \Delta_2, \Delta_3)$ s.t. Δ_1 is a finite set of context variables, called* non-empty context constraints*; Δ_2 is a finite set of environment variables, called* non-empty environment constraints*; and Δ_3 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, for short). A ground substitution ρ* satisfies *Δ iff for $i = 1, 2, 3$, ρ satisfies Δ_i, where ρ satisfies Δ_1 iff $\rho(D) \neq [\cdot]$ for all $D \in \Delta_1$; ρ satisfies Δ_2 iff $\rho(E) \neq \emptyset$ for all $E \in \Delta_2$; and ρ satisfies Δ_3 iff $Var(\rho(t)) \cap CV(\rho(d)) = \emptyset$ for all $(t, d) \in \Delta_3$. If there exists a ρ that satisfies Δ, then Δ* is satisfiable*. The* concretizations *of (s, Δ) are $\gamma(s, \Delta) := \{\rho(s) \mid \rho$ is a ground substitution, $\rho(s)$ fulfills the LVC, ρ satisfies $\Delta\}$.*

Example 3.2. For $\Delta = (\Delta_1, \Delta_2, \Delta_3) = (\emptyset, \{E_1, E_2\}, \{(\mathtt{letr}\ E_1\ \mathtt{in}\ c, \mathtt{letr}\ E_2\ \mathtt{in}\ [\cdot])\})$, the constrained expression $(\mathtt{letr}\ E_1\ \mathtt{in}\ \mathtt{letr}\ E_2\ \mathtt{in}\ S, \Delta)$ represents all LRS-expressions that are nested letr-expressions s.t. both letr-environments are non-empty and the let-variables of the inner environment are distinct from all variables occurring in the outer environment.

An example that requires non-empty context constraints is the following reduction rule from the calculus L_{need} [17] which copies an abstraction into a needed position in a letr-environment by following indirections:

$$\begin{array}{l}\mathtt{letr}\, E; Ch_1[X_n, \lambda X.S]; Ch[Y, A_1[\mathtt{var}\, X_n]]\, \mathtt{in}\, A[\mathtt{var}\, Y] \\ \to \mathtt{letr}\, E; Ch_1[X_n, \lambda X.S]; Ch[Y, A_1[\lambda X.S]]\, \mathtt{in}\, A[\mathtt{var}\, Y]\end{array}$$

where $cl(Ch){=}\mathcal{A}$, $cl(Ch_1){=}Triv$. If A_1 is empty, then the target of the copy operation should be the variable Y in $A[\mathtt{var}\, Y]$. Thus the case $A_1 = [\cdot]$ should be excluded which can be expressed by setting $\Delta_1 = \{A_1\}$.

Example 3.3. Reconsider the reduction rule in Example 2.5: The rule must not be applied to instances where the variable y occurs in expression s_1, since otherwise, the variable y is captured in the generated letr-expression $(\mathtt{letr}\ y = s_1\ \mathtt{in}\ s_0)$. To forbid such captures (and instances) we can add the NCC $(S_1, \lambda Y.[\cdot])$ to the rule.

$$\begin{array}{llll} CV_M(D[d]) = CV_M(D)\cup CV_M(d) & CV_M(\texttt{letr}\, env\, \texttt{in}\, d) & = CV_M(env)\cup CV_M(d) & CV_M(x) = \emptyset \\ CV_M(x.d) = \{x\}\cup CV_M(d) & CV_M(\texttt{letr}\, z.d; env\, \texttt{in}\, s) & = CV_M(env)\cup\{z\}\cup CV_M(d) & CV_M(S) = \emptyset \\ CV_M(D) = \emptyset, \text{ if } cl(D)=\mathcal{A} & CV_M(\texttt{letr}\, Ch[z,d]; env\, \texttt{in}\, s) & = CV_M(env)\cup\{Ch,z\}\cup CV_M(d) & CV_M([\cdot]) = \emptyset \\ CV_M(D) = \{D\}, \text{ if } cl(D)\neq\mathcal{A} & CV_M(f\, s_1 \ldots d \ldots s_n) & = CV_M(d) & \end{array}$$

$$CV_M(env) = \bigcup\{\{Ch,z\} \mid Ch[z,s];env'=env\}\cup\{E \mid E;env'=env\}\cup\{z \mid z.s;env'=env\}$$

Fig. 2. The function CV_M

When computing with NCCs it is often easier to split the NCCs into *atomic NCCs* (u,v) where u,v are variables or meta-variables (of any kind): For a constraint tuple $(\Delta_1,\Delta_2,\Delta_3)$, let $split_{ncc}(\Delta_3) := \bigcup_{(s,d)\in\mathcal{S}}\{(u,v) \mid u \in Var_M(s), v \in CV_M(d)\}$, where $Var_M(s) := MV(s) \cup Var(s)$, 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. (see Fig. 2 for the definition CV_M). For an atomic NCC (u,v) and a ground substitution ρ, let $Var_A(\rho(u)) = Var(\rho(u))$ and $CV_A(\rho(x)) = \{\rho(x)\}$, $CV_A(\rho(D)) = CV(\rho(D))$, $CV_A(\rho(E)) = LV(\rho(E))$, $CV_A(\rho(Ch)) = LV(\rho(Ch))$. Note that for an NCC (s,d) and a ground substitution ρ the equalities $Var(\rho(s)) = \{Var_A(\rho(u)) \mid u \in Var_M(s)\}$ and $CV(\rho(d)) = \{CV_A(\rho(u)) \mid u \in CV_M(d)\}$ hold. For instance, a constraint tuple $\Delta = (\Delta_1,\Delta_2,\Delta_3)$ is satisfiable iff $split_{ncc}(\Delta_3)$ does not contain a pair (u,u) where u is a variable $\mathtt{x}$, a meta-variable X, or an E-variable with $E \in \Delta_2$.

For example, for $\Delta = (\emptyset,\emptyset,\{(\texttt{var}\ Z, \lambda X.\lambda Y.[\cdot]), (\texttt{letrec}\ E\ \texttt{in}\ S, \texttt{letr}\ Z.\texttt{var}\ Z; E\ \texttt{in}\ [\cdot])\})$ we have $split_{ncc}(\Delta_3) = \{(Z,X),(Z,Y),(E,Z),(S,Z),(E,E),(S,E)\}$ which is satisfiable, but for $\Delta = (\emptyset,\{E\},\{(\texttt{var}\ Z, \lambda Z.\lambda Y.[\cdot]), (\texttt{letrec}\ E\ \texttt{in}\ S, \texttt{letr}\ Z.\texttt{var}\ Z; E\ \texttt{in}\ [\cdot])\})$ we have $split_{ncc}(\Delta_3) = \{(Z,Z),(Z,Y),(E,Z),(S,Z),(E,E),(S,E)\}$ which is not satisfiable since $(Z,Z) \in split_{ncc}(\Delta_3)$ and also since $(E,E) \in split_{ncc}(\Delta_3)$ where E must not be instantiated by the empty environment since $E \in \Delta_2$.

We define letrec rewrite rules to rewrite LRS-expressions. The rules have occurrence restrictions for the meta-variables which make the corresponding unification and matching problems easier to solve. They are sufficient to express reductions and transformations of usual program calculi (see also [16]). The semantics of meta letrec rewrite rules are all ground instances of the rule which satisfy the corresponding constraints. The rules are always applied to the top of expressions, since the strategy and the corresponding positions are expressed by the contexts used in the left and right hand sides of the rules.

Definition 3.4. *Let ℓ, r be* LRSX*-expressions, Δ be a constraint tuple, s.t. $MV(\Delta) \subseteq MV(\ell) \cup MV(r)$, and n be a name. Then $\ell \xrightarrow{n}_\Delta r$ is called a* meta letrec rewrite rule*, provided the following restrictions hold: For expressions ℓ and r, every variable of type S occurs at most twice in an expression; every variable of kind E, Ch, D occurs at most once in an expression; and Ch-variables occurring in ℓ must occur in one* letr*-environment only, i.e. ℓ is of the form $d[\texttt{letr}\ Ch_1[x_1,s_1];\ \ldots;\ Ch_k[x_k,s_k];\ env\ \texttt{in}\ t]$ s.t. d,t,env,s_i do not contain any Ch-variable. Furthermore, for any ground substitution ρ that satisfies Δ, $\rho(\ell)$ fulfills the LVC iff $\rho(r)$ fulfills the LVC. A meta letrec rewrite rule represents a (perhaps infinite) set of rewrite rules, i.e. the semantics is: $\gamma(\ell \xrightarrow{n}_\Delta r) := \{(\rho(\ell),\rho(r)) \mid \rho$ is a ground substitution for ℓ, r, s.t. $\rho(\ell),\rho(r)$ fulfill the LVC, ρ satisfies $\Delta\}$. Given a set $\mathcal{S}$ of meta letrec rewrite rules, we write $\mathtt{s} \xrightarrow{n} \mathtt{t}$ if $(\mathtt{s},\mathtt{t}) \in \gamma(\ell \xrightarrow{n}_\Delta r)$ with $\ell \xrightarrow{n}_\Delta r \in \mathcal{S}$. We write $\mathtt{s} \to \mathtt{t}$ if some rule named n exists in $\mathcal{S}$ s.t. $\mathtt{s} \xrightarrow{n} \mathtt{t}$.*

Example 3.5. The reduction rule used in Examples 2.5 and 3.3 and the reduction rules from Example 3.2 can be written as meta letrec rewrite rules:

- $\texttt{letr}\ E;\ Ch[X_1, \texttt{app}\ (\lambda X.S_0)\ S_1]\ \texttt{in}\ A[\texttt{var}\ X_1]$
  $\xrightarrow{\text{lbeta}}_{(\emptyset,\emptyset,\{(S_1,\lambda X.[\cdot])\})}\ \texttt{letr}\ E;\ Ch[X_1, \texttt{letr}\ X.S_1\ \texttt{in}\ S_0]\ \texttt{in}\ A[\texttt{var}\ X_1]$
- $\texttt{letr}\ E_1\ \texttt{in}\ \texttt{letr}\ E_2\ \texttt{in}\ S \xrightarrow{\text{llet-in}}_{(\emptyset,\{E_1,E_2\},\{(\texttt{letrec}\ E_1\ \texttt{in}\ c, \texttt{letrec}\ E_2\ \texttt{in}\ [\cdot])\})}\ \texttt{letr}\ E_1; E_2\ \texttt{in}\ S$
- $\texttt{letr}\ E;\ Ch_1[X_n, \lambda X.S];\ Ch[Y, A_1[\texttt{var}\ X_n]]\ \texttt{in}\ A[\texttt{var}\ Y]$
  $\xrightarrow{\text{cp-e}}_{(\{A_1\},\emptyset,\emptyset)}\ \texttt{letr}\ E;\ Ch_1[X_n, \lambda X.S];\ Ch[Y, A_1[\lambda X.S]]\ \texttt{in}\ A[\texttt{var}\ Y]$

Note that the NCC in rule named llet-in is needed to ensure that the rule does not introduce a capture of variables occurring in the environment E_1 by bindings from environment E_2.

To apply a meta letrec rewrite rule $\ell \xrightarrow{n}_\Delta r$ to LRS-expression $\mathtt{s}$, we need to find a ground substitution ρ s.t. $\rho(\ell) \sim_{let} \mathtt{s}$ and ρ satisfies Δ. This task can be performed by a matching algorithm and also by the unification algorithm from [16]. However, our goal is to apply meta letrec rewrite rules to constrained meta-expressions, i.e. for constrained expression (s,∇), we want to compute successors (t_i,∇_i') of (s,∇) w.r.t. $\ell \xrightarrow{n}_\Delta r$. This rewriting has to be sound, i.e. whenever $\mathtt{t} \in (t_i,\nabla_i')$, then there exists $\mathtt{s} \in (s,\nabla)$ s.t. $\mathtt{s} \xrightarrow{n} \mathtt{t} \in \gamma(\ell \xrightarrow{n}_\Delta r)$. Thus we have to guarantee that if (s,∇) is symbolically rewritten to (t_i,∇_i'), then this is also possible for every ground instance of s which satisfies ∇. We therefore introduce the letrec matching problem and in the subsequent section an algorithm to solve this problem.

Usually matching means to solve directed equations of the form $s \trianglelefteq t$ where s is a meta-expression with meta-variables and t is a ground expression. However, our matching equations are of the form $s \trianglelefteq t$ where s is a meta-expression with *instantiable meta-variables* and t is meta-expression with meta-variables which are treated like "meta-constants". We thus distinguish two sets of meta-variables, instantiable meta-variables and fixed meta-variables. We use blue font for instantiable meta-variables and red font and underlining for fixed meta-variables. With $MV_I(\cdot)$ and $\underline{MV_F}(\cdot)$ we denote functions to compute the sets.

Definition 3.6. *A* letrec matching problem *(LMP, for short) is a tuple $P{=}(\Gamma,\Delta,\nabla)$ where Γ is a set of matching equations $s \trianglelefteq t$ s.t. $MV_I(t) = \emptyset$; $\Delta{=}(\Delta_1,\Delta_2,\Delta_3)$ is a constraint tuple, called* needed constraints*; $\nabla{=}(\nabla_1,\nabla_2,\nabla_3)$ is a constraint tuple, called* given constraints*, where $MV_I(\nabla_i){=}\emptyset$ for $i = 1, 2, 3$, ∇ is satisfiable, for all expressions in Γ, the LVC must hold, and every instantiable variable of kind S occurs at most twice in Γ; every instantiable variable of kind E, Ch, D occurs at most once in Γ. A* matcher *of P is a substitution σ where $\mathtt{Dom}(\sigma) = MV_I(\Gamma)$, $MV_I(\sigma(s)) = \emptyset$ and $\underline{MV_F}(\sigma(s)) \subseteq \underline{MV_F}(P)$ for all $s \trianglelefteq t \in \Gamma$, s.t. for any ground substitution ρ with $\mathtt{Dom}(\rho) = \underline{MV_F}(P)$ which satisfies ∇, $\rho(\sigma(s)), \rho(t)$ fulfill the LVC for all $s \trianglelefteq t \in \Gamma$, we have $\rho(\sigma(s)) \sim_{let} \rho(t)$ for all $s \trianglelefteq t \in \Gamma$ and there exists a ground substitution ρ_0 with $\mathtt{Dom}(\rho_0) = MV_I(\rho(\sigma(\Delta)))$ s.t. $\rho_0(\rho(\sigma(\Delta)))$ is satisfied.*

If $MV_I(\Gamma) = MV_I(\Delta)$, then the definition of a matcher ensures that the given constraints ∇ imply the needed constraints Δ.

Example 3.7. The LMP $(\{s \trianglelefteq t\}, \Delta, \nabla)$ with $s = \mathtt{letr}\ E_1\ \mathtt{in}\ S_1$, $t = \mathtt{letr}\ \underline{E_2}\ \mathtt{in}\ \underline{S_2}$, $\Delta = (\emptyset, \{E_1\}, \{(S_1, \mathtt{letr}\ E_1\ \mathtt{in}\ [\cdot])\})$, and $\nabla = (\emptyset, \{\underline{E_2}\}, \emptyset)$ has no matcher: The substitution $\sigma = \{E_1 \mapsto \underline{E_2}, S_1 \mapsto \underline{S_2}\}$ is not a matcher, since the given constraints do not imply the needed constraints: For instance, for $\rho = \{\underline{E_2} \mapsto \mathtt{x.var\ x}, \underline{S_2} \mapsto \mathtt{var\ x}\}$ we have $\rho(\sigma(s)) = \rho(t)$, ρ satisfies ∇, but $\rho(\sigma(\Delta))$ is not satisfied, since the NCC $\rho(\sigma((S_1, \mathtt{letr}\ E_1\ \mathtt{in}\ [\cdot]))) = (\mathtt{var\ x}, \mathtt{letr\ x.var\ x\ in}\ [\cdot])$ is violated. However, the substitution σ is a matcher of the LMP $(s \trianglelefteq t, \Delta, \nabla')$ with $\nabla' = (\emptyset, \{\underline{E_2}\}, \{(\underline{S_2}, \mathtt{letr}\ \underline{E_2}\ \mathtt{in}\ [\cdot])\})$.

Note that the unification algorithm in [16] cannot be reused for matching, since its occurrence restrictions are too strong (fixed meta-variables may occur more often and chain variables may occur on the right hand sides of matching equations) and the algorithm cannot infer whether the given constraints ∇ imply the needed constraints Δ.

As a further note, we explain the role of the additional substitution ρ_0 in the definition of a matcher. It is needed for the case that the transformation or reduction introduces "fresh" variables. E.g., in

$$\mathtt{letr}\ X.c\ S_1\ \mathtt{in}\ S_2 \rightarrow_{(\emptyset,\emptyset,\Delta_3)} \mathtt{letr}\ X.c\ (\mathtt{var}\ Y); Y.S_1\ \mathtt{in}\ S_2$$

with $\Delta_3 = \{(\mathtt{var}\ X, \lambda Y.[\cdot]), (S_1, \lambda Y.[\cdot]), (S_2, \lambda Y.[\cdot])\}$, the constraints ensure that Y is fresh. Matching the left hand side of the rule against some expression, for instance, $\mathtt{letr\ u}.c\ (\mathtt{var\ y})\ \mathtt{in\ var\ u}$, will not instantiate the variable Y. Thus, after instantiation, the NCCs in Δ_3 become $\{(\mathtt{var\ u}, \lambda Y.[\cdot]), (\mathtt{var\ v}, \lambda Y.[\cdot])\}$. Validity depends on the instantiation of Y. The definition of a matcher allows us to choose an instance that satisfies the constraints (e.g. $\rho_0 = \{Y \mapsto \mathtt{w}\}$). 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. We show that a matcher indeed can be used to apply meta letrec rewrite rules to constrained expressions:

Proposition 3.8. *Let (s, ∇) be a constrained expression, $\ell \xrightarrow{n}_\Delta r$ be a meta letrec rewrite rule, σ be a matcher for $(\{\ell \trianglelefteq s\}, \Delta, \nabla)$, and ρ be a ground substitution, s.t. ρ satisfies ∇, $\rho(s)$ and $\rho(\sigma(\ell))$ fulfill the LVC. Then there exists a ground substitution ρ_0 s.t. $\rho(s) \xrightarrow{n} \rho_0(\rho(\sigma(r))) \in \gamma(\ell \xrightarrow{n}_\Delta r)$.*

4 Solving the Letrec Matching Problem

We present the algorithm MatchLRS. A *state* of MatchLRS is a tuple $(Sol, \Gamma, \Delta, \nabla)$ where Sol is a computed substitution and (Γ, Δ, ∇) is a LMP, where Γ consists of expression-, environment, binding-, and variable-equations. For (Γ, Δ, ∇), the state is initialized with $(Id, \Gamma, \Delta, \nabla)$ where Id is the identity. A final state is of the form $(Sol, \emptyset, \Delta, \nabla)$. The output of MatchLRS is either a final state or $Fail$. The rules of MatchLRS are inference rules $\frac{S}{S_1 \mid \dots \mid S_n}$ s.t. for given state S, the algorithm non-deterministically branches into derived states $S_1, \dots, S_n$. This non-determinism is don't know non-determinism. Rule application between the rules is don't care non-determinism. Variables occurring in $S_1, \dots, S_n$ but not in S

$$\text{(SolX)}\ \frac{(Sol,\Gamma\dot\cup\{X \trianglelefteq x\},\Delta)}{(Sol\circ\{X\mapsto x\},\Gamma[x/X],\Delta[x/X])} \quad \text{(SolS)}\ \frac{(Sol,\Gamma\dot\cup\{S \trianglelefteq s\},\Delta)}{(Sol\circ\{S\mapsto s\},\Gamma[s/S],\Delta[s/S])} \quad \text{(DecH)}\ \frac{\Gamma\dot\cup\{x.s \trianglelefteq y.t\}}{\Gamma\dot\cup\{x \trianglelefteq y, s \trianglelefteq t\}}$$

$$\text{(DecL)}\ \frac{\Gamma\dot\cup\{\texttt{letr}\,env\,\texttt{in}\,s \trianglelefteq \texttt{letr}\,env'\,\texttt{in}\,t\}}{\Gamma\dot\cup\{env \trianglelefteq env', s \trianglelefteq t\}} \quad \text{(DecF)}\ \frac{\Gamma\dot\cup\{f\,s_1\dots s_n \trianglelefteq f\,t_1\dots t_n\}}{\Gamma\dot\cup\{s_1 \trianglelefteq t_1,\dots,s_n \trianglelefteq t_n\}} \quad \text{(DecD)}\ \frac{\Gamma\dot\cup\{\underline{D}[s] \trianglelefteq \underline{D}[t]\}}{\Gamma\dot\cup\{s \trianglelefteq t\}}$$

$$\text{(CxPx)}\ \frac{(Sol,\Gamma\dot\cup\{D[s] \trianglelefteq \underline{D'}[s']\},\Delta,\nabla)}{(Sol\circ\sigma,\Gamma\dot\cup\{D''[s] \trianglelefteq s'\},\Delta\sigma,\nabla)\ \text{s.t.}\ \sigma{=}\{D\mapsto\underline{D'}[D'']\},cl(D''){=}cl(D)} \ \begin{array}{l}\text{if } D{\in}\Delta_1 \iff \underline{D'}{\in}\nabla_1\\ \text{and } cl(\underline{D'}) \le cl(D)\end{array}$$

$$\text{(ElX)}\ \frac{\Gamma\dot\cup\{x \trianglelefteq x\}}{\Gamma} \quad \text{(CxCG)}\ \frac{(Sol,\Gamma\dot\cup\{D[s] \trianglelefteq \underline{D'}[s']\},\Delta,\nabla)}{(Sol\circ\sigma,\Gamma\dot\cup\{s \trianglelefteq \underline{D'}[s']\},\Delta\sigma,\nabla)\ \text{where}\ \sigma=\{D\mapsto[\cdot]\}}\ \text{if } D{\notin}\Delta_1,\ cl(\underline{D'}) > cl(D)$$

$$\text{(ElS)}\ \frac{\Gamma\dot\cup\{\underline{S} \trianglelefteq \underline{S}\}}{\Gamma} \quad \text{(CxGuess)}\ \frac{(Sol,\Gamma\dot\cup\{D[s] \trianglelefteq t\},\Delta,\nabla)}{\begin{array}{c}(Sol\circ\{D\mapsto[\cdot]\},\Gamma\dot\cup\{s \trianglelefteq t\},\Delta[[\cdot]/D],\nabla)\\ \mid (Sol,\Gamma\dot\cup\{D[s] \trianglelefteq t\},(\Delta_1\dot\cup\{D\},\Delta_2,\Delta_3),\nabla)\end{array}}\ \begin{array}{l}\text{if } D \notin \Delta_1,\ t \neq \underline{D'}[s] \text{ with}\\ \underline{D'} \notin \nabla_1 \text{ or } cl(\underline{D'}) > cl(D)\end{array}$$

$$\text{(CxF)}\ \frac{(Sol,\Gamma\dot\cup\{D[s] \trianglelefteq f\ s_1\dots s_n\},\Delta,\nabla)}{\left|\begin{array}{l}(Sol\circ\sigma_i,\Gamma\dot\cup\{D'[s] \trianglelefteq s_i\},\Delta\sigma_i,\nabla)\ \text{s.t.}\ D',X_1,\dots,X_m\ \text{are fresh},\ cl(D')=cl(D),\\ \text{and}\ \sigma_i=\{D\mapsto f\,s_1\dots s_{i-1}\,X_1\dots X_m.D'\,s_{i+1}\dots s_n\},\end{array}\right.}\ \text{if } D{\in}\Delta_1$$

$$i \in I,\ \text{where}\ I = sp(f),\ \text{if}\ cl(D){=}\mathcal{A},\ I=\{i \mid oar(f)(i)=0\}\ \text{if}\ cl(D){=}\mathcal{T},\ I=\{i \mid oar(f)(i)\neq V\}\ \text{if}\ cl(D)=\mathcal{C}$$

$$\text{(CxL)}\ \frac{(Sol,\Gamma\dot\cup\{D[s] \trianglelefteq \texttt{letr}\ env\ \texttt{in}\ s'\},\Delta,\nabla)}{(Sol\circ\sigma,\Gamma\cup\{D'[s] \trianglelefteq s'\},\Delta\sigma,\nabla)\ \text{s.t.}\ \sigma=\{D\mapsto\texttt{letr}\ env\ \texttt{in}\ D'\},cl(D')=cl(D)}\ \begin{array}{l}\text{if } D \in \Delta_1,\\ cl(D) \ge T\end{array}$$
$$\mid (Sol\circ\sigma,\Gamma\cup\{E;Ch[X,D'[s]] \trianglelefteq env\},\Delta\sigma,\nabla)\ \text{s.t.}$$
$$\sigma=\{D{\mapsto}\texttt{letr}\ E;Ch[X,D']\ \texttt{in}\ s'\},cl(D'){=}cl(D),cl(Ch){=}\mathcal{A}$$

Fig. 3. Rules of MatchLRS for expression and binding equations

are always meant as fresh variables. The non-failure rules of MatchLRS are shown in Figs. 3 and 4. Rules (SolveX) and (SolveS) solve, and (ElX) and (ElS) 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 $D[s] \trianglelefteq t$, where (CxPx) covers the case that t is $\underline{D'}[t']$ and $\underline{D'}$ is a prefix of D where D must be at least as general as $\underline{D'}$. If $\underline{D'}$ is non-empty, but D may be empty, then rule (CxGuess) is applicable. If the class of $\underline{D'}$ 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 `letr`-expression. Rules (EnvEm) and (ElE) 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 $env = b; env'$, $env = \underline{Ch}[y,s]; env$, or $env = \underline{E'}; env$ with $\underline{E'} \in \nabla_2$. 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 instantiable 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 and instantiable chain-variable.

In Fig. 5 the failure rules of MatchLRS are defined. The NCC-implication check (used in rule (NCCFail)) decides whether the given NCCs imply the needed NCCs of a LMP:

Definition 4.1. *Let $(Sol,\emptyset,\Delta,\nabla)$ be a final state of* MatchLRS *for input $(\Gamma_I,\Delta_I,\nabla_I)$. The set $NCC_{lvc} := \bigcup\{NCC_{lvc}(r) \mid r \in \{Sol(s),t\}, s \trianglelefteq t \in \Gamma_I\}$, where $NCC_{lvc}(\cdot)$ on expressions is defined in Fig. 6, contains atomic NCCs that are implied by the LVC. The* NCC-implication check *is* valid *iff for all $(u,v) \in split_{ncc}(\Delta_3)$ one of the following cases holds:*

1. *$u = \mathsf{x}$ and $v = \mathsf{y}$ where $\mathsf{x} \neq \mathsf{y}$.*
2. *$(u,v) \in split_{ncc}(\nabla_3) \cup NCC_{lvc}$.*
3. *$u = v$ and $u = Ch$ or $u = D$ or $u = E$ with $E \notin \Delta_2$.*
4. *$u \neq v$ and $u \in \{Ch, S, D, E, X\}$.*
5. *$u \neq v$ and $v = Ch$, or $v = D$, or $v = E$, or $v = X$.*
6. *$u = \underline{E}$ or $u = \underline{Ch}$ with $cl(Ch) = Triv$ and $(u,u) \in split_{ncc}(\nabla_3) \cup NCC_{lvc}$.*
7. *$v \in \{\underline{E}, \underline{Ch}, \underline{D}\}$ and $(v,v) \in split_{ncc}(\nabla_3) \cup NCC_{lvc}$.*
8. *(u,v) is $(\underline{X},\mathsf{y})$, $(\mathsf{x},\underline{Y})$, $(\underline{X},\underline{Y})$, $(\mathsf{x},\underline{D})$, $(\underline{X},\underline{D})$, $(\mathsf{x},\underline{E})$, $(\underline{X},\underline{E})$, $(\mathsf{x},\underline{Ch})$, $(\underline{X},\underline{Ch})$, $(\underline{Ch_1},\mathsf{x})$, $(\underline{Ch_1},\underline{X})$, $(\underline{Ch_1},\underline{E})$, $(\underline{Ch_1},\underline{D})$, or $(\underline{Ch_1},\underline{Ch_2})$ where $cl(\underline{Ch_1}) = Triv$ and in all cases $(v,u) \in split_{ncc}(\nabla_3) \cup NCC_{lvc}$.*

$$\text{(EnvAE)}\ \frac{(Sol,\Gamma\dot\cup\{E_1,\dots,E_n \trianglelefteq \emptyset\},\Delta)}{(Sol\circ\sigma,\Gamma,\Delta\sigma) \text{ s.t. } \sigma=\{E_i\mapsto\emptyset\}_{i=1}^{n}} \text{ if } \forall i{:}E_i\notin\Delta_2 \quad \text{(EIE)}\ \frac{\Gamma\dot\cup\{\underline{E};env_1 \trianglelefteq \underline{E};env_2\}}{\Gamma\dot\cup\{env_1\trianglelefteq env_2\}} \quad \text{(EnvEm)}\ \frac{\Gamma\dot\cup\{\emptyset\trianglelefteq\emptyset\}}{\Gamma}$$

$$\text{(EnvE)}\ \frac{(Sol,\Gamma\dot\cup\{env \trianglelefteq \underline{E};env'\},\Delta,\nabla)}{\Big|_{\forall E':env=E';env_1 \text{ and } \underline{E}\notin\nabla_2 \implies E'\notin\Delta_2} (Sol\circ\sigma,\Gamma\dot\cup\{E'';env_1\trianglelefteq env'\},\Delta\sigma,\nabla) \text{ with } \sigma=\{E'\mapsto E'';\underline{E}\}} \quad \text{if } env\neq\underline{E};env_1,\ \exists E{:}env=E;env_1 \text{ s.t. } \underline{E}\notin\nabla_2\implies E\notin\Delta_2$$

$$\text{(SolveE)}\ \frac{(Sol,\Gamma\dot\cup\{E\trianglelefteq env\},\Delta,\nabla)}{(Sol\circ\sigma,\Gamma,\Delta\sigma,\nabla) \text{ where } \sigma=\{E\mapsto env\}} \quad \text{if } E\in\Delta_2 \iff env \text{ is non-empty}$$

$$\text{(EnvB)}\ \frac{(Sol,\Gamma\dot\cup\{env\trianglelefteq b;env'\},\Delta,\nabla)}{\begin{array}{l}
\Big|_{\forall b':env=b';env''} (Sol,\Gamma\dot\cup\{b'\trianglelefteq b, env''\trianglelefteq env'\},\Delta,\nabla) \\
\mid\ \Big|_{\forall E:env=E;env''} (Sol\circ\sigma,\Gamma\dot\cup\{E';env''\trianglelefteq env'\},\Delta\sigma,\nabla) \text{ where } \sigma=\{E\mapsto b;E'\} \\
\mid\ \Big|_{\forall Ch:env=Ch[y,s];env''} (Sol\circ\sigma,\Gamma\dot\cup\{y.D[s]\trianglelefteq b, env''\trianglelefteq env'\},\Delta\sigma,\nabla) \\
\qquad \text{where } \sigma=\{Ch[\cdot_1,\cdot_2]\mapsto[\cdot_1].D[\cdot_2]\} \text{ and } cl(D)=cl(Ch) \\
\mid\ \Big|_{\forall Ch:env=Ch[y,s];env''} (Sol\circ\sigma,\Gamma\dot\cup\{y.D[X]\trianglelefteq b, Ch_2[X,s];env''\trianglelefteq env'\},\Delta\sigma,\nabla) \\
\qquad \text{where } \sigma=\{Ch[\cdot_1,\cdot_2]\mapsto[\cdot_1].D[X];Ch_2[X,\cdot_2]\},\ cl(D)=cl(Ch_2)=cl(Ch) \\
\mid\ \Big|_{\forall Ch:env=Ch[y,s];env''} (Sol\circ\sigma,\Gamma\dot\cup\{Y.D_1[X]\trianglelefteq b, Ch_1[y,D_2[Y]];Ch_2[X,s];env''\trianglelefteq env'\},\Delta\sigma,\nabla) \\
\qquad \text{where } \sigma=\{Ch[\cdot_1,\cdot_2]\mapsto Ch_1[\cdot_1,D_2[Y]];Y.D_1[X];Ch_2[X,\cdot_2]\},\ cl(D_i)=cl(Ch_i)=cl(Ch) \\
\mid\ \Big|_{\forall Ch:env=Ch[y,s];env''} (Sol\circ\sigma,\Gamma\dot\cup\{X_1.D[s]\trianglelefteq b, Ch_1[y,D'[X_1]];env''\trianglelefteq env'\},\Delta\sigma,\nabla) \text{ where} \\
\qquad \sigma=\{Ch[\cdot_1,\cdot_2]\mapsto Ch_1[\cdot_1,D'[X_1]];X_1.D[\cdot_2]\},\ cl(D)=cl(D')=cl(Ch_1)=cl(Ch)
\end{array}}$$

$$\text{(EnvC)}\ \frac{(Sol,\Gamma\dot\cup\{env_1\trianglelefteq \underline{Ch}[y,s];env_2\},\Delta,\nabla)}{\begin{array}{l}
\Big|_{\forall \underline{Ch}:env_1=\underline{Ch}[y',s'];env_1'} (Sol\circ\sigma,\Gamma\dot\cup\{y'\trianglelefteq y, s'\trianglelefteq s, env_1'\trianglelefteq env_2\},\Delta\sigma,\nabla) \\
\mid\ \Big|_{\forall E:env_1=E;env_1'} (Sol\circ\sigma,\Gamma\dot\cup\{E';env_1'\trianglelefteq env_2\},\Delta\sigma,\nabla) \text{ where } \sigma=\{E\mapsto E';\underline{Ch}[y,s]\} \\
\mid\ \Big|_{\forall Ch_1:env_1=Ch_1[y_1,s_1];env_1' \text{ and } cl(Ch_1)\geq cl(\underline{Ch})} \Big|_{\forall(d,t)\in split_{cl(Ch_1)}(s)} (Sol\circ\sigma,\Gamma\dot\cup\{env_1'\trianglelefteq env_2, y_1\trianglelefteq y, s_1\trianglelefteq t\},\Delta\sigma,\nabla) \text{ with } \sigma=\{Ch_1[\cdot,\cdot]\mapsto\underline{Ch}[\cdot,d[\cdot_2]]\} \\
\mid\ \Big|_{\forall Ch_1:env_1=Ch_1[y_1,s_1];env_1' \text{ and } cl(Ch_1)\geq cl(\underline{Ch})} \Big|_{\forall(d,t)\in split_{cl(Ch_1)}(s)} (Sol\circ\sigma,\Gamma\dot\cup\{Ch_2[y_1,D[\mathtt{var}\ y]];env_1'\trianglelefteq env_2, s_1\trianglelefteq t\},\Delta\sigma,\nabla) \\
\qquad \text{where } \sigma=\{Ch_1[\cdot_1,\cdot_2]\mapsto Ch_2[\cdot_1,D[\mathtt{var}\ y]];\underline{Ch}[y,d[\cdot_2]]\},\ cl(D)=cl(Ch_2)=cl(Ch_1) \\
\mid\ \Big|_{\forall Ch_1:env_1=Ch_1[y_1,s_1];env_1' \text{ and } cl(Ch_1)\geq cl(\underline{Ch})} (Sol\circ\sigma,\Gamma\dot\cup\{Ch_2[X,s_1];env_1'\trianglelefteq env_2, D[\mathtt{var}\ X]\trianglelefteq s, y_1\trianglelefteq y\},\Delta\sigma,\nabla) \\
\qquad \text{where } \sigma=\{Ch_1[\cdot_1,\cdot_2]\mapsto\underline{Ch}[\cdot_1,s];Ch_2[X,\cdot_2]\} \text{ and } cl(D)=cl(Ch_2)=cl(Ch_1) \\
\mid\ \Big|_{\forall Ch_1:env_1=Ch_1[y_1,s_1];env_1' \text{ and } cl(Ch_1)\geq cl(\underline{Ch})} (Sol\circ\sigma,\Gamma\dot\cup\{Ch_2[y_1,D[X]];Ch_3[Y,s_1];env_1'\trianglelefteq env_2, D_1[Y]\trianglelefteq s\},\Delta\sigma,\nabla) \\
\qquad \text{where } \sigma=\{Ch_1[\cdot_1,\cdot_2]\mapsto Ch_2[\cdot_1,D[X]];\underline{Ch}[y,s];Ch_3[Y,\cdot_2]\} \text{ and } cl(D)=cl(D_1)=cl(Ch_2)=cl(Ch_3)=cl(Ch_1)
\end{array}}$$

Fig. 4. Rules of MATCHLRS for environment equations. In rule (EnvC) the function $split_{\mathcal{K}}$ is defined as follows: $split_{Triv}(t)=\{([\cdot],t)\}$; $split_{\mathcal{A}}(f\,s_1\dots s_n)=\{([\cdot],(f\,s_1\dots s_n))\}\cup\{(f\,s_1\dots s_{i-1}\,d\,s_{i+1}\dots s_n,s')\mid(d,s')\in split_{\mathcal{A}}(s_i), i\in sp(f)\}$; $split_{\mathcal{A}}(\underline{A}[s])=\{([\cdot],\underline{A}[s])\}\cup\{(\underline{A}[d],s')\mid(d,s')\in split_{\mathcal{A}}(s)\}$; and $split_{\mathcal{A}}(t)=\{([\cdot],t)\}$, if $t\neq(f\,s_1\dots s_n)$ and $t\neq\underline{A}[s]$.

Let $(Sol, \Gamma, \Delta, \nabla)$ be the state of MATCHLRS and $(\Gamma_I, \Delta_I, \nabla_I)$ be the LMP from the input. The algorithm delivers *Fail* if Γ contains an equation

(VarFail)	$\mathsf{x} \trianglelefteq \mathsf{y}$, $\mathsf{x} \trianglelefteq \underline{Y}$, $\underline{X} \trianglelefteq \mathsf{x}$, or $\underline{X} \trianglelefteq \underline{Y}$.
(ExFailF)	$(f\, s_1 \ldots s_n) \trianglelefteq t$ s.t. $t = (f'\, t_1 \ldots t_m)$ and $f \neq f'$, $t = \mathtt{letr}\; env\; \mathtt{in}\, t'$, $t = \underline{S}$, or $t = \underline{D}[s]$.
(ExFailL)	$\mathtt{letr}\; env\; \mathtt{in}\, s \trianglelefteq t$ and t is $(f\, s_1 \ldots s_n)$, $\underline{S}$, or $\underline{D}[s]$.
(ExFailS)	$\underline{S} \trianglelefteq t$ where $t = (f\, s_1 \ldots s_n)$, $t = \mathtt{letr}\ env\ \mathtt{in}\ s$, $t = \underline{S_1}$ with $\underline{S} \neq \underline{S_1}$, or $t = \underline{D}[t']$.
(CxFailF)	$\underline{D}[s] \trianglelefteq t$ where $t \neq \underline{D}[s']$
(CxFailI)	$D[s] \trianglelefteq t$ where $D \in \Delta_1$ and $t{=}f$ with $ar(f){=}0$, or $t{=}\underline{S}$, or $t{=}\underline{D_2}[t']$ with $cl(\underline{D_2}){>}cl(D)$, or $t{=}(f\, s_1 \ldots s_n)$ with $cl(D){\in}\{\mathcal{A}, \mathcal{T}\}$, $n{>}\,0$, $oar(f){=}(l_1, \ldots, l_n)$, and $\forall i : (l_i{\neq}0)$, or $t{=}(f\, s_1 \ldots s_n)$ with $cl(D){=}\mathcal{A}$ and $sp(f) = \emptyset$, or $t{=}(f\, s_1 \ldots s_n)$ with $n{>}0$ and $oar(f)(i){=}V$ for all i, or $t{=}\underline{D_2}[t']$ with $\underline{D_2}{\notin}\nabla_1$, or $t{=}\mathtt{letr}\; env\; \mathtt{in}\ t'$ and $cl(D){=}\mathcal{A}$.
(EFailEm)	$env \trianglelefteq \emptyset$ or $\emptyset \trianglelefteq env$ where env is non-empty.
(EFailB)	$b; env \trianglelefteq env'$ where $env' \neq b'; env''$.
(EFailCI)	$Ch_1[z, s]; env \trianglelefteq env'$ where $env' \neq b; env''$, and $env' \neq \underline{Ch_2}[z', s']; env''$ s.t. $cl(Ch_1){\geq}cl(\underline{Ch_2})$.
(EFailCFR)	$env \trianglelefteq \underline{Ch_1}[z, s]; env'$, where $env \neq E; env_1$, $env \neq Ch_2[z', s']; env_1$ with $cl(\underline{Ch_1}){\leq}cl(Ch_2)$, and $env \neq \underline{Ch_1}[z', s']; env_1$.
(EFailCFL)	$\underline{Ch}[z, s]; env \trianglelefteq env'$ where $env' \neq \underline{Ch}[z', s']$
(EFailEL)	$\underline{E}; env \trianglelefteq env'$ and $env' \neq \underline{E}; env''$.
(EFailER)	$env{\trianglelefteq}\underline{E}; env'$, $env{\neq}E'; env''$, and $env{\neq}\underline{E}; env''$.
(EFailEE)	$E_1; \ldots; E_n \trianglelefteq \underline{E'_1}; \ldots; \underline{E'_m}$, $\forall i{:}E_i{\in}\Delta_2$,$\forall i{:}\underline{E'_i}{\notin}\underline{\Delta_2}$.

If $\Gamma = \emptyset$ then MATCHLRS delivers *Fail* if

(LVCFail)	for $s \trianglelefteq t \in \Gamma_I$, $Sol(s)$ does not fulfill the LVC, or
(NCCFail)	the NCC-implication check (Def. 4.1) is invalid.

Fig. 5. Failure rules of MATCHLRS

$$
\begin{aligned}
NCC_{lvc}(s) = {} & \{(x, y) \mid x.s; y.s'; env \in \mathcal{E}\} \cup \{(x, y) \mid x.s; \underline{Ch}[y, s']; env \in \mathcal{E}\} \\
& \cup \{(x, y) \mid \underline{Ch}[x, s]; y.s'; env \in \mathcal{E}\} \cup \{(x, y) \mid \underline{Ch}[x, s]; \underline{Ch'}[y, s']; env \in \mathcal{E}\} \\
& \cup \{(x, \underline{E}) \mid x.s; \underline{E}; env \in \mathcal{E}\} \cup \{(x, \underline{E}) \mid \underline{Ch}[x, s]; \underline{E}; env \in \mathcal{E}\} \\
& \cup \{(x, \underline{Ch}) \mid x.s; \underline{Ch}[y, s]; env \in \mathcal{E}\} \cup \{(x, \underline{Ch}) \mid \underline{Ch'}[x, s]; \underline{Ch}[y, s]; env {\in} \mathcal{E}\} \\
& \cup \{(\underline{Ch}, \underline{E}) \mid \underline{Ch}[y, s]; \underline{E}; env \in \mathcal{E}, cl(\underline{Ch}) = Triv\} \\
& \cup \{(\underline{Ch_1}, \underline{Ch_2}) \mid \underline{Ch_1}[y, s]; \underline{Ch_2}[y', s']; env \in \mathcal{E}, cl(\underline{Ch_1}) = Triv\}
\end{aligned}
$$

Fig. 6. Computing the set $NCC_{lvc}(s)$ where $\mathcal{E}$ is the set of all `letr`-environments in s

Example 4.2. We illustrate MATCHLRS on the LMP $(\{s \trianglelefteq t\}, \Delta, \nabla)$ with

$$
\begin{array}{ll}
s = \mathtt{letr}\ Ch[X, S_1]\ \mathtt{in}\ S_2 & \Delta = (\Delta_1, \Delta_2, \Delta_3) = (\emptyset, \emptyset, \{(S_1, \lambda X.[\cdot])\}) \\
t = \mathtt{letr}\ \underline{Y}.\mathsf{app}\ \underline{S_3}\ \underline{S_4}\ \mathtt{in}\ \underline{S_5} & \nabla = (\nabla_1, \nabla_2, \nabla_3) = (\emptyset, \emptyset, \{(\underline{S_3}, \lambda \underline{Y}.[\cdot])\})
\end{array}
$$

where $cl(Ch) = \mathcal{A}$. After applying rules (DecL) and (SolveS), the state of MATCHLRS is $(\{S_2 \mapsto \underline{S_5}\}, Ch[X, S_1] \trianglelefteq \underline{Y}.\mathsf{app}\ \underline{S_3}\ \underline{S_4}, \Delta, \nabla)$. Now rule (EnvB) is applicable and branches into four states for the chain-variable Ch, where all but the first case result in *Fail*, since they imply that Ch contains more than one binding. For the remaining case, the state of MATCHLRS is $(\{S_2 \mapsto \underline{S_5}, Ch[\cdot_1, \cdot_2] \mapsto [\cdot_1].A[\cdot_2]\}, X.A[S_1] \trianglelefteq \underline{Y}.\mathsf{app}\ \underline{S_3}\ \underline{S_4}, \Delta, \nabla)$. Applying (DecH) and then (SolveX) results in $(\{S_2 \mapsto \underline{S_5}, Ch[\cdot_1, \cdot_2] \mapsto [\cdot_1].A[\cdot_2], X \mapsto \underline{Y}\}, A[\underline{S_1}] \trianglelefteq \mathsf{app}\ \underline{S_3}\ \underline{S_4}, \Delta, \nabla)$. Now rule (CxGuess) is applied which branches into two cases.

If A is guessed as empty, then the next state is $(\{S_2 \mapsto \underline{S_5}, Ch[\cdot_1, \cdot_2] \mapsto [\cdot_1].A[\cdot_2], X \mapsto \underline{Y}\}, S_1 \trianglelefteq \mathsf{app}\ \underline{S_3}\ \underline{S_4}, \Delta[\underline{Y}/X], \nabla)$. Applying (SolveS) yields $(\{S_2 \mapsto \underline{S_5}, Ch[\cdot_1, \cdot_2] \mapsto [\cdot_1].[\cdot_2], X \mapsto \underline{Y}, S_1 \mapsto \mathsf{app}\ \underline{S_3}\ \underline{S_4}\}, \emptyset, \Delta', \nabla)$ where $\Delta' = (\emptyset, \emptyset, \{(\mathsf{app}\ \underline{S_3}\ \underline{S_4}, \lambda\underline{Y}.[\cdot])\})$. However, the NCC-implication check fails since $split_{ncc}(\Delta_3) = \{(\underline{S_3}, \underline{Y}), (\underline{S_4}, \underline{Y})\}$, $split_{ncc}(\nabla_3) = \{(\underline{S_3}, \underline{Y})\}$, and $NCC_{lvc} = \emptyset$ and thus for the atomic NCC $(\underline{S_4}, \underline{Y}) \in split_{ncc}(\Delta_3)$ none of the cases of Definition 4.1 holds. Thus this branch ends with *Fail*.

In the second case A is added to the set Δ_1, i.e. with $\Delta'' = (\{A\}, \emptyset, \{(S_1, \lambda\underline{Y}.[\cdot])\})$ the new state is $(\{S_2 \mapsto \underline{S_5}, Ch[\cdot_1, \cdot_2] \mapsto [\cdot_1].A[\cdot_2], X \mapsto \underline{Y}\}, A[S_1] \trianglelefteq \mathsf{app}\ \underline{S_3}\ \underline{S_4}, \Delta'', \nabla)$. Rule (CxF) is applied and results in (assuming that $sp(\mathsf{app}) = \{1\}$) $(\{S_2 \mapsto \underline{S_5}, Ch[\cdot_1, \cdot_2] \mapsto [\cdot_1].A[\cdot_2], X \mapsto \underline{Y}, A \mapsto \mathsf{app}\ A'\ \underline{S_4}\}, A'[S_1] \trianglelefteq \underline{S_3}, \Delta'', \nabla)$. Now rule (CxGuess) is applied and branches into two cases: for the case that A' is guessed to be non-empty, rule (CxFailI) is applicable and leads to *Fail*, and for the case that A' is guessed to be empty, the next state is $(\{S_2 \mapsto \underline{S_5}, Ch[\cdot_1, \cdot_2] \mapsto [\cdot_1].A[\cdot_2], X \mapsto \underline{Y}, A \mapsto \mathsf{app}\ [\cdot]\ \underline{S_4}\}, S_1 \trianglelefteq \underline{S_3}, \Delta'', \nabla)$ and rule (SolveS) results in the state $(\{S_2 \mapsto \underline{S_5}, Ch[\cdot_1, \cdot_2] \mapsto [\cdot_1].A[\cdot_2], X \mapsto \underline{Y}, A \mapsto \mathsf{app}\ [\cdot]\ \underline{S_4}, S_1 \trianglelefteq \underline{S_3}\}, \emptyset, \Delta''', \nabla)$ where $\Delta''' = (\emptyset, \emptyset, \{(\underline{S_3}, \lambda\underline{Y}.[\cdot])\})$. The NCC-implication check is valid since $split_{ncc}(\Delta_3) = \{(\underline{S_3}, \underline{Y})\}$ and

$(\underline{S_3}, \underline{Y}) \in split_{ncc}(\nabla_3)$. Thus the algorithm delivers the matcher $\{S_2 \mapsto \underline{S_5}, Ch[\cdot_1, \cdot_2] \mapsto [\cdot_1].A[\cdot_2], X \mapsto \underline{Y}, A \mapsto \texttt{app}\ [\cdot]\ \underline{S_4}, S_1 \trianglelefteq \underline{S_3}\}$.

We define the notion of a matcher for an (intermediate) state of MatchLRS:

Definition 4.3. *For LMP $P = (\Gamma_I, \Delta_I, \nabla_I)$ and state $S = (Sol, \Gamma, \Delta, \nabla)$ of* MatchLRS *for input P, a* matcher of state S *is a substitution σ where $\mathtt{Dom}(\sigma) = MV_I(\Gamma)$, $\sigma(U) = \sigma(Sol(U))$ for all $U \in \mathtt{Dom}(Sol)$, $MV_I(\sigma(s)) = \emptyset$ and $\underline{MV_F}(\sigma(s)) \subseteq \underline{MV_F}(P)$ for all $s \trianglelefteq t \in \Gamma_I$, s.t. for any ground substitution ρ with $\mathtt{Dom}(\rho) = \underline{MV_F}(P)$ which satisfies ∇, $\rho(\sigma(s)), \rho(t)$ fulfill the LVC for all $s \trianglelefteq t \in \Gamma_I$, we have $\rho(\sigma(s)) \sim_{let} \rho(t)$ for all $s \trianglelefteq t \in \Gamma$, and there exists a ground substitution ρ_0 with $\mathtt{Dom}(\rho_0) = MV_I(\rho(\sigma(\Delta)))$ s.t. $\rho_0(\rho(\sigma(\Delta)))$ is satisfied.*

We show soundness of the NCC-implication check.

Lemma 4.4. *Let $S = (Sol, \emptyset, \Delta, \nabla)$ be a state of* MatchLRS *for input $P = (\Gamma_I, \Delta_I, \nabla_I)$ s.t. s, t fulfill the LVC for all $s \trianglelefteq t \in \Gamma_I$ and the NCC-implication check is valid for S. Let ρ be a ground substitution with $\mathtt{Dom}(\rho) = \underline{MV_F}(P)$ which satisfies ∇, $\rho(Sol(s)), \rho(t)$ fulfill the LVC for all $s \trianglelefteq t \in \Gamma_I$, and $\rho(Sol(s)) \sim_{let} \rho(t)$ for all $s \trianglelefteq t \in \Gamma_I$. Then there exists a ground substitution ρ_0 with $\mathtt{Dom}(\rho_0) = MV_I(\rho(\Delta))$ s.t. $\rho_0(\rho(\Delta))$ is satisfied.*

Proof. We first show that all atomic NCCs in NCC_{lvc} are satisfied by each ground substitution ρ which fulfills the conditions of the lemma. For $(x, y) \in NCC_{lvc}$, x, y are let-variables of the same environment and thus ρ must map x and y to distinct concrete variables, since otherwise the LVC is violated w.r.t. ρ. For $(x, \underline{E}) \in NCC_{lvc}$, either $\rho(\underline{E}) = \emptyset$ and thus $CV_A(\rho(\underline{E})) = \emptyset$, or $\rho(\underline{E}) = \mathsf{x}_1.\mathsf{s}_1; \ldots; \mathsf{x}_n; \mathsf{s}_n$ where $\rho(x) \neq \mathsf{x}_i$, since otherwise the LVC would be violated for the environment containing $\underline{E}$ and let-variable x. For $(x, \underline{Ch}) \in NCC_{lvc}$ either $\rho(\underline{Ch}) = [\cdot_1].\mathsf{d}[\cdot_2]$ where $CV_A(\rho(\underline{Ch})) = \emptyset$ and thus $(\rho(x), \rho(\underline{Ch}))$ is satisfied, or $\rho(\underline{Ch}) = [\cdot_1].\mathsf{d}[\mathsf{x}_1]; \ldots; \mathsf{x}_n.[\cdot_2]$ where $\rho(x) \neq \mathsf{x}_i$ must hold, since otherwise the LVC is violated for the environment that contains $\underline{Ch}$ and let-variable x. For $(\underline{Ch}, \underline{E}) \in NCC_{lvc}$, the atomic NCC is satisfied if $\rho(\underline{E}) = \emptyset$ or $\rho(\underline{Ch}) = [\cdot_1].[\cdot_2]$, and otherwise $Var_A(\rho(\underline{Ch}))$ is exactly the set of let-bound variables in $\rho(\underline{Ch})$ which must be pairwise disjoint from the let-bound variables in $\rho(\underline{E})$ since otherwise the LVC is violated for the environment containing $\underline{Ch}$ and $\underline{E}$. For $(\underline{Ch}, \underline{Ch'})$ the same argument applies: $Var_A(\rho(\underline{Ch}))$ is exactly the set of let-bound variables in $\rho(\underline{Ch})$ and $CV_A(\rho(\underline{Ch'}))$ is exactly the set of let-bound variables in $\rho(\underline{Ch'})$, and thus both sets must be pairwise disjoint to satisfy the LVC.

Now let $(u, v) \in split_{ncc}(\Delta_3)$ s.t. one of the cases of the NCC-implication check applies. We consider the different cases and use the following instantiation ρ_0 for instantiable meta-variables: $\rho_0(Ch) = [\cdot_1].[\cdot_2]$ for all Ch; $\rho_0(S) = \lambda\mathsf{x}_S.\mathsf{x}_S$ for a fresh variable x_S; $\rho_0(D) = [\cdot]$ if $D \notin \Delta_1$, and $\rho_0(D) = \mathsf{d}$ where d is a context with $CV(\mathsf{d}) = \emptyset$ (see Definition 2.3); $\rho_0(E) = \emptyset$ if $E \notin \Delta_2$ and $\rho_0(E) = \mathsf{x}_E.\mathtt{var}\ \mathsf{x}_E$, otherwise where x_E is a fresh variable; $\rho_0(X) = \mathsf{x}_X$ for a fresh variable x_X.

1. If $(u, v) = (\mathsf{x}, \mathsf{y})$, then the constraint is satisfied.
2. If $(u, v) \in split_{ncc}(\nabla_3) \cup NCC_{lvc}$ then $Var_A(\rho(u)) \cap CV_A(\rho(v)) = \emptyset$ and $\rho(u), \rho(v)$ are ground.
3. If $u = v$ and $u = Ch$ or $u = D$ or $u = E$ with $E \notin \Delta_2$, then $\rho(u) = \rho(v) = u$, and $Var_A(\rho_0(u)) = CV_A(\rho_0(u)) = \emptyset$.
4. If $u \neq v$ and $u = Ch$, or $u = S$, or $u = D$ or $u = E$, or $u = X$, then $Var_A(\rho_0(\rho(u))) = \rho_0(u)$ contains only fresh variables and these variables must be disjoint from $CV_A(\rho_0(\rho(v)))$.
5. If $u \neq v$ and $v = Ch$, or $v = D$, or $v = E$, or $v = X$, then $\rho_0(\rho(v)) = \rho_0(v)$ and $CV_A(\rho_0(v))$ contains only fresh variables which cannot occur in $\rho_0(\rho(u))$.
6. If $u = \underline{E}$ or $u = \underline{Ch}$ with $cl(Ch) = Triv$ and $(u, u) \in split_{ncc}(\nabla_3)$, then $Var_A(\rho(u)) \cap CV_A(\rho(u)) = \emptyset$ must hold which is only possible if $\rho(u) = \emptyset$ (for $u = \underline{E}$) or $\rho(u) = [\cdot_1].[\cdot_2]$ (for $u = \underline{Ch}$). In both cases $Var_A(\rho(u)) = \emptyset$ holds, and thus $Var_A(\rho_0(\rho(u))) \cap CV_A(\rho_0(\rho(v))) = \emptyset$ for any v.
7. If $v = \underline{E}$, $v = \underline{Ch}$, or $v = \underline{D}$ and $(v, v) \in split_{ncc}(\nabla_3)$, then $Var_A(\rho(v)) \cap CV_A(\rho(v)) = \emptyset$ must hold, which requires that $\rho(v) = \emptyset$ (for $v = \underline{E}$), $\rho(v) = [\cdot_1].\mathsf{d}[\cdot_2]$ with $CV_A(\mathsf{d}) = \emptyset$ (for $v = \underline{CC}$), $\rho(v) = \mathsf{d}$ with $CV_A(\mathsf{d}) = \emptyset$ (for $v = \underline{D}$). In all cases $CV_A(\rho(v)) = \emptyset$ and thus $Var_A(\rho_0(\rho(u))) \cap CV_A(\rho_0(\rho(v))) = \emptyset$.
8. For the case that $(v, u) \in split_{ncc}(\nabla_3) \cup NCC_{lvc}$ and (u, v) is of the form $(\underline{X}, \mathsf{y})$, $(\mathsf{x}, \underline{Y})$, $(\underline{X}, \underline{Y})$, $(\mathsf{x}, \underline{D})$, $(\underline{X}, \underline{D})$, $(\mathsf{x}, \underline{E})$, $(\underline{X}, \underline{E})$, $(\mathsf{x}, \underline{Ch})$, $(\underline{X}, \underline{Ch})$, $(\underline{Ch_1}, \mathsf{x})$, $(\underline{Ch_1}, \underline{X})$, $(\underline{Ch_1}, \underline{E})$, $(\underline{Ch_1}, \underline{D})$, or $(\underline{Ch_1}, \underline{Ch_2})$ where $cl(\underline{Ch_1}) = Triv$, it suffices to show that if $Var_A(\rho(v)) \cap CV_A(\rho(u)) = \emptyset$, then also $Var_A(\rho(u)) \cap CV_A(\rho(v)) = \emptyset$. For $(u, v) \in \{(\underline{X}, \mathsf{y}), (\mathsf{x}, \underline{Y}), (\underline{X}, \underline{Y})\}$ this holds since $Var_A(\mathsf{y}) = CV_A(\mathsf{y})$ for every variable y. For $(u, v) = (\mathsf{x}, \underline{U})$ or $(\underline{X}, \underline{U})$ where $\underline{U}$ is an $\underline{D}$-, $\underline{E}$-, or $\underline{Ch}$-variable, $CV_A(\rho(v)) \subseteq Var_A(\rho(v))$ and $Var_A(\rho(u)) = CV_A(\rho(u))$ and thus $Var_A(\rho(v)) \cap CV_A(\rho(u)) = \emptyset$ implies $Var_A(\rho(u)) \cap CV_A(\rho(v)) = \emptyset$. For $(u, v) = (\underline{Ch_1}, \mathsf{x})$ or $(u, v) = (\underline{Ch_1}, \underline{U})$ where $cl(\underline{Ch_1}) = Triv$ and $\underline{U}$ is an $\underline{X}$-, $\underline{E}$-, $\underline{D}$-, or $\underline{Ch}$-variable, we have $Var_A(\rho(u)) = CV_A(\rho(u))$ and also $CV_A(\rho(v)) \subseteq Var_A(\rho(v))$ and thus $Var_A(\rho(u)) = CV_A(\rho(u))$ and thus $Var_A(\rho(v)) \cap CV_A(\rho(u)) = \emptyset$ implies $Var_A(\rho(u)) \cap CV_A(\rho(v)) = \emptyset$. ◻

We now prove completeness of the NCC-implication check:

Lemma 4.5. *Let $S = (Sol, \emptyset, \Delta, \nabla)$ be a final state of the matching algorithm for input $P = (\Gamma_I, \Delta_I, \nabla_I)$ which passes the LVC check but the NCC-implication check is not valid for S. Then S has no matcher.*

Proof. Assume that S and P are given as in the claim and that the NCC-implication check for S is invalid. By definition of a matcher of state S, Sol can be the only matcher of state S. Soundness of the matching algorithm implies that for all ground substitutions ρ with $\mathtt{Dom}(\rho) = MV_F(P)$, ρ satisfies ∇, $\rho(Sol(s)), \rho(t)$ fulfill the LVC for all $s \trianglelefteq t \in \Gamma_I$, also $\rho(Sol(s)) \sim_{let} \rho(t)$ holds for all $s \trianglelefteq t \in \Gamma_I$. Thus we have to show that there exists such a ρ s.t. for all ground substitutions ρ_0 with $\mathtt{Dom}(\rho_0) = MV_I(\rho(\Delta_3))$ we have $\rho_0(\rho(\Delta))$ is invalid.

Let ρ be the following ground substitution on fixed meta-variables: $\rho(X) = \mathsf{x}_X$, $\rho(\underline{S}) = \lambda \mathsf{x}_S.\mathsf{x}_S$, $\rho(\underline{E}) = \emptyset$ if $\underline{E} \notin \nabla_2$ and $\underline{E} = \mathsf{x}_E.\mathtt{var}\ \mathsf{x}_E$ if $\underline{E} \in \nabla_2$, $\rho(\underline{Ch}) = [\cdot_1].[\cdot_2]$ $\rho(\underline{D}) = [\cdot]$ if $\underline{D} \notin \nabla_1$ and $\rho(D) = \mathsf{d} \neq [\cdot]$ with $CV(\mathsf{d}) = \emptyset$, otherwise (see Definition 2.3), where all variables $\mathsf{x}_X, \mathsf{x}_S, \mathsf{x}_E$ are fresh. By the definition of a LMP ∇ is satisfiable and the LVC holds for all expressions $Sol(s), t$ with $s \trianglelefteq t$ in Γ_I. Thus one can verify that also ρ must satisfy ∇, and the LVC must hold for $\rho(Sol(s)), t$ for all $s \trianglelefteq t \in \Gamma_I$.

Since state S fails the NCC-implication check, there exists $(u, v) \in split_{ncc}(\Delta_3)$ where none of the cases of the NCC-implication check applies.

First assume that u is an instantiable variable. Then $u = v$ and $u = E$ with $E \in \Delta_2$ or $u = X$ must hold. We have $\rho(u) = \rho(v) = u$. Any ground substitution ρ_0 must instantiate E with at least one binding (X with a concrete variable, resp.), i.e. $\rho_0(\rho(u)) = \mathsf{x.s}; env$ ($\rho_0(\rho(u)) = \mathsf{x}$, resp.). But then $\mathsf{x} \in Var_A(\rho_0(\rho(u)))$ and $\mathsf{x} \in CV_A(\rho_0(\rho(u)))$ and thus $\rho_0(\rho(\Delta_3))$ is invalid.

If u is not an instantiable meta-variable, then v cannot be an instantiable meta-variable, since otherwise case (5) of the NCC-implication check would hold.

Thus the remaining cases are that u and v are fixed meta-variables or concrete variables. We consider all possible cases: If $(u, v) = (\mathsf{x}, \mathsf{x})$ then $\rho_0(\rho(\Delta_3))$ is invalid for any ρ_0. For all other cases, we modify the definition of ρ, i.e. we provide a substitution ρ' with $\rho'(U) = \rho(U)$ for all $U \notin \{u, v\}$, and $\rho'(u)$ and $\rho'(v)$ are defined in Table 1, where 'n.a.' means not applicable, since u or v is not a meta-variable, variables x_U occurring in the columns for $\rho'(u).\rho'(v)$ are always fresh and pairwise distinct, $cl(Ch^{Triv}) = Triv$, $cl(Ch^{\mathcal{A}}) = \mathcal{A}$. Note that the context d in the last five rows always exists due to our assumption in Definition 2.3. We have to verify that ρ' still satisfies ∇: This holds for all cases, since $split_{ncc}(\nabla) \cup NCC_{lvc}$ cannot contain (u, v) (due to item (2)), (u, u) or (v, v) (either since both are concrete variables, or due to items (6), (7), (8)), and also either $Var_A(\rho'(v)) \cap CV_A(\rho'(u)) = \emptyset$, or $(v, u) \notin split_{ncc}(\nabla) \cup NCC_{lvc}$ since either (v, u) is impossible (e.g. for $u = \underline{S}$) or due to items (6), (7), (8). Furthermore, one has to verify that ρ' satisfies the LVC for $Sol(s), t$ with $s \trianglelefteq t \in \Gamma_I$ which holds since ρ satisfies the LVC for these expressions and since $(u, v) \notin NCC_{lvc}$ (or $(v, u) \notin NCC_{lvc}$ for specific cases). Finally, we verify that $Var_A(\rho'(u)) \cap CV_A(\rho'(v)) \neq \emptyset$ in all cases, and thus for all ground instantiations ρ_0, the atomic NCC (u, v) is violated for $\rho_0 \circ \rho'$. Hence, $\rho_0(\rho'(\Delta_3))$ is violated for all ρ_0.

Lemma 4.6. *Let $S = (Sol_S, \Gamma_S, \Delta_S, \nabla_S)$ be a state of the matching algorithm and $\frac{S}{S_1 \mid \ldots \mid S_n}$ be a rule in Figs. 3 and 4. If σ is a matcher of state $S_i = (Sol_i, \Gamma_i, \Delta_i, \nabla_i)$, then σ is a matcher of state S.*

Proof. This follows by inspecting all rules, verifying that the rules respect the constraints in Δ w.r.t. the given constraints in ∇, verifying that every instance of Sol_i is also an instance of Sol_S, and verifying that for each instance σ of Sol_i $\sigma(s) \sim_{let} t$ for all $s \trianglelefteq t \in \Gamma_i$ also implies $\sigma(s) \sim_{let} t$ for all $s \trianglelefteq t \in \Gamma_S$.

Now we are able to prove soundness of MatchLRS:

Proposition 4.7. *The matching algorithm is sound, i.e. let P be a LMP and the matching algorithm delivers $S = (Sol, \emptyset, \Delta, \nabla)$ for input P where S passes the failure-tests, then Sol is a matcher of P.*

Proof. Let $P = (\Gamma, (\Delta_{I,1}, \Delta_{I,2}, \Delta_{I,3}), \nabla)$ be a LMP and S_1 be the initial state of the matching algorithm for input P. Let $S_1 \to \ldots \to S_n$ be a derivation of the matching algorithm where $S_n = (Sol_n, \emptyset, (\Delta_{n,1}, \Delta_{n,2}, \Delta_{n,3}), \nabla)$ is an accepted state or $S_n = Fail$. We use induction on n. If $n = 1$, then any matcher of state S_1 is also a matcher of P. If $n > 1$, then consider the last derivation step $S_{n-1} \to S_n$. By the induction hypothesis we have that a matcher for state S_{n-1} is also a matcher for P. If S_n is $Fail$, then soundness holds. If S_n is an accepted state, then Lemma 4.6 shows that a matcher of S_n is also a matcher of S_{n-1} and by the induction hypothesis we thus have that a matcher of S_n is a matcher for P. We finally observe that Sol_n is a matcher which is also ensured by Lemma 4.4, since it shows that Δ_n is satisfiable. Moreover, Δ_n is equal to $Sol_n(\Delta_{I,3})$ and thus the obtained ρ_0 in Lemma 4.4 can also be used to show that $\rho_0(\rho(Sol_n(\Delta_I)))$ is satisfiable.

(u,v)	$\rho'(u)$	$\rho'(v)$	(u,v)	$\rho'(u)$	$\rho'(v)$
$(\mathsf{x}, \underline{Y})$	n.a.	x	$(\underline{S}, \underline{X})$	$\mathtt{var}\ \mathsf{x}_X$	$\mathsf{x}_X,$
$(\mathsf{x}, \underline{D})$	n.a.	$\mathtt{letr}\ \mathsf{x}.\mathtt{var}\ \mathsf{x}\ \mathtt{in}\ [\cdot]$	$(\underline{S}, \underline{E})$	$\mathtt{var}\ \mathsf{x}_S$	$\mathsf{x}_S.\mathsf{x}_S$
$(\mathsf{x}, \underline{Ch})$	n.a.	$\mathsf{x}.\mathtt{var}\ \mathsf{x}$	$(\underline{S}, \underline{Ch})$	$\mathtt{var}\ \mathsf{x}_S$	$[\cdot_1].\mathtt{var}\ \mathsf{x}_X;\mathsf{x}_X.[\cdot_2]$
$(\mathsf{x}, \underline{Ch})$	n.a.	$[\cdot_1].\mathtt{var}\ \mathsf{x};\mathsf{x}.[\cdot_2]$	$(\underline{S}, \underline{D})$	$\mathtt{var}\ \mathsf{x}_S$	$\mathtt{letr}\ \mathsf{x}_S.\mathtt{var}\ \mathsf{x}_S\ \mathtt{in}\ [\cdot]$
$(\underline{Y}, \mathsf{x})$	x	n.a.	$(\underline{E}, \mathsf{x})$	$\mathsf{x}.\mathtt{var}\ \mathsf{x}$	n.a.
$(\underline{X}, \underline{Y})$	x_X	x_X	$(\underline{E}, \underline{X})$	$\mathsf{x}_X.\mathtt{var}\ \mathsf{x}_X$	$\mathsf{x}_X,$
$(\underline{Y}, \underline{D})$	x_Y	$\mathtt{letr}\ \mathsf{x}_Y.\mathtt{var}\ \mathsf{x}_Y\ \mathtt{in}\ [\cdot]$	$(\underline{E_1}, \underline{E_2})$	$\mathsf{x}_{E_1}.\mathtt{var}\ \mathsf{x}_{E_2}$	$\mathsf{x}_{E_2}.\mathtt{var}\ \mathsf{x}_{E_2}$
$(\underline{Y}, \underline{E})$	x_Y	$\mathsf{x}.\mathtt{var}\ \mathsf{x}$	$(\underline{E}, \underline{Ch})$	$\mathsf{x}_E.\mathtt{var}\ \mathsf{x}_{Ch}$	$[\cdot_1].\mathtt{var}\ \mathsf{x}_{Ch};\mathsf{x}_{Ch}.[\cdot_2]$
$(\underline{Y}, \underline{Ch})$	x_Y	$[\cdot_1].\mathtt{var}\ \mathsf{x}_Y;\mathsf{x}_Y.[\cdot_2]$	$(\underline{E}, \underline{D})$	$\mathsf{x}_E.\mathtt{var}\ \mathsf{x}_D$	$\mathtt{letr}\ \mathsf{x}_D.\mathtt{var}\ \mathsf{x}_D\ \mathtt{in}\ [\cdot]$
$(\underline{S}, \mathsf{x})$	$\mathtt{var}\ \mathsf{x}$	n.a.			

(u,v)	$\rho'(u)$	$\rho'(v)$
$(\underline{Ch^{Triv}}, \mathsf{x})$	$[\cdot_1].\mathtt{var}\ \mathsf{x};\mathsf{x}.[\cdot_2]$	n.a.
$(\underline{Ch^{Triv}}, \underline{X})$	$[\cdot_1].\mathtt{var}\ \mathsf{x}_X;\mathsf{x}_X.[\cdot_2]$	$\mathsf{x}_X,$
$(\underline{Ch^{Triv}}, \underline{E})$	$[\cdot_1].\mathtt{var}\ \mathsf{x}_{Ch};\mathsf{x}_{Ch}.[\cdot_2]$	$\mathsf{x}_{Ch}.\mathtt{var}\ \mathsf{x}_{Ch}$
$(\underline{Ch_1^{Triv}}, \underline{Ch_2})$	$[\cdot_1].\mathtt{var}\ \mathsf{x}_{Ch};\mathsf{x}_{Ch}.[\cdot_2]$	$[\cdot_1].\mathtt{var}\ \mathsf{x}_{Ch};\mathsf{x}_{Ch}.[\cdot_2]$
$(\underline{Ch^{Triv}}, \underline{D})$	$[\cdot_1].\mathtt{var}\ \mathsf{x}_{Ch};\mathsf{x}_{Ch}.[\cdot_2]$	$\mathtt{letr}\ \mathsf{x}_{Ch}.\mathtt{var}\ \mathsf{x}_{Ch}\ \mathtt{in}\ [\cdot]$
$(\underline{Ch^{\mathcal{A}}}, \mathsf{x})$	$[\cdot_1].\mathsf{d}[\mathsf{x}_{Ch}];\mathsf{x}_{Ch}.[\cdot_2]$ s.t. $\mathit{Var}(\mathsf{d}) = \{\mathsf{x}\}$	n.a.
$(\underline{Ch^{\mathcal{A}}}, \underline{X})$	$[\cdot_1].\mathsf{d}[\mathsf{x}_{Ch}];\mathsf{x}_{Ch}.[\cdot_2]$ s.t. $\mathit{Var}(\mathsf{d}) = \{\mathsf{x}_X\}$	$\mathsf{x}_X,$
$(\underline{Ch^{\mathcal{A}}}, \underline{E})$	$[\cdot_1].\mathsf{d}[\mathsf{x}_{Ch}];\mathsf{x}_{Ch}.[\cdot_2]$ s.t. $\mathit{Var}(\mathsf{d}) = \{\mathsf{x}_E\},$	$\mathsf{x}_E.\mathtt{var}\ \mathsf{x}_E$
$(\underline{Ch_1^{\mathcal{A}}}, \underline{Ch_2})$	$[\cdot_1].\mathsf{d}[\mathsf{x}_{Ch_1}];\mathsf{x}_{Ch_1}.[\cdot_2]$ s.t. $\mathit{Var}(\mathsf{d}) = \{\mathsf{x}_{Ch_2}\}$	$[\cdot_1].\mathtt{var}\ \mathsf{x}_{Ch_2};\mathsf{x}_{Ch_2}.[\cdot_2]$
$(\underline{Ch^{\mathcal{A}}}, \underline{D})$	$[\cdot_1].\mathsf{d}[\mathsf{x}_{Ch}];\mathsf{x}_{Ch}.[\cdot_2]$ s.t. $\mathit{Var}(\mathsf{d}) = \{\mathsf{x}_D\}$	$\mathtt{letr}\ \mathsf{x}_D.\mathtt{var}\ \mathsf{x}_D\ \mathtt{in}\ [\cdot]$

Table 1. Modifications of ρ depending on (u, v)

For completeness, we inspect the rules, verify that branching covers all cases, and derive:

Lemma 4.8. *Let S be a state of the matching algorithm and $\frac{S}{S_1 \mid \ldots \mid S_n}$ be a rule in Figs. 3 and 4. If σ is a matcher of S, then σ is a matcher of some state S_i.*

Lemma 4.9. *Let $S = (Sol, \emptyset, \Delta, \nabla)$ be a final state of the matching algorithm for input $P = (\Gamma_I, \Delta_I, \nabla_I)$ which passes the LVC check but the NCC-implication check is not valid for S. Then S has no matcher.*

As a further property we require that MatchLRS never does get stuck in a non-final state.

Proposition 4.10. *The matching algorithm does not get stuck.*

Proof. We check that at long as Γ is non-empty, at least one rule is applicable. First consider the case that Γ contains variable equations: Then either (SolveX), (ElX) or failure rule (VarFail) is applicable. If Γ contains binding equations, then rule (DecH) is applicable. If Γ contains an expression equation $s \trianglelefteq t$ then we distinguish the cases for s: If $s = \underline{S}$, then rule (ElS) or failure rule (ExFailS) is applicable. If $s = S$, then rule (SolveS) is applicable. If $s = (f\ s_1 \ldots s_n)$, then rule (DecF) or (ExFailF) is applicable. If $s = x.s'$ then rule (DecH) is applicable. If $s = \mathtt{letr}\ env\ \mathtt{in}\ s'$, then rule (DecL) or (ExFailL) is applicable. If $s = \underline{D}[s']$, then rule (DecD) or (CxFailF) is applicable. If $s = D[s']$, then we distinguish the cases for t: If $t = \underline{D'}[s']$ and $cl(\underline{D'}) \leq cl(D)$ then either rule (CxPx) or (CxGuess) is applicable, and if $cl(\underline{D'}) > cl(D)$ then rule (CxGuess) or (CxCG) is applicable. If $t = f\ s_1 \ldots s_n$ then one of the rules (CxGuess), (CxF), (CxFailI) is applicable. If $t = \underline{S}$ then rule (CxFailI) is applicable. If t is a $\mathtt{letr}$-expression then rule (CxGuess), (CxL), or (CxFailI) is applicable.

Finally, we consider the case that Γ contains an environment equation $env \trianglelefteq env'$. If $env' = \emptyset$, then one of the rules (EnvEm), (EnvAE), or (EFailEm) is applicable. If $env' = \underline{Ch}[z, s]; env'$, then rule (EFailEm) is applicable if env is empty; or rule (EFailCFR) is applicable, if $env \neq E'; env_0$, $env \neq Ch'[z', s']; env_0$ with $cl(Ch') \geq cl(\underline{Ch})$, or $env \neq \underline{Ch}[z', s']; env_0$; or rule (EnvC) or (SolveE) is applicable, if env contains some E', some $Ch'[z', s']$ with $cl(Ch') \geq cl(\underline{Ch})$, or $\underline{Ch}[z', s']$. Now let env' be non-empty s.t. env' does not contain $\underline{Ch}$-variables and $env' = b; env''$. If env is empty then (EFailEm) is applicable. If env contains $\underline{Ch}$-variables then (EFailCFL) is applicable. Now assume env is non-empty and contains no $\underline{Ch}$-variables. If env contains bindings or Ch- or E-variables, then rule (EnvB) or (SolveE) is applicable. Now assume that env contains only $\underline{E}$-variables then (EFailEL) or (ElE) is applicable. and $\underline{E}$ variables then either (EFailCFL), (EFailEL), (ElE), or (EnvC) Now assume that env' is non-empty and that it does not contain bindings and $\underline{Ch}$-variables. Then $env' = \underline{E}_1; \ldots; \underline{E}_m$. If env is empty, then (EFailEm) is applicable. If env contains a binding, then (EFailB) is applicable. If env contains a $\underline{Ch}$-variable then (EFailCFL) is applicable. If env

contains a Ch-variable then (EFailCl) is applicable. The remaining cases are that env contains only E' and $\underline{E''}$ components. If env contains no E'-variables and no component $\underline{E}_i$ then (EFailER) is applicable. If env contains $\underline{E}_i$, then rule (ElE) is applicable. Now assume env contains only E'-variables. If $env = E'$ then (SolveE) is applicable. If $env = E'; env_0$ with $\underline{E}_i \in \Delta_2$ or $E' \notin \Delta_2$ then (EnvE) is applicable. If $env = E'_1; \ldots; E'_k$ and $\underline{E}_i \notin \Delta_2$ for all $i = 1, \ldots, m$ and $E'_i \in \Delta_2$ for all $i = 1, \ldots, k$ then (EnvFailEE) is applicable.

We show termination of MatchLRS:

Proposition 4.11. MatchLRS *always terminates with Fail or with an accepted state.*

Proof. For a state $(Sol, \Gamma, \Delta, \nabla)$, let $\mu = (\mu_1, \mu_2, \mu_3, \mu_4, \mu_5)$ where μ_1 is the number of `letr`-expressions in Γ, μ_2 is the number of bindings in environment equations in Γ (equations $x.s \trianglelefteq x'.s'$ are counted as binding equations, not as environment equations), μ_3 is the number of occurrences of fixed chain variables in Γ, μ_4 is the size of Γ and μ_5 is the number of context variables occurring in Γ that are not in Δ_1. We use the lexicographic ordering on the measure μ and show that each rule application strictly decreases the measure or leads to $Fail$. The rule (CxL) strictly decreases μ_1. The rule (EnvB) does not increase μ_1 and strictly decreases μ_2. The rule (EnvC) does not increase μ_1 and μ_2 but strictly decreases μ_3. All other rules except for the second branch of (CxGuess) do not increase μ_1, μ_2, μ_3 and strictly decrease μ_4. The second branch of (CxGuess) does not increase $\mu_1, \mu_2, \mu_3, \mu_4$ but strictly decreases μ_5. Proposition 4.10 shows that for all non-final states a rule is applicable.

Now completeness of MatchLRS can be proved:

Proposition 4.12. *The matching algorithm* MatchLRS *is complete, i.e. if a LMP $P = (\Gamma, \Delta, \nabla)$ has a matcher σ, then there exists an accepted state $S = (\sigma, \emptyset, \Delta_S, \nabla_S)$ of the matching algorithm for input P.*

Proof. For applications of non-failure rules this follows from Lemma 4.8. For the failure rules this can be verified by inspecting the rules, where Lemma 4.9 shows that invalidity of the NCC-implication check implies that P has no matcher. Proposition 4.11 shows that MatchLRS terminates and does not get stuck.

Theorem 4.13. MatchLRS *is sound and complete.*

Proposition 4.14. *All derivations of the matching algorithm are of polynomial height in the size of the input, and the size of each state is polynomial in the size of the input.*

Proof. We again use the measure $\mu = (\mu_1, \mu_2, \mu_3, \mu_4, \mu_5)$ from the proof of Proposition 4.11. We estimate the number of applications of each derivation rule. First assume that $\mu_0 = (\mu_{0,1}, \mu_{0,2}, \mu_{0,3}, \mu_{0,4}, \mu_{0,5})$ is the measure μ for the initial state. Clearly all components of μ_0 are bounded by the size of the input.

Rule (CxL) strictly decreases μ_1 (in both cases) and all other rules do not increase μ_1. Thus the total number of (CxL)-steps is bounded by μ_1. Each application of (CxL) does not increase μ_2, μ_3, and can increase μ_4 by at most 2, and increase μ_5 by at most 1. Thus in the derivation the measure $(\mu_{0,2}, \mu_{0,3}, \mu_{0,4}, \mu_{0,5})$ plus the increase over all (CxL)-steps is bounded by $(\mu_{0,2}, \mu_{0,3}, 2\mu_{0,1}+\mu_{0,4}, \mu_{0,1}+\mu_{0,5})$.

Now we consider the derivation without counting the (CxL) steps. Rule (EnvB) strictly decreases μ_2 and all other remaining rules do not increase μ_2. Thus there at most μ_2-applications of rule (EnvB). Each application of μ_2 does not increase μ_3, may increase μ_4 by at most 7, and may increase μ_5 by at most 2. Thus adding the increase of all (EnvB) steps to the initial measure for (μ_3, μ_4, μ_5) leads to $(\mu_{0,3}, 7\mu_{0,2} + 2\mu_{0,1} + \mu_{0,4}, 2\mu_{0,2} + \mu_{0,1} + \mu_{0,5})$.

Now we consider the derivation without counting the (CxL) and (EnvB) steps. Rule (EnvC) strictly decreases μ_3 and all other remaining rules do not increase μ_3. Thus there at most $\mu_{0,3}$ applications of (EnvC). Each (EnvC) application may increase μ_4 by at most 5, and may increase μ_5 by at most 2. Thus adding the increase of all (EnvC) steps to the initial measure for (μ_4, μ_5) leads to $(5\mu_{0,3} + 7\mu_{0,2} + 2\mu_{0,1} + \mu_{0,4}, 2\mu_{0,3} + 2\mu_{0,2} + \mu_{0,1} + \mu_{0,5})$. Now we consider the derivation without counting the (CxL), (EnvB), (EnvC) steps. All remaining do not increase μ_4 and except for the second branch of (CxGuess) they strictly decrease μ_4. Thus there at most $(5\mu_{0,3} + 7\mu_{0,2} + 2\mu_{0,1} + \mu_{0,4})$ of those steps. Each of these steps may increase μ_5 by at most 2 (only rules (CxPx), (CxF), (CxL) can increase μ_5). Thus adding this increase leads to $2\mu_{0,4} + 12\mu_{0,3} + 16\mu_{0,2} + 5\mu_{0,1} + \mu_{0,5}$. for μ_5. Since the second branch of (CxGuess) does not increase $\mu_1, \mu_2, \mu_3, \mu_4$ but strictly decreases μ_5 there at most $2\mu_{0,4} + 12\mu_{0,3} + 16\mu_{0,2} + 5\mu_{0,1} + \mu_{0,5}$. applications of the second branch of (CxGuess).

By inspecting all rules, we verify that the size increase of each rule application is constant and thus also the size of each state is of the matching algorithm is polynomially bounded by the size of the input.

Theorem 4.15. *The matching algorithm runs in non-deterministic polynomial time, and the letrec matching problem is NP-complete.*

Proof. All derivations of MatchLRS are of polynomial height in the size of the input (Proposition 4.14). Propositions 4.7 and 4.12 imply that MatchLRS is an NP-decision procedure for the LMP. To show NP-hardness, we reduce the Monotone one-in-three-3-SAT problem to the LMP. Let $C = \{C_1, \ldots, C_n\}$ be an instance of the Monotone one-in-three-3-SAT problem where $C_i = \{S_{i,1}, S_{i,2}, S_{i,3}\}$ and $S_{i,j}$ are propositional variables. For each clause C_i, generate the matching equation $\texttt{letr}\ Y_{i,1}.S_{i,1}; Y_{i,2}.S_{i,2}; Y_{i,3}.S_{i,3}\ \texttt{in}\ (\texttt{var}\ \mathsf{x}_i) \trianglelefteq \texttt{letr}\ \mathsf{y}_{i,1}.\mathsf{x}_f; \mathsf{y}_{i,2}.\mathsf{x}_f; \mathsf{y}_{i,3}.\mathsf{x}_t\ \texttt{in}\ (\texttt{var}\ \mathsf{x}_i)$ where $Y_{i,j}$, $\mathsf{y}_{i,j}$, x_i are fresh for i and $S_{i,j}$ are expression variables corresponding to the propositional variables. The LMP is solvable iff the Monotone one-in-three-3-SAT instance is satisfiable: $S_{i,j}$ is mapped to $\texttt{var}\ \mathsf{x}_t$ ($\texttt{var}\ \mathsf{x}_f$, resp.) iff propositional variable $S_{i,j}$ is `true` (`false`, resp.).

We demonstrate how to use the matching algorithm to perform reductions and transformations on the meta-expressions. Note that the main difference between a compute meta rewrite step defined below and the direct use of the matcher in Proposition 3.8 is the treatment of the additional substitution ρ_0: In a computed meta rewrite step the requirements on ρ_0 are added to the constraint tuple and thus no explicit construction of ρ_0 is necessary.

Definition 4.16. *Let* (s, ∇) *be a constrained expression,* $(\ell \xrightarrow{n}_{\Delta} r)$ *be a meta letrec rewrite rule,* (Sol, Δ', ∇) *be an accepted output of the matching algorithm for input* $(\{\ell \trianglelefteq s\}, \Delta, \nabla)$*, then* $(s, \nabla) \overset{n}{\Rightarrow} (Sol(r), \nabla \cup \Delta')$ *is a* computed meta rewrite step*.*

The properties of MatchLRS and matchers imply:

Theorem 4.17. *Let* $(s, \nabla) \overset{n}{\Rightarrow} (t, \nabla')$ *and* $\rho(s) \in \gamma(s, \nabla)$*. For all ground substitutions* ρ_0 *s.t.* $\rho_0 \circ \rho$ *satisfies* ∇' *and* $\texttt{Dom}(\rho_0) = MV(\rho(t)) \cup MV(\rho(\nabla'))$*, we have* $\rho(s) \xrightarrow{n} \rho_0(\rho(t))$*. Moreover, at least one such substitution* ρ_0 *exists.*

5 Conclusion

We presented an approach to rewrite higher-order meta expressions of the language `LRSX` by introducing the letrec matching problem and developing 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`.

References

1. Zena M. Ariola and Matthias Felleisen. The call-by-need lambda calculus. *J. Funct. Program.*, 7(3):265–301, 1997.
2. Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. In *POPL 1995*, pages 233–246. ACM, 1995.
3. Christophe Calvès and Maribel Fernández. Nominal matching and alpha-equivalence. In *WoLLIC 2008*, volume 5110 of *LNCS*, pages 111–122. Springer, 2008.
4. Christophe Calvès and Maribel Fernández. A polynomial nominal unification algorithm. *Theor. Comput. Sci.*, 403(2-3):285–306, 2008.
5. James Cheney. Toward a general theory of names: Binding and scope. In *MERLIN 2005*, pages 33–40. ACM, 2005.
6. Maribel Fernández and Murdoch Gabbay. Nominal rewriting. *Inf. Comput.*, 205(6):917–965, 2007.
7. Jordi Levy and Mateu Villaret. Nominal unification from a higher-order perspective. In *RTA 2008*, volume 5117 of *Lecture Notes in Computer Science*, pages 246–260. Springer, 2008.
8. Elena Machkasova. Computational soundness of a call by name calculus of recursively-scoped records. In *WRS 2007*, ENTCS, 2007.
9. Elena Machkasova and Franklyn A. Turbak. A calculus for link-time compilation. In *ESOP 2000*, volume 1782, pages 260–274. Springer, 2000.
10. Andrew K. D. Moran, David Sands, and Magnus Carlsson. Erratic fudgets: A semantic theory for an embedded coordination language. *Sci. Comput. Program.*, 46(1-2):99–135, 2003.
11. Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *PLDI 1988*, pages 199–208. ACM, 1988.
12. Andrew Pitts. Nominal techniques. *ACM SIGLOG News*, 3(1):57–72, 2016.

13. Conrad Rau and Manfred Schmidt-Schauß. A unification algorithm to compute overlaps in a call-by-need lambda-calculus with variable-binding chains. In *UNIF 2011*, pages 35–41, 2011.
14. David Sabel and Manfred 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.
15. Manfred Schmidt-Schauß, Temur Kutsia, Jordi Levy, and Mateu Villaret. Nominal unification of higher order expressions with recursive let. In *LOPSTR 2016*, 2016. informal proceedings.
16. Manfred Schmidt-Schauß and David Sabel. Unification of program expressions with recursive bindings. In *PPDP 2016*, pages 160–173. ACM, 2016.
17. Manfred Schmidt-Schauß, David Sabel, and Elena Machkasova. Simulation in the call-by-need lambda-calculus with letrec. In *RTA 2010*, volume 6 of *LIPIcs*, pages 295–310. Schloss Dagstuhl, 2010.
18. Manfred Schmidt-Schauß, Marko Schütz, and David Sabel. Safety of Nöcker's strictness analysis. *J. Funct. Programming*, 18(04):503–551, 2008.
19. Christian Urban, Andrew M. Pitts, and Murdoch Gabbay. Nominal unification. In *CSL 2003*, volume 2803, pages 513–527. Springer, 2003.
20. J. B. Wells, Detlef Plump, and Fairouz Kamareddine. Diagrams for meaning preservation. In *RTA 2003*, volume 2706, pages 88 –106. Springer, 2003.
21. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.