Rewriting with Generalized Nominal Unification

Yunus Kutz and Manfred Schmidt-Schauß

Goethe-University, Frankfurt, Germany

Technical Report Frank-63

Research group for Artificial Intelligence and Software Technology
Institut für Informatik,
Fachbereich Informatik und Mathematik,
Johann Wolfgang Goethe-Universität,
Postfach 11 19 32, D-60054 Frankfurt, Germany

October 23, 2019

Abstract. Abstract. We consider matching, rewriting, critical pairs and the Knuth-Bendix confluence test on rewrite rules in a nominal setting extended by atom-variables. Computing critical pairs is done using nominal unification, and rewriting using nominal matching. We utilise atom-variables to formulate rewrite rules, which is an improvement over previous approaches, using usual nominal unification, nominal matching and nominal equivalence of expressions coupled with a freshness constraint. We determine the complexity of several problems in a quantified freshness logic. In particular we show that nominal matching is Π_2^p-complete. We prove that the adapted Knuth-Bendix confluence test is applicable to a nominal rewrite system with atom-variabes and thus, that there is a decidable test whether confluence of the ground instance of the abstract rewrite system holds. We apply the nominal Knuth Bendix confluence criterion to the theory of monads, and compute a convergent nominal rewrite system modulo alpha-equivalence.

1 Introduction

The goal of this paper is to demonstrate the expressive power of nominal modeling with atom-variables [23] also in applications. Therefore we consider rewriting, matching and critical pairs ala Knuth-Bendix [12] in a higher-order language with alpha-equivalence and nominal modeling, where in the nominal unification and matching algorithm also atom-variables are permitted in addition to expression-variables and where the rewriting is done using a corresponding form of nominal matching with atom-variables. This improves upon the approaches in [9, 2] by modeling equivariance through atom-variables. The application of nominal unification with atom-variables avoids guessing of (dis-)equality of atoms, which would be necessary by the previous approaches of nominal unification in rewriting, where this is also necessary in corresponding rewriting sequences in every single rewriting step.

Nominal techniques [18, 17] support machine-oriented reasoning on the syntactic level for higher-order languages and support reasoning modulo alpha-equivalence. An algorithm for nominal unification was first described in [26], which outputs unique most general unifiers. More efficient algorithms are given in [3, 13], exhibiting a quadratic algorithm. The approach is also used in higher-order logic programming [4, 5] and in automated theorem provers like nominal Isabelle [24, 25]. Nominal unification was generalized to permit also atom-variables [23] where also in the generalization, unique most general unifiers are computed, while the decision problem is NP-complete.

A simple example to motivate the use of atom-variables for nominal modeling is the reduction rule (cpcx) in the concurrent calculus CHF [20, 19] or in other functional programming calculi. It permits rewriting a subexpression using the rule

$$(\texttt{let } y = c\ x_1 \dots x_n \texttt{ in } s) \to (\texttt{let } y = c\ x_1 \dots x_n \texttt{ in } s'),$$

where s' is the expression s where one free occurrence of the variable y is replaced by $(c\ x_1 \dots x_n)$. This rule can be applied as a correct transformation even if the variables x_i are not pairwise different,

which is in contrast to usual nominal rewriting using atoms instead of atom-variables, since a single unifier in our proposed algorithm covers all possibilities of equal/unequal variables.

Our motivation to study nominal rewriting is to improve automated reasoning methods in higher-order programming languages. For example program transformations can often be defined by nominal rewriting rules. The advantageous feature is that a single nominal rewriting step is usually possible in polynomial time and it is unique. The satisfiability check of the introduced constraints is usually in NP, or in the polynomial hierarchy. The contrast is second-order rewriting which is usually undecidable and not unique.

The *results* of this paper are as follows. We define a logic QFL over nominal constraints and equations in Section 3 and determine the complexity of validity of freshness and equivalence formulas which is later used to determine the complexity of matching (Corollary 3.14). We describe a matching algorithm in Section 4 and give a definition of nominal rewriting of expressions under constraints and with atom-variables in Section 4.1.

The complexity of nominal matching with atom-variables due to the complexity of constraint satisfiability is proved to be Π_2^p-complete (Theorem 4.10).

A variant of the Knuth Bendix confluence test under atom-variables is described in Section 4 and proved correct for detecting confluence on the induced rewriting system on ground expressions (Theorem 4.19). We compute the completion and prove a confluence result modulo alpha-equivalence for the (completed) rewrite rules of the monad theory (Theorem 5.4), which is more general than previous ones and also demonstrates the power of our method.

The structure of the paper is as follows. In Section 2 the languages of nominal expressions are described. Section 3 describes the quantified freshness logic for quantified freshness constraints and alpha-equivalence. Section 4 is a presentation and adaptation of rewrite rules and the Knuth-Bendix confluence test. In Section 5 we apply nominal matching, rewriting and nominal confluence test with atom-variables to the theory of monads. In Section 6 we give a comparison to the classical nominal rewriting framework introduced by [7]. We conclude in Section 7.

2 Nominal Terms

We first introduce some notation [23].

Let $\mathcal{F}$ be a set of function symbols $f \in \mathcal{F}$, s.t. each f has a fixed arity $ar(f) \geq 0$. Let $\mathcal{A}t$ be *the set of atoms* ranged over by a, b, c. The ground language NL_a is defined by the grammar:

$$e ::= a \mid (f\ e_1 \dots e_{ar(f)}) \mid \lambda a.e$$

where λ is a binder for atoms. The basic constraint $a\#e$ is valid if a does not occur freely in e and a set of constraints ∇ is valid if all constraints are valid. Constructs of the form $(a\ b)$ will denote a *swapping* of the two atoms a, b in an expression e.

We will use the following definition of α-equivalence on NL_a:

Definition 2.1. *Syntactic α-equivalence $\sim$ in NL_a is inductively defined:*

$$\frac{}{a \sim a} \qquad \frac{\forall i : e_i \sim e_i'}{(f\ e_1 \dots e_{ar(f)}) \sim (f\ e_1' \dots e_{ar(f)}')} \qquad \frac{e \sim e'}{\lambda a.e \sim \lambda a.e'} \qquad \frac{a\#e' \wedge e \sim (a\ b)\cdot e'}{\lambda a.e \sim \lambda b.e'}$$

Note that $\sim$ is identical to the equivalence relation generated by α-equivalence by renaming binders, which can be proved in a simple way by arguing on the (binding-)structure of expressions (using deBruijn-indices) and hence $\sim$ is an equivalence relation on NL_a. It is also a congruence on NL_a, i.e., for any context C, we have $e_1 \sim e_2$ implies $C[e_1] \sim C[e_2]$.

We introduce two further languages, where we also permit permutations and atom- and expression-variables.

Definition 2.2. *Let $\mathcal{S}$ be the set of expression-variables ranged over by S, T and let $\mathcal{A}$ be the set of atom-variables ranged over by A, B. The grammar of the nominal language NL_{aAS} with atoms, atom-variables and expression-variables is:*

$$\begin{array}{l} e \ ::= W \mid \pi{\cdot}S \mid (f\ e_1 \ldots e_{ar(f)}) \mid \lambda W.e \\ \pi \ ::= \emptyset \mid (W\ W') \cdot \pi' \\ W ::= \pi{\cdot}a \mid \pi{\cdot}A \end{array}$$

where π is a permutation and $\emptyset$ denotes the identity.
The language $NL_{AS} \subset NL_{aAS}$ is defined by:

$$\begin{array}{l} e \ ::= V \mid \pi{\cdot}S \mid (f\ e_1 \ldots e_{ar(f)}) \mid \lambda V.e \\ \pi \ ::= \emptyset \mid (V\ V') \cdot \pi' \\ V ::= \pi{\cdot}A \end{array}$$

Note that we permit nested permutation expressions. The expression $((\pi{\cdot}A)\ (\pi'{\cdot}A'))$ is a single nested swapping. The inverse π^{-1} of a permutation $\pi = sw_1{\cdot}\ldots{\cdot}sw_n$ with swappings sw_i is the expression $sw_n{\cdot}\ldots{\cdot}sw_1$. The set $AtVar(e)$ are the atom-variables contained in e, $ExVar(e)$ the expression-variables contained in e and $Var(e) = AtVar(e) \cup ExVar(e)$. Furthermore, $FVar(e)$ denotes the set of free variables in e, i.e. all expression-variables and all atom-variables which are not bound. These notations will also be used for other syntactic objects.

The languages of interest in this paper are NL_a and NL_{AS}. The ground language of NL_{AS} is NL_a, i.e. expressions s of NL_{AS} can be instantiated to ground expressions by ground substitutions that replace atom-variables by atoms and expression-variables by ground expressions. The language NL_{aAS} serves as an intermediate language during the interpretation of NL_{AS} terms.

3 A Quantified Logic of Freshness Constraints

The logic *QFL* is the background logic for the analysis of the formalism in this paper. It is used to make statements such as correctness, completeness and complexity about matching algorithms and equivalence of constrained expressions.

Definition 3.1. *The formulas of* Quantified Freshness Logic *QFL are defined as follows, where e denotes NL_{aAS}-expressions, W atom-variable suspensions as in NL_{aAS}, X denotes an A or S, and logical operations work as usual.*

$$\begin{array}{l} \Phi := \ Q_1X_1 \ldots Q_kX_k.\varphi \qquad \text{where } Q_i \in \{\forall, \exists\}, \\ \varphi := \ W_i\#e \mid e \sim e \mid \top \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \end{array}$$

Since we use NL_{aAS} there are permutations permitted in the language of QFL. We assume that the simplification and application of ground permutations is done if possible. Thus we can assume that ground expressions e are in NL_a.

Validity of closed formulas is defined as follows:

$\exists A\Phi$ *iff* $\Phi[a/A]$ *for some* $a \in \mathcal{A}t$;	$\neg\varphi$ *iff* φ *is not valid;*
$\exists X\Phi$ *iff* $\Phi[e/X]$ *for some* $e \in NL_a$;	$\varphi_1 \vee \varphi_2$ *iff* φ_1 *or* φ_2;
$\forall A\Phi$ *iff* $\Phi[a/A]$ *for all* $a \in \mathcal{A}t$;	$\varphi_1 \wedge \varphi_2$ *iff* φ_1 *and* φ_2;
$\forall X\Phi$ *iff* $\Phi[e/X]$ *for all* $e \in NL_a$;	$a\#e$ *iff* $a\#e$ *holds in* NL_a;
$\top$ *is always true;*	$e_1 \sim e_2$ *iff* $e_1 \sim e_2$ *holds in* NL_a

For simplicity we will sometimes use the notation $V =_{\#} V'$ instead of $V\#\lambda V'.V$, since the constraint is valid if and only if V and V' are mapped to the same atom.

Note that unlike in first order logic, *QFL* has an implicit fixed domain, NL_a. Note also that equality in NL_a is α-equivalence implying that relations on NL_a cannot distinguish between α-equivalent terms.

Similar to the first order we define a variant of equality where its semantics is α-equivalence, which is necessary, since α-equality of non-ground expressions is not as straightforward as equality in the first-order case.

A set of of quantifier-free formulas of the form $W\#e$ is called a *freshness environment* or *freshness constraint*. These sets are interpreted as conjunctions of their elements. Furthermore, for a free variable X (either an expression-variable or an atom-variable) in a formula φ, we say that an expression e or an atom a is an interpretation of X if we consider the formulas $\varphi[e/X]$ or $\varphi[a/X]$ in the context at hand. The semantics of a formula $\exists X.\Phi$ is then equivalent to there being an interpretation of X s.t. Φ holds and conversely; $\forall X.\Phi$ holds if for all interpretations of X the formula Φ holds.

Definition 3.2. *Let φ be a quantifier-free formula. If there is a ground substitution γ into NL_a, such that $\varphi\gamma$ is closed and valid, then we say γ is* solution *of φ, and also φ is* satisfiable.

The final satisfiability check of unification [23] for example can be written as existential formula $\exists\overline{X}.\nabla$, for a computed unifier (∇, σ).

Definition 3.3. *Let Δ_1, Δ_2 be two freshness environments s.t. $\Delta_1 \subseteq \Delta_2$ and let $\overline{X_1} = Var(\Delta_1), \overline{X_2} = Var(\Delta_2) \setminus \overline{X_1}$. Then a QFL-formula of the form :*

$$\forall\overline{X_1}\exists\overline{X_2}.(\Delta_1 \implies \Delta_2)$$

is called a continuation check *of Δ_1 to Δ_2. If the formula holds we say Δ_1 can be* extended *to Δ_2.*

Corollary 3.4. *Let Δ_1, Δ_2 be two freshness environments. Then Δ_1 can be extended to Δ_2 iff every solution of Δ_1 can be extended to a solution of Δ_2.*

3.1 Decision Procedures and Complexity Analysis

We provide decision algorithms for special forms and quantification of equality $e_1 \sim e_2$, under constraints and for freshness extensions.

For equality the idea is as follows: We transform equations (at positive occurrences in the formulas) into a freshness constraint which is equivalent to the equation, i.e. $\forall\overline{X}.(\Delta \implies e_1 \sim e_2)$ iff $\forall\overline{X}.(\Delta \implies \nabla)$, and then use results for equality-free constraints.

Definition 3.5. *For two NL_{aAS}-expressions e_1, e_2 the algorithm* `EqToCons` *starts with the pair $(\{e_1 \sim e_2\}, \emptyset)$ and applies the rules in Figure 1. If it terminates with $\Gamma = \emptyset$, then the output is ∇.*

(M1) $\dfrac{(\Gamma \cup \{e \sim e\}, \nabla)}{(\Gamma, \nabla)}$ (M2) $\dfrac{(\Gamma \cup \{\pi_1{\cdot}S \sim \pi_2{\cdot}S\}, \nabla)}{(\Gamma, \nabla \cup \{A\#\lambda\pi_1^{-1}\pi_2 A.S \mid A \in AtVar(\pi_1, \pi_2)\})}$

(M3) $\dfrac{(\Gamma \cup \{\pi_1{\cdot}A \sim \pi_2{\cdot}B\}, \nabla)}{(\Gamma, \nabla \cup \{\pi_1{\cdot}A =_\# \pi_2{\cdot}B\})}$ (M4) $\dfrac{(\Gamma \cup \{(f\ e_1 \dots e_{ar(f)}) \sim (f\ e'_1 \dots e'_{ar(f)})\}, \nabla)}{(\Gamma \cup \{e_1 \sim e'_1, \dots, e_{ar(f)} \sim e'_{ar(f)}\}, \nabla)}$

(M5) $\dfrac{(\Gamma \cup \{\lambda\pi_1{\cdot}A_1.e_1 \sim \lambda\pi_2{\cdot}A_2.e_2\}, \nabla)}{(\Gamma \cup \{e_1 \sim ((\pi_1{\cdot}A_1)\ (\pi_2{\cdot}A_2)){\cdot}e_2\}, \nabla \cup \{\pi_1{\cdot}A_1\#\lambda\pi_2{\cdot}A_2.e_2\})}$

(M6) $\dfrac{(\Gamma \cup \{e_1 \sim e_2\}, \nabla)}{(\emptyset, \{\neg\top\} \cup \nabla)}$ If the case of $e_1 \sim e_2$ is not covered by the other rules.

Fig. 1. Rules of `EqToCons`

Lemma 3.6. *The algorithm* `EqToCons` *takes an equation $e_1 \sim e_2$ and a freshness environment Δ as input. If Δ is satisfiable, then it produces a freshness environment ∇ in polynomial time s.t. $\forall\overline{X}.(\Delta \implies e_1 \sim e_2)$ iff $\forall\overline{X}.(\Delta \implies \nabla)$ and only fails if $\forall\overline{X}.(\Delta \implies e_1 \sim e_2)$ is false.*

Proof. It is easy to verify that the rules (M1) ... (M5) retain exactly the set of solutions. The rule (M6) fires, if the top symbols of the expressions e_1, e_2 are different. If one or two are variables or suspensions, then only the (M2) and the (M3)-cases can occur in case there is a solution. Other possibilities prevent solutions, for example $X_1 \sim X_2$ for $X_1 \neq X_2$ is not solvable, since both are universally quantified, and there are sufficiently many different NL_a-expressions, in particular infinitely many different atoms. Note also that (M2) is correct, since every atom A occurring in the permutations π_1, π_2 contributes the constraint $A\#\lambda\pi_1^{-1}\pi_2 A.S$, which informally means: If A is changed by $\pi_1^{-1}\pi_2$, then A is fresh for S. More detailed arguments can be derived from [23], since the rules assure equivalence on ground substitutions.

The complexity follows from the following observations: First note, that every rule strictly decreases the size of Γ by at least 1. So there is at most a linear number of rule applications before termination. The only rule which can increase the size of the data structure, and in particular the size of permutations is (M5). This increase of the total size is bounded by 1 in depth and 1 in the number of swappings per rule application, which bound the maximum permutation-size polynomially in the input size. It follows that every rule runs in polynomial time in the input size, yielding a polynomial time algorithm. In particular, the rule (M2) contributes only polynomial time, because the maximal size of any permutations and the number of atoms is polynomially bounded in the input size.

Now we define the algorithm that decides *QFL*-formulas containing $\sim$-literals

Definition 3.7. *The algorithm* `CheckEq` *decides QFL*($\sim$) *formulas of the form:*

$$\forall \overline{X} : (\Delta \implies e_1 \sim e_2)$$

as follows:

(1) Apply `EqToCons` *to* $e_1 \sim e_2$ *resulting in* ∇.
(2) Check the formula $\forall \overline{X} : (\Delta \implies \nabla)$ *for validity*

Proposition 3.8. *There is a polynomial time transformation from QFL-formulas without equations* $e_1 \sim e_2$ *to QBL maintaining validity such that the quantifier schema in the form* $\forall^*\exists^* \ldots$ *or* $\exists^*\forall^* \ldots$ *remains the same and the complete quantifier prefix increases only quadratically.*

Proof. Let $\varphi = Q_1\overline{X_1} \ldots Q_k\overline{X_k}.\varphi'$ be a *QFL*-formula where m is the number of atom-variables and l the number of expression-variables. To decide the validity of φ it is sufficient to take $NL_{a,m}$, which is NL_a, but there are only m atoms $M = \{a_1, \ldots, a_m\}$. The justification is that there are only m atom-variables, and no atoms in the formula; hence m atoms are sufficient for the interpretation of atom-variables. Every permutation π will be interpreted as permutation over M, and suspensions $\pi{\cdot}A$ will also be interpreted by some atom in M. This holds for all interpretations. Without loss of generality we can assume that for every interpretation the same set of atoms is chosen. An NL_a-interpretation of nominal terms e may contain more atoms, but only the atoms from M that occur in e are relevant for the truth-value of single freshness constraint. Hence it is sufficient to interpret expression-variables S as a subset of M and then compute the freshness constraints. Now we can use this interpretation to argue on the complexity. Every interpretation of φ is of at most quadratic size in m. In addition, the check of validity of φ' under the interpretation can be done in polynomial time. Hence the complexity of the set of freshness formulas with a fixed quantifier prefix $\forall^*\exists^* \ldots$ is in the corresponding complexity class for QBF-formulas with the same quantifier prefix.

The number of necessary variables corresponds to the size of the interpretation of variables as bit-strings, which is $\leq m$, hence a corresponding QBF formula has at most a quadratic number of quantifiers.

Proposition 3.8 implies the following theorem.

Theorem 3.9. *QFL-formulas without equations* $e_1 \sim e_2$ *are in the complexity class in the polynomial hierarchy as indicated by the quantifier prefix interpreted as QBF quantifier prefix.*

Lemma 3.6 and Theorem 3.9 imply:

Corollary 3.10. *The validity check of QFL-formulas* $\forall\overline{X} : (\Delta \implies e_1 \sim e_2)$ *and* $\forall\overline{V} : (\Delta \implies \nabla)$ *is in* coNP. *The validity check for formulas of the form* $\forall\overline{X_1}\exists\overline{X_2} : (\Delta \implies \nabla)$ *is in* Π_2^p.

3.2 Hardness of the Validity Check

To demonstrate the hardness of *QFL*-formulas we encode quantified Boolean formulas, in fact quantified 3-CNF, into *QFL* formulas. This is sufficient to show hardness results [14].

The following constructions are used in our proofs below:
There are two atom-variables – `True` and `False`. For a Boolean formula with the variables $\{x_1, \ldots, x_n\}$ the set $\{A_1, \ldots, A_n\}$ defines the respective atom-variables and the set of (different) atom-variables $\{\overline{A_1}, \ldots, \overline{A_n}\}$ their respective negation. Therefore, we define the following constraints:

(1) $\Delta_1 = \{A_i \# \lambda\texttt{True}.\lambda\texttt{False}.A_i \mid 1 \leq i \leq n\}$
(2) $\Delta_2 = \{A_i \# \overline{A_i}\} \cup \{\overline{A_i} \# \lambda\texttt{True}.\lambda\texttt{False}.\overline{A_i} \mid 1 \leq i \leq n\}$

The constraints Δ_1, Δ_2 ensure that the atom-variables behave like Boolean variables.

Now one needs to encode any given 3-CNF formula. For every clause $l_1 \vee l_2 \vee l_3$ the literal l_i is A_j or $\overline{A_j}$ for some j. Let L_i be either A_j or $\overline{A_j}$ depending on l_i. The constraint $\texttt{True}\#\lambda L_1.\lambda L_2.\lambda L_3.\texttt{True}$ encodes that the clause must be true. For a clause set $\{C_1, \ldots, C_k\}$ define

$$\nabla = \{\texttt{True}\#\lambda L_1^i.\lambda L_2^i.\lambda L_3^i.\texttt{True} \mid 1 \leq i \leq k\}$$

with the construction described as above.

Theorem 3.11. *Decidability of validity of QFL-formulas of the form* $\forall\overline{X}.(\Delta \implies \nabla)$ *and* $\forall\overline{X}.(\Delta \implies e_1 \sim e_2)$ *is* coNP*-hard.*

Proof. Let φ be a universally quantified 3-CNF formula with variables $\{x_1, \ldots, x_n\}$.
Let $\{A_1, \ldots, A_n\}, \{\overline{A_1}, \ldots, \overline{A_n}\}, \Delta_1, \Delta_2, \nabla$ be constructed as described above. Then φ is valid iff

$$\forall\texttt{True}, \texttt{False}, A_1, \ldots, A_n\overline{A_1}, \ldots, \overline{A_n}.(\{\texttt{True}\#\texttt{False}\} \cup \Delta_1 \cup \Delta_2 \implies \nabla) \text{ is valid.}$$

is valid. Hence the class of formulas in the first claim is coNP-hard [14].

For the second claim, we construct e_1, e_2 that reduce to ∇ using `EqToCons`, and hence are equivalent to ∇

Let $\texttt{True}\#\lambda L_1^i.\lambda L_2^i.\lambda L_3^i.\texttt{True}$ be any constraint in ∇. Let $e_1^i = \lambda\texttt{True}.(\texttt{True}\ L_1^i) \cdot (\lambda L_2^i.\lambda L_3^i.\texttt{True})$ and $e_2^i = \lambda L_1^i.\lambda L_2^i.\lambda L_3^i.\texttt{True}$. Then `EqToCons` reduces $e_1^i \sim e_2^i$ to the required constraint. Furthermore $f\ e_1^1\ \ldots e_1^k \sim f\ e_2^1\ \ldots e_2^k$ reduces to ∇. Hence the second claim of the theorem holds.

Corollary 3.12. *Decidability of validity of QFL-formulas of the form* $\forall\overline{X}.(\Delta \implies \nabla)$ *and* $\forall\overline{X}.(\Delta \implies e_1 \sim e_2)$ *is* coNP*-complete.*

Theorem 3.13. *Decidability of validity of QFL-formulas of the form* $\forall\overline{X_1}\exists\overline{X_2} : (\Delta \implies \nabla)$ *is* Π_2^p*-hard.*

Proof. Let $\varphi = \forall x_1, \ldots, x_k \exists x_{k+1} \ldots x_n.\varphi'$ be a QBF with φ' being a 3-CNF.
Let $\{A_1, \ldots, A_n\}, \{\overline{A_1}, \ldots, \overline{A_n}\}, \Delta_1, \Delta_2, \nabla$ be constructed as described above. Then φ is valid iff

$$\forall\texttt{True}, \texttt{False}, A_1, \ldots, A_k \exists A_{k+1}, \ldots, A_n, \overline{A_1}, \ldots, \overline{A_n}.(\{\texttt{True}\#\texttt{False}\} \implies \Delta_1 \cup \Delta_2 \cup \nabla)$$

is valid. Hence Π_2^p-hardness follows [14].

Corollary 3.14. *Decidability of validity of QFL-formulas of the form* $\forall\overline{X_1}\exists\overline{X_2} : (\Delta \implies \nabla)$ *is* Π_2^p*-complete.*

4 Nominal Rewriting

In this section we define the operations of rewriting, matching and unification for nominal expressions. In order to reason about terms in NL_a on a meta level we define this on pairs (Δ, e), called constrained expressions. The advantage is that it leads to a decidable criterion for confluence on the ground level. As a slight disadvantage, it complicates the algorithms and reasoning. For example, the notion of equivalence of constrained expressions would permit several variants. We will use a variant that exactly supports the joining of critical pairs.

In the following we develop and explain the nominal matching, nominal unification, nominal rewrite and nominal Knuth Bendix confluence check.

4.1 Nominal Rewriting, Unification and Matching

We start by defining expressions under constraints, which will be the targets to be rewritten.

Definition 4.1. *Let ∇ be a freshness constraint and e be an expression in NL_{AS}. Then the pair (∇, e) is called a* constrained expression. *The semantics $Sem(\nabla, e)$ is defined as the set $\{e\rho \mid e\rho \in NL_a$ and $\nabla\rho$ is ground and valid$\}$.*

Now we define rewrite rules, which are used in two ways: to rewrite constrained NL_{AS}-expressions, as well as (unconstrained) NL_a-expressions. First we define ground rewriting, and after some preparations we define general rewriting in Definition 4.14.

Definition 4.2. *A* (nominal) rewrite rule *is of the form $R = (\nabla \vdash l \to r)$, where ∇ is a freshness context and l, r are expressions of NL_{AS}, and $FVar(r) \subseteq FVar(l)$. Let $\mathcal{R} = \{R_1, \ldots, R_n\}$ be a* rewrite system *consisting of a set of rewrite rules.*

The induced (semantical) rewrite relation on NL_a is defined as:

$$\xrightarrow{\mathcal{R},NL_a} = \{(C[l_i\gamma], C[r_i\gamma]) \mid (\nabla_i \vdash l_i \to r_i) \in \mathcal{R}, C \text{ is any } NL_a\text{-context}, \gamma \text{ is a ground substitution}, dom(\gamma) = Var(R_i), \nabla_i\gamma \text{ is valid}\}.$$

The equational theory $=_{\mathcal{R},NL_a}$ on NL_a generated by $\mathcal{R}$ is the equivalence and contextual closure of $\xrightarrow{\mathcal{R},NL_a}$.

A rule that fits the definition would be a version of the η-expansion rule: $\{B\#A\} \vdash A \to \lambda B.A\ B$, since the variable B is bound by the lambda on the right hand side. We permit also rules with a non-deterministic behavior. For example, non-determinism is generated by the rule $\{(A\#\lambda B.\lambda C.A\},\ A \to \lambda B.\lambda C.A)$, which permits a rewrite to two different expressions that are not α-equivalent.

Another effect shows up in the rule $\{(A\#\lambda B.\lambda C.A, A\#\lambda C.\lambda D.A\},\ A \to \lambda B.C)$, which is not valid according to our Definition 4.2, since a free atom is introduced. However, the rule is equivalent to $\{(A\#\lambda B.\lambda C.A, A\#\lambda C.\lambda D.A\},\ A \to \lambda B.A)$, which is permitted.

Definition 4.3. *A binary relation Q on NL_a is called* equivariant, *if $(s_1, s_2) \in Q$ iff $\pi.(s_1, s_2) \in Q$ for any atom-permutation π.*

Proposition 4.4. *For any rewrite system $\mathcal{R}$, the rewrite relation $\xrightarrow{R,NL_a}$ as well as the equational theory $=_{\mathcal{R},NL_a}$ are equivariant.*

Proof. This simply holds, since the atom names are not mentioned in $\mathcal{R}$ and any ground substitution can be used.

As a corollary, we obtain, for example, that $=_{\mathcal{R},NL_a}$ either makes all atoms equal, or makes all atoms different.

Unification is classically defined as an algorithm to make terms equal via a substitution, or in the nominal case a substitution and freshness environment [27, 13, 23].

Definition 4.5. *Let $P = (\Delta, \Gamma)$ be a unification problem consisting of a freshness constraint and a set Γ of equations $s_i \doteq t_i$ between NL_{AS}-expressions. Then (∇, σ) is a* nominal unifier *if for every ρ such that $P\sigma\rho$ is ground and $\nabla\rho$ is valid, $s_i\sigma\rho \sim t_i\sigma\rho$ holds for all equations in Γ and $\Delta\sigma\rho$ holds. Furthermore, (∇, σ) is a* most general unifier *if for all ground solutions ρ of P there is a ground substitution γ s.t. $\nabla\sigma\gamma$ is valid and $(\sigma \circ \gamma)(X) \sim \rho(X)$ for all $X \in Var(P)$.*

A matcher of a matching problem $l \preceq r$ is usually defined as a unifier which does not instantiate (or further restrict) the right hand side by applying a substitution to it. Again we need to slightly adapt the previous definitions.

Definition 4.6. *[Matching] Let $(\Delta_1, s), (\Delta_2, t)$ be two constrained expressions that are variable disjoint, i.e. $V_1 \cap V_2 = \emptyset$ for $V_1 = Var(\Delta_1, s)$ and $V_2 = Var(\Delta_2, t)$. A* matcher *of the matching problem $(\Delta_1, s) \preceq (\Delta_2, t)$ is a pair (∇, θ) of a constraint and a substitution as follows:*

- *The right hand side is not restricted, i.e. $dom(\theta) \subseteq V_1$ and every solution of Δ_2 can be extended to a solution of $\nabla \cup \Delta_1\theta$. This corresponds to the formula*

$$\forall V_2 \exists V_1 : (\Delta_2 \implies \nabla \cup \Delta_1\theta)$$

- *It is a unifier of $(\Delta_1 \cup \Delta_2, \{s \doteq t\})$*

We call the tuple a most general matcher *if it is a matcher and a most general unifier of $(\Delta_1 \cup \Delta_2, \{s \doteq t\})$*

When matching is used for rewriting, the disjoint variable condition can be easily fulfilled by completely renaming the rewrite rule.

We provide an algorithm which computes a most general matcher based on the most general unifier algorithm of [23].

Definition 4.7 (Matching algorithm). *The input of the algorithm NomMatchAS is a matching-problem $(\Delta_1, s) \preceq (\Delta_2, t)$ with $V_1 = Var(\Delta_1, s)$, $V_2 = Var(\Delta_2, t)$, where $V_1 \cap V_2 = \emptyset$. The matching algorithm operates on a triple consisting of: a set of equations, a freshness environment, and a substitution. The matching algorithm starts with $(\{s \preceq t\}, \emptyset, \emptyset)$.*
In its first phase it performs the rules in Fig. 2 until the triple is of the form $(\emptyset, \nabla, \theta,)$, i.e. Γ is empty. If the process gets stuck, then there is no match.
Afterwards the second matching condition needs to be checked, i.e. the validity of the formula

$$\forall V_2 \exists V_1 : (\Delta_2 \implies \nabla \cup \Delta_1\theta)$$

must hold.

The output of the algorithm is $(\Delta_2 \cup \nabla \cup \Delta_1\theta, \theta)$.

Example 4.8. These examples help to understand the meaning and effects of the definition and algorithm of nominal matching.

- $(\emptyset, f(S, S)) \preceq (\emptyset, f(S_1, S_2))$. This problem is not solvable, since S_1 and S_2 cannot be made equal.
- $(\emptyset, f(A, A)) \preceq (\emptyset, f(A_1, A_2))$. In this case the algorithm computes $\nabla = (\{A =_\# A_1, A =_\# A_2\}, \theta = \emptyset$. The final check $\forall A_1, A_2 \exists A : (\emptyset \implies \nabla)$ fails. Thus, the problem is not matchable.

$$\text{(M1)}\ \frac{(\Gamma \uplus \{e \preceq e\}, \nabla, \theta)}{(\Gamma, \nabla, \theta)} \qquad \text{(M2)}\ \frac{(\Gamma \uplus \{\pi \cdot S \preceq e\}, \nabla, \theta), S \in V_1}{(\Gamma[(\pi^{-1} \cdot e)/S], \nabla[(\pi^{-1} \cdot e)/S], \theta \cup \{S \mapsto (\pi^{-1} \cdot e)\})}$$

$$\text{(M3)}\ \frac{(\Gamma \uplus \{\pi_1 \cdot S \preceq \pi_2 \cdot S\}, \nabla), S \in V_2}{(\Gamma, \nabla \cup \{A \# \lambda\pi_1^{-1}\pi_2 A.S \mid A \in AtVar(\pi_1, \pi_2)\})} \qquad \text{(M4)}\ \frac{(\Gamma \uplus \{\pi_1 \cdot A \preceq \pi_2 \cdot B\}, \nabla, \theta)}{(\Gamma, \nabla \cup \{A =_\# \pi_1^{-1} \cdot \pi_2 \cdot B\}, \theta)}$$

$$\text{(M4)}\ \frac{(\Gamma \uplus \{(f\ e_1 \dots e_{ar(f)}) \preceq (f\ e'_1 \dots e'_{ar(f)})\}, \nabla, \theta)}{(\Gamma \uplus \{e_1 \preceq e'_1, \dots, e_{ar(f)} \preceq e'_{ar(f)}\}, \nabla, \theta)}$$

$$\text{(M5)}\ \frac{(\Gamma \uplus \{\lambda\pi_1 \cdot A_1.e_1 \preceq \lambda\pi_2 \cdot A_2.e_2\}, \nabla, \theta)}{(\Gamma \uplus \{((\pi_1 \cdot A_1)\ (\pi_2 \cdot A_2)) \cdot e_1 \preceq e_2\}, \nabla \cup \{(A_1 \# \pi_1^{-1} \cdot (\lambda\pi_2 \cdot A_2.e_2))\}, \theta)}$$

Fig. 2. Rules of *NomMatchAS*

- $(\{A \# B\}, f(A, B)) \preceq (\emptyset, f(A_1, A_2))$. In this case the algorithm computes $\nabla = (\{A =_\# A_1, B =_\# A_2\}, \theta = \emptyset$. The final check $\forall A_1, A_2 \exists A, B : (\emptyset \implies \{A =_\# A_1, B =_\# A_2, A \# B\})$ fails, since A_1 and A_2 can be chosen to be equal. Again, the problem is not matchable.
- $(\emptyset, f(A, A)) \preceq (\{A_1 \# \lambda A_2.A_1\}, f(A_1, A_2))$ is solvable, since only instances are valid, where A_1, A_2 are instantiated by the same atom.

Theorem 4.9. *NomMatchAS is sound and complete and computes a most general matcher if there is some matcher.*

Proof. Soundness: Soundness of the rules follows from [23]. If the final test succeeds it is a most general matcher, since the rules produce most general unifiers.

For completeness, we need to show, that an output is produced if a matcher exist.

Let $(\Delta_1, s) \preceq (\Delta_2, t)$ be a matching-problem which has a matcher (∇, σ) and let $V_1 = Var(\Delta_1, s)$, $V_2 = Var(\Delta_2, t)$. More concretely this means that

$$\forall V_2 \exists V_1 : (\Delta_2 \implies \Delta_1\sigma \cup \nabla)$$

holds and (∇, σ) is a unifier of $(\Delta_1 \cup \Delta_2, s \doteq t)$.

If the first phase failed, there could not be a matcher. So we can safely assume that it produces some $(\tilde{\nabla}, \sigma')$, which is by construction a most general unifier of $(\emptyset, s \doteq t)$. The would-be output of the algorithm, $(\Delta_2 \cup \Delta_1\sigma \cup \tilde{\nabla}, \sigma')$, is then by construction a most general unifier of $(\Delta_1 \cup \Delta_2, s \doteq t)$. Let $\nabla' = \Delta_2 \cup \Delta_1\sigma \cup \tilde{\nabla}$.

One still need to show, that this output satisfies the matching formula:

$$\forall V_2 \exists V_1 : (\Delta_2 \implies \nabla')$$

Let γ be any ground substitution, s.t. $dom(\gamma) = V_2$ and $\Delta_2\gamma$ is valid. Let ρ by some ground substitution, s.t. $dom(\rho) = V_1$ and $(\Delta_1\sigma \cup \nabla)\gamma\rho$ is valid. The conditions are equivalent to $\Delta_2(\sigma \circ \gamma \circ \rho)$, $\Delta_1(\sigma \circ \gamma \circ \rho)$ valid and $\nabla(\sigma \circ \gamma \circ \rho)$ is valid.

Because (∇, σ) was a unifier of $(\Delta_1 \cup \Delta_2, s \doteq t)$, $s(\sigma \circ \gamma \circ \rho) \sim t(\sigma \circ \gamma \circ \rho)$ must also hold, which in turn implies that $(\sigma \circ \gamma \circ \rho)$ is a solution of the unification problem.

Because (∇', σ') was a most general unifier, there must be ρ' s.t.

(1) For all $X \in V_1, V_2$: $(\sigma' \circ \rho')(X) \sim \sigma \circ \gamma \circ \rho(X)$.
(2) $\nabla'(\sigma' \circ \rho')$ is valid.

Because γ was chosen as an arbitrary interpretation of V_2, which satisfied Δ_2 and $(\sigma' \circ \rho')$ differs from γ on V_2 at most by α-equivalence the formula:

$$\forall V_2 \exists V_1 : (\Delta_2 \implies \nabla')$$

holds as well.

Theorem 4.10. *Matching is Π_2^p-complete.*

Proof. We can encode an equivalent problem to the formula in the proof of Theorem 3.13 as a matching problem. To that end, let $\Delta_1' = (\Delta_1 \cup \Delta_2 \cup \nabla)[\texttt{True}'/\texttt{True}]$ be a freshness environment with the same structure as in the proof of Theorem 3.13 but with a new atom-variable $\texttt{True}'$. Then $(\Delta_1', \texttt{True}') \preceq (\{\texttt{True}\#\texttt{False}\}, \texttt{True})$ is solvable iff the formula in Theorem 3.13 is solvable. This implies Π_2^p-hardness. The problem is in Π_2^p because *NomMatchAS* runs in two phases which are both in Π_2^p.

4.2 Equivalence of Constrained Expressions

We define equivalence of two constrained expressions with the motivation to use it in a Knuth-Bendix confluence test for the join of critical pairs.

Definition 4.11. *Let (Δ_1, e_1), (Δ_2, e_2) be two constrained expressions, let V be a set of variables, let $V_i = Var(\Delta_i, e_i) \setminus V$ and $V_1 \cap V_2 = \emptyset$.*
Then $(\Delta_1, e_1) \equiv_V (\Delta_2, e_2)$ iff the following holds:

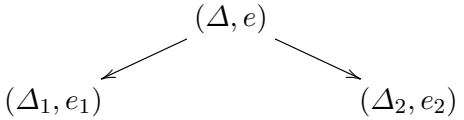
(1) $\forall V \exists V_1, V_2 : (\Delta_1 \iff \Delta_2)$
(2) $\forall V \forall V_1, V_2 : (\Delta_1 \cup \Delta_2 \implies e_1 \sim e_2)$

Example 4.12. We illustrate the $\equiv_V$-definition for several examples:

- $(A\#B, \lambda A.B\ A)$ and $(A'\#B, \lambda A'.B\ A')$ are equal w.r.t. $\equiv_{\{B\}}$:
  $\forall B.\exists A', A : (A\#B \iff A'\#B)$ holds.
  Also $\forall B.\forall A, A' : (A\#B; A'\#B, \implies \lambda A.B\ A \sim \lambda A'.B\ A')$ holds.
  For $V = \{A, B\}$, or $V = \{A', B\}$, this also holds but not for $V = \{A, A', B\}$.
- As another example consider $(A\#\lambda B.A, (A, B))$ and $(A\#\lambda C.A, (A, C))$ with $V = \{A\}$. Then $\forall A \exists B, C : (A\#\lambda B.A \iff A\#\lambda C.A)$ holds, and
  $\forall A, B, C : (A\#\lambda B.A, A\#\lambda C.A \implies (A, C) \sim (A, B)$ is valid.

The relation $\equiv_V$ should not be seen as some equivalence relation – especially since it is in general not transitive. It simply provides a criterion for determining that a confluence diagram is really closed. More specifically, every time a forking occurs in a diagram, one can choose the current variables as V and use $\equiv_V$ to check whether two constrained expressions below do in fact refer to the same related expressions. We write this more formally as a lemma:

Lemma 4.13. *Let (Δ, e), (Δ_1, e_1), (Δ_2, e_2) be constrained expressions, let $V = Var(\Delta, e)$, $V_i = Var(\Delta_i, e_i) \setminus V$ and let $\to$ be any relation on constrained expressions. Suppose*



holds, with $(\Delta_1, e_1) \equiv_V (\Delta_2, e_2)$. Let γ be any ground substitution with $dom(\gamma) = V$ s.t. $\Delta\gamma$ is valid and let ρ_i be any ground substitution with $dom(\rho_i) = V_i$ s.t. $\Delta_i\gamma\rho_i$ holds. Then $e_1\gamma\rho_1 \sim e_2\gamma\rho_2$. Furthermore, such a ρ_1 exists iff such a ρ_2 exists.

Proof. The first claim follows directly from criterion 2 and the second one from criterion 1 of Definition 4.11.

Another way to think of Lemma 4.13 is to say, that if $(\Delta, e) \to (\Delta_1, e_1)$, $(\Delta, e) \to (\Delta_2, e_2)$ and $(\Delta_1, e_1) \equiv_V (\Delta_2, e_2)$ then the corresponding induced relations on NL_a are identical modulo α-equivalence.

4.3 Nominal Rewriting

We define nominal rewriting on NL_{AS} on constrained expressions $(\Delta, C[s])$ as targets where s is the sub-expression that is to be modified, C is the context representing the position, and Δ is a freshness constraint.

Definition 4.14. *Let $(\nabla \vdash l \to r)$ be a rewrite rule and let $(\Delta, C[s])$ be the constrained expression to be rewritten, where $Var(\nabla, l \to r) \cap Var((\Delta, C[s]) = \emptyset$. The condition can be achieved by renaming of $(\nabla, l \to r)$.) A* rewrite step *is defined as follows:*

Let (∇', σ) be a most general matcher of $(\nabla, l) \preceq (\Delta, s)$ computed with NomMatchAS and let $\nabla'' = \nabla' \cup \nabla\sigma$. Then the result of rewriting is $(\Delta \cup \nabla'', C[r\sigma])$.
Thus the rewrite step is $(\Delta, C[s]) \to (\Delta \cup \nabla'', C[r\sigma])$.

For a rewrite system $\mathcal{R}$, this defines a rewriting relation $\xrightarrow{\mathcal{R}}$ on constrained expressions over NL_{AS} with transitive closure $\xrightarrow{\mathcal{R},*}$.

4.4 Overlaps, Critical Pairs and Knuth-Bendix Confluence Criterion

Now we define overlap, join and critical pairs, the adapted Knuth Bendix-criterion for confluence and sketch completion in our setting of nominal rewriting on NL_a, where rules are formulated in NL_{AS}.

Definition 4.15. *An* overlap *of two rewrite rules is computed by the following algorithm. First the rewrite rules are renamed such that they are variable-disjoint, where also the same rule may be used twice: $(\nabla_1, l_1 \to r_1)$ and $(\nabla_2, l_2 \to r_2)$. Select a non-variable expression-position p in l_1, represented by a context C, such that $C[l_1'] = l_1$ and the hole of C is at position p, and the expression at the position p is not a variable. Apply the unification algorithm in [23] to $(\nabla_1, l_1') \doteq (\nabla_2, l_2)$. If there is an overlap, then the result of the unification algorithm is a most general unifier (∇, σ). The resulting overlap is the constrained expression is $(\Delta, l_1\sigma)$ where $\Delta = (\nabla \cup \nabla_1 \cup \nabla_2)\sigma$.*

The critical pair *consists of the corresponding rewriting results $((\Delta, r_1\sigma), (\Delta, C\sigma[r_2\sigma]))$. The* overlap triple *is then $((\Delta, l_1\sigma), (\Delta, r_1\sigma), (\Delta, C\sigma[r_2\sigma]))$. If $(\Delta, r_1\sigma) \equiv_V (\Delta, C\sigma[r_2\sigma])$ where $V = Var(\Delta, r_1\sigma, r_2\sigma)$ holds, the critical pair is trivial.*

Note that the technical treatment is slightly different from the criterion on first-order theories, insofar as rewriting and joining in NL_{AS} uses the common freshness constraints of a rewriting sequence.

Definition 4.16 (Nominal Knuth-Bendix Confluence Criterion). *Let $\mathcal{R}$ be a finite nominal rewrite system over NL_{AS}. Let the following two properties hold:*

1. *Rewriting terminates on NL_{AS}.*
2. *All critical pairs can be joined as follows: For every overlap triple $((\Delta, s), (\Delta, s_1), (\Delta, s_2))$ according to Definition 4.15 either the critical pair is trivial, or there are $\mathcal{R}$-reduction sequences $(\Delta, s_1) \to (\Delta_{1,1}, s_{1,2}) \to \ldots \to (\Delta_{1,k_1}, s_{1,k_1})$ and $(\Delta, s_2) \to (\Delta_{2,1}, s_{2,2}) \to \ldots \to (\Delta_{2,k_2}, s_{2,k_2})$ such that $(\Delta_{1,k_1}, s_{1,k_1}) \equiv_V (\Delta_{2,k_2}, s_{2,k_2})$, where $V = Var((\Delta, s)$.*

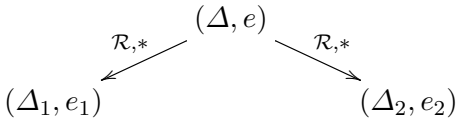
Then we conclude that the rewrite relation of $\xrightarrow{\mathcal{R},NL_a}$ is terminating and confluent. We can also conclude that the congruence generated by $\mathcal{R}$ on NL_a is decidable by rewriting.

4.5 Proofs of Correctness

Lemma 4.17. *Let $\mathcal{R}$ be a finite nominal rewrite system over NL_{AS}. If C is a ground context and $s \xrightarrow{\mathcal{R},NL_a} s'$ for two ground expressions, then also $C[s] \xrightarrow{\mathcal{R},NL_a} C[s']$.*

Proof. This follows from Definition 4.14 of rewriting.

Lemma 4.18. *Let $\mathcal{R}$ be rewrite system, and let (Δ, e) be a constrained expression with $Var(\Delta, e) = V$ and let $(\Delta_1, e_1), (\Delta_2, e_2)$ be two constrained expressions, which arise from different branches during rewriting of (Δ, e), i.e.*



Then $(\Delta_1, e_1) \equiv_V (\Delta_2, e_2)$ holds iff the second criterion holds, i.e.

$$\forall V \forall V_1, V_2 : (\Delta_1 \cup \Delta_2 \implies e_1 \sim e_2)$$

Proof. Because (Δ_i, e_i) arise from rewriting and no shared new variables can be introduced during the procedure, the introduced variables along the different reduction sequences are disjoint. Due to the the matching condition, for $\Delta_i' = \Delta \setminus \Delta_i$ the formula $\forall V \exists V_i : (\Delta \implies \Delta_i')$ holds.

Using this we get the equivalence between the formulas $\Delta_1 \iff \Delta_2$ and $\Delta \implies (\Delta_1' \iff \Delta_2')$. The first condition is then equivalent to:

$$\forall V \exists V_1, V_2 : (\Delta \implies (\Delta_1' \iff \Delta_2'))$$

Because for every choice of V that satisfies Δ one can always choose V_i, s.t. Δ_i' holds, the formula must hold as well.

Theorem 4.19. *Let $\mathcal{R}$ be a set of rewrite rules over NL_{AS}. If the Knuth-Bendix confluence criterion in Definition 4.16 holds, and if $\xrightarrow{\mathcal{R}}$ is terminating, then the rewrite relation $\xrightarrow{R,NL_a}$ is confluent.*

Proof. Due to the Hindley-Rosen Lemma, it is sufficient to show that $\xrightarrow{R,NL_a}$ is locally confluent. There are three types of divergences:

(i) Two reduction steps of $\xrightarrow{R,NL_a}$ at independent positions in an NL_a-expressions. Then the reductions are $s_1 \xrightarrow{R,NL_a} s_1'$, $s_2 \xrightarrow{R,NL_a} s_2'$, and since reductions in contexts are always possible by Lemma 4.17: $C[s_1, s_2] \to C[s_1', s_2]$ and $C[s_1, s_2] \to C[s_1, s_2']$ can both be reduced to $C[s_1', s_2']$, and hence joined.

(ii) The reduction steps of $\xrightarrow{R,NL_a}$ are at dependent positions (the overlap at or below a variable position) in an NL_a-expression. Then the situation can be captured by $C_1[C_2[s]] \xrightarrow{R,NL_a} C_1[C_2'[s, \ldots, s]]$ and $C_1[C_2[s]] \xrightarrow{R,NL_a} C_1[C_2[s']]$, where the rewrites are $\nabla \vdash \overline{C_2}[X] \xrightarrow{R} \overline{C_2}'[X, \ldots, X]$ with $C_2[s] = \overline{C_2}[X]\gamma$ s.t. $\nabla\gamma$ is valid and $\nabla_s \vdash \bar{s_0} \xrightarrow{R} \bar{s_0}'$ with $s = C_3[s_0\rho]$ for some NL_a context $C_3[]$ and ground substitution ρ with $\nabla_s\rho$ valid.
Since $s \xrightarrow{R,NL_a} s'$, we have $C_1[C_2'[s, \ldots, s]] \xrightarrow{R,NL_a} C_1[C_2'[s', \ldots, s']]$ by Lemma 4.17. The same rewrite step for $C_2[s']$ as $C_2[s]$ is permitted, since the free atoms of s' are all contained in s, and thus the constraints cannot block this rewrite step. It yields $C_1[C_2'[s', \ldots, s']]$. Hence, the expression can be joined by perhaps several reduction steps.

(iii) The two reduction steps are at dependent positions, but not at or below a variable position. This corresponds to a critical pair. Since rewriting $\xrightarrow{R,NL_a}$ is derived from the general rewrite rules, there is a critical pair that has these two reductions as instance. By assumption, the critical pair can be joined up to $\equiv_V$ where V is the set of variables of the overlap. Then Lemma 4.18 shows that the instance critical pair can be joined such that the final expressions are α-equivalent. Note that the used ground substitution may be extended during the reductions, but only for bound variables.

Theorem 4.20. *Let $\mathcal{R}$ be a rewrite system over NL_{AS}. If the Knuth-Bendix Confluence Criterion in Definition 4.16 holds, then the rewrite theory $=_{\mathcal{R},NL_a}$ on NL_a is decidable.*

Proof. This follows from Theorem 4.19, and the fact that all (finite) critical pairs can be effectively computed and also tested whether they are joinable. Moreover, rewriting is terminating by assumption, the rewrite steps are effective, and the final test $\equiv_V$ is also decidable.

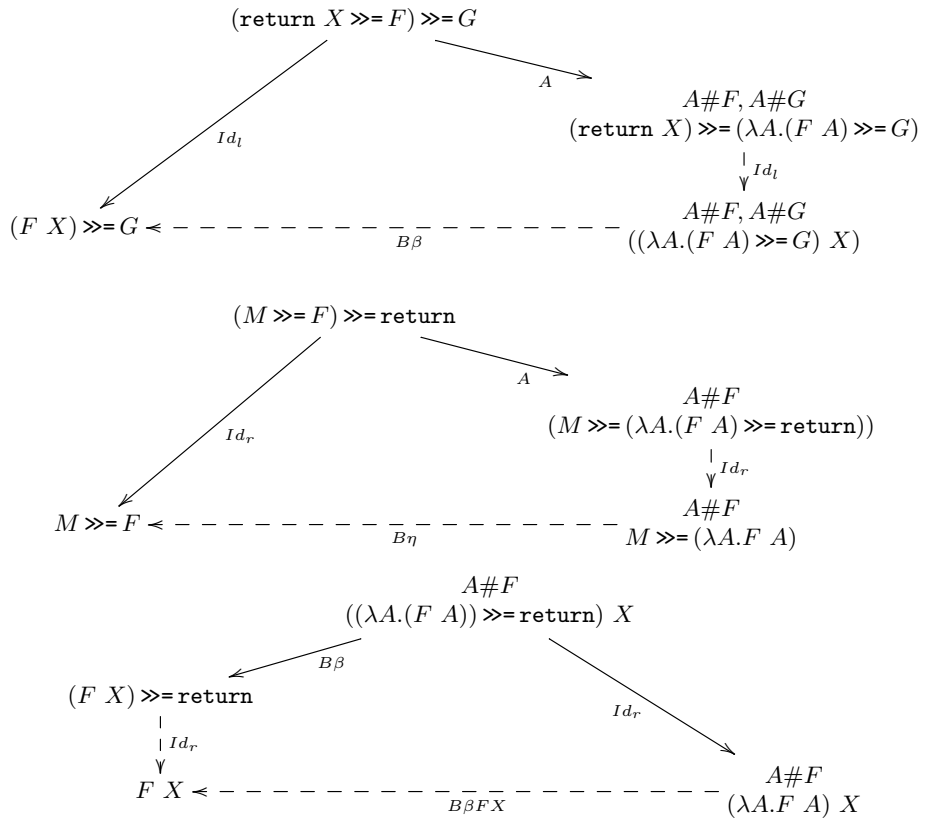


Fig. 3. Completion of the Monad Rules

Remark 4.21. For a finite rewrite System $\mathcal{R}$, we have a decidable test for confluence on NL_a, but not on the general level. To investigate the issue of confluence on the constrained expressions is future work. This might entail the generalization of the current confluence test and/or the replacement of $\equiv_V$ with some $\equiv'$ which is independent of the start expression (Δ, e) of the critical triple.

5 A Convergent Rewrite System for the Monad Laws

As an extended example, illustrating also the ideas and potentials of the nominal modeling and unification in rewriting, in particular with atom-variables, we consider the monad laws [28]. An informal explanation is that monads are a functional implementation of sequential actions, as an extension of the lambda calculus, where $a_1 \gg\!= a_2$ means a sequential combination of actions: a_1 is executed before a_2, and the return-value v of a_1 is used in action a_2, written in lambda notation as $(a_2\ v)$. These are used as programming device in Haskell [15, 11, 16] for programming with state and to implement IO and concurrency. Besides the operational behavior, there is a set of monad laws, describing the desired behavior of monadic combinations as a set of rewrite rules (see below). [10] used second-order unification, which is modulo the theory defined by the α, β, and η axioms, to show confluence. However, second-order unification is undecidable, and thus the application of this idea to other examples will in general lead to undecidable algorithmic questions. In addition, η is not correct in call-by-need functional languages like Haskell. Thus, we use (decidable) nominal unification with atom-variables to obtain also confluence, however, for a finer notion of unification and of equivalence, since we use only α-equivalence. An application of the rewrite system may be normalization of larger monadic expressions. This would require the correctness of the monad theory in the programming language in question.

We will use the following encoding: `return` is a function symbol of arity 0, `app` and `≫=` are function symbols of arity 2, where we write `≫=` as infix, and `app` as juxtaposition. A, B, C denote atom-variables, and other upper-case letters X, F, G, M expression-variables.

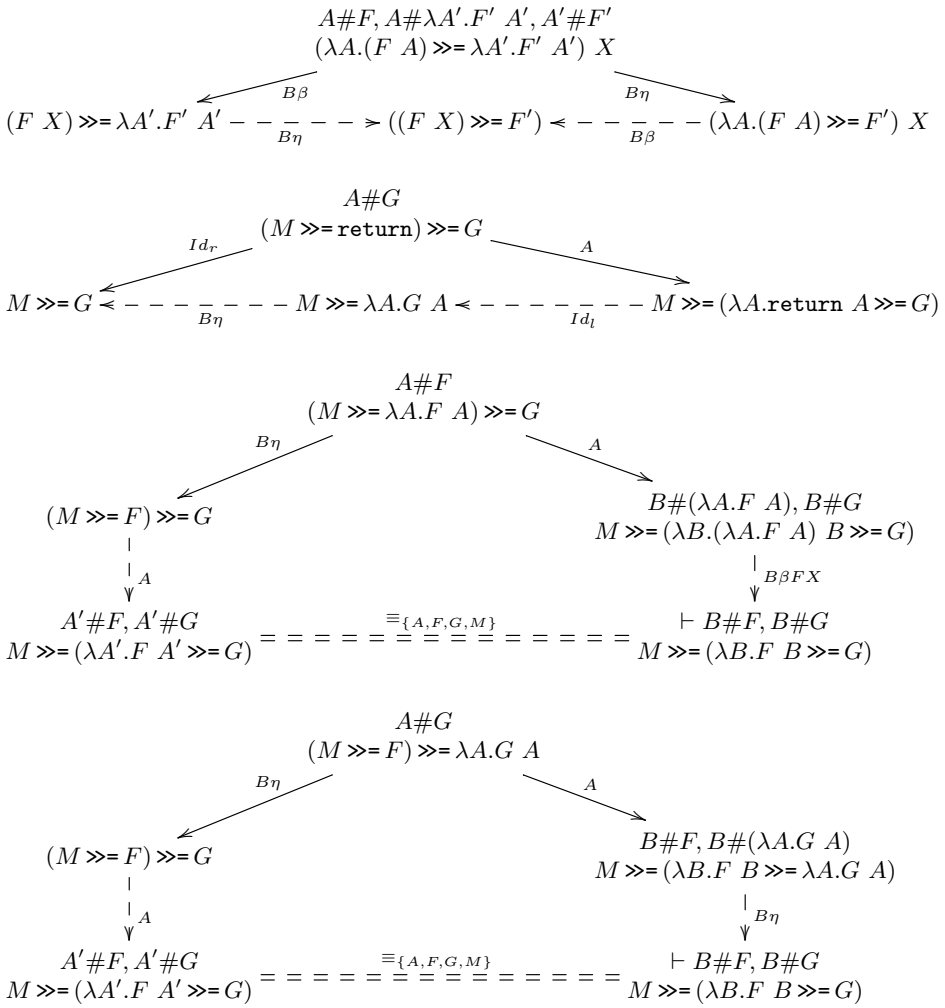


Fig. 4. Joining the nontrivial critical pairs of Monad Theory: diagrams first part.

The three monad laws are encoded as rewrite rules as follows:

$$\begin{array}{lll} (Id_l)\ \emptyset & \vdash (\texttt{return}\ X)\ \texttt{>>=}\ F & \to F\ X \\ (Id_r)\ \emptyset & \vdash M\ \texttt{>>=}\ \texttt{return} & \to M \\ (A)\ \ A\#F, A\#G & \vdash (M \texttt{>>=} F) \texttt{>>=} G & \to M \texttt{>>=} (\lambda A.(F\ A)\ \texttt{>>=}\ G) \end{array}$$

It is a bit unusual that there are variables in the right-hand side that do not occur in the left-hand side. In this case we assume that the names of the introduced variables are fresh ones, i.e. these do not occur elsewhere in the same computation. Note also that the constraints in rules essentially restrict the introduced variables on the right hand side of rewrite rules. An overlap of the rules Id_r, Id_l with A leads to two extra rules $B\beta$, $B\eta$, and an overlap of $B\beta$ with Id_r to $B\beta FX$, see Fig. 3.

Note that the three rules $B\eta$, $B\beta$, and $B\beta FX$ are consequences of the monad laws as equations (w.r.t. α-equivalence), see Fig. 3. Note also that β or η as more general rules are inappropriate, since they would prevent an embedding of monad laws into call-by-need lambda calculi with contextual equivalence due to inconsistency of η in these semantics, and since the rule β (in general) leads to nontermination of rewrite rules [19].

$$\begin{array}{lll} (B\eta) & A\#F & \vdash (M \texttt{>>=} \lambda A.F\ A) & \to M \texttt{>>=} F \\ (B\beta) & A\#F, A\#G & \vdash (\lambda A.(F\ A) \texttt{>>=} G)\ X & \to (F\ X) \texttt{>>=}\ G \\ (B\beta FX) & A\#F & \vdash (\lambda A.(F\ A))\ X & \to (F\ X) \end{array}$$

The combined rewriting system $\mathcal{R}_{monad}$ consists of the 6 rules $\{Id_l,\ Id_r,\ A,\ B\eta,\ B\beta,\ B\beta FX\}$. Note that the rewrite rules satisfy Definition 4.2, which follows, since (A) only introduces bound fresh

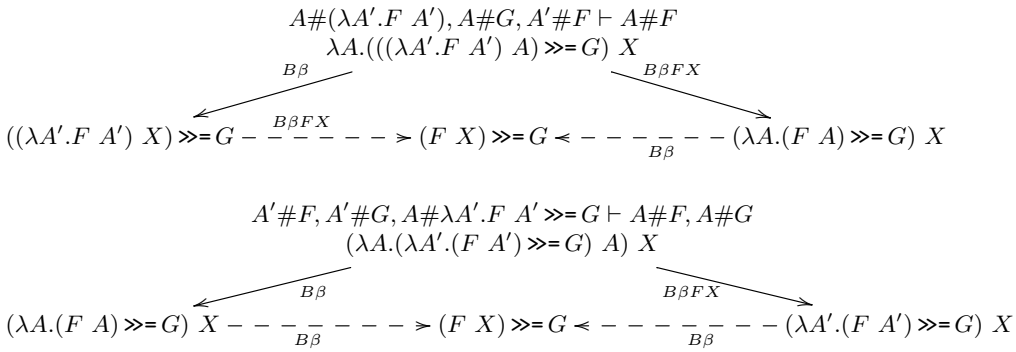


Fig. 5. Joining the nontrivial critical pairs of Monad Theory: $B\beta FX$ and $B\beta$

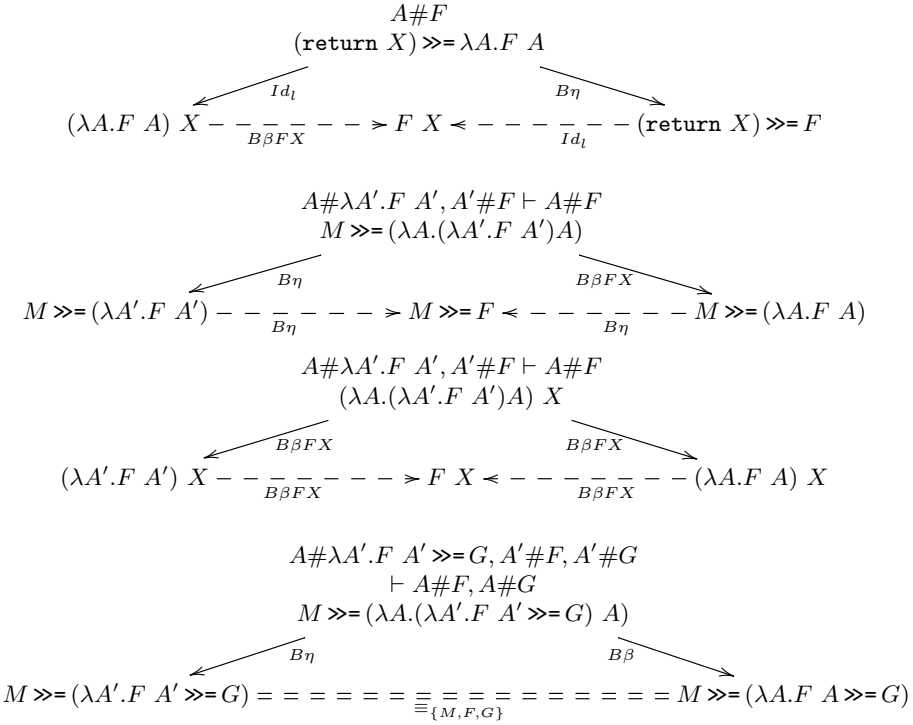


Fig. 6. Joining the nontrivial critical pairs of Monad Theory: diagrams second part.

names, and the other rules do not introduce fresh names. It is terminating, since the rules either strictly decrease the size, or move the ≫=-bracketing to the right and increase the size by a constant (say 3). Termination of the rewrite system is a prerequisite for applying the Knuth-Bendix confluence test.

The nominal monad rewrite system $\mathcal{R}_{monad}$ is between first-order and higher-order. We use nominal unification for computing the critical pairs and nominal matching for rewriting, where we permit atom-variables in every case.

The following table shows the overlap possibilities, where three are used for completion

	Id_l	Id_r	A	$B\eta$	$B\beta$	$B\beta FX$
Id_l		*trivial*	*Fig.*3	*Fig.*6	−−	−−
Id_r			*Fig.*3, and 4	−−	*Fig.*3	−−
A			*Fig.*6	*Fig.*4, c) and d)	−−	−−
$B\eta$				−−	*Fig.*4 and 6	*Fig.*6
$B\beta$					*Fig.*8	*Fig.*5
$B\beta FX$						*Fig.*6

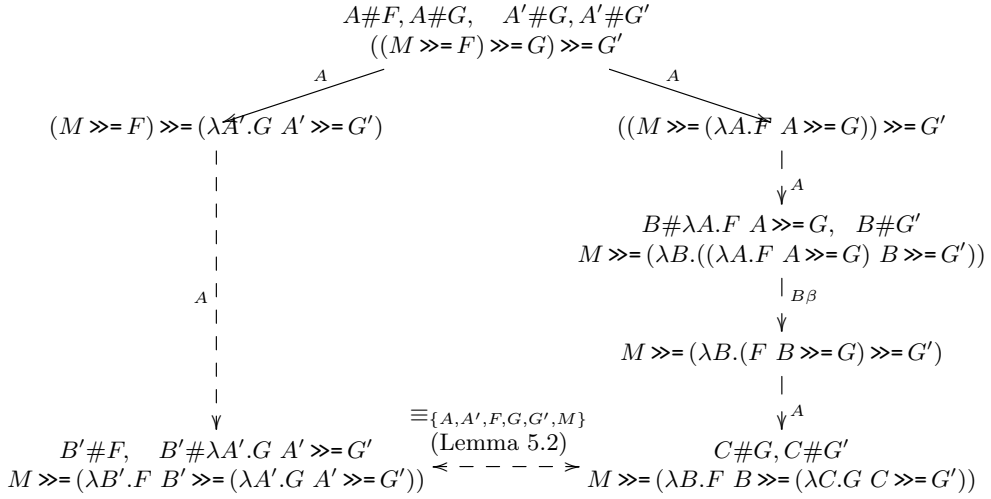


Fig. 7. Joining the nontrivial critical pairs of Monad Theory: the A-A-diagram

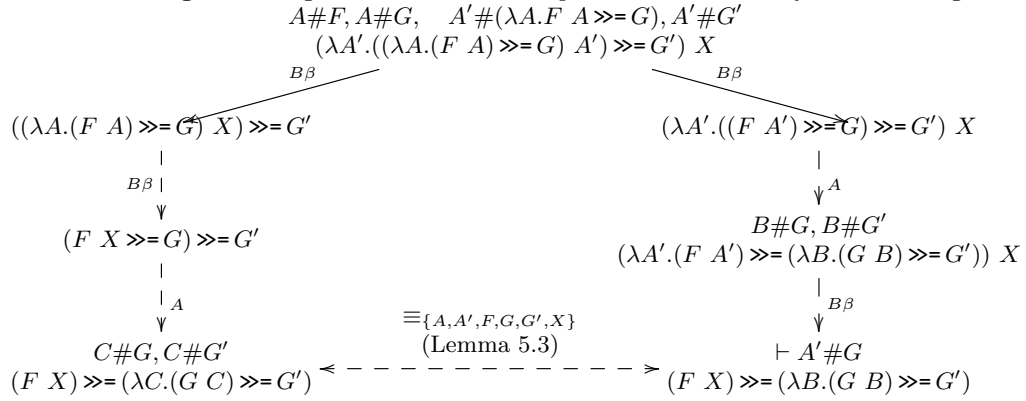


Fig. 8. Joining the nontrivial critical pairs of Monad Theory: the $B\beta$-$B\beta$-diagram

Where *trivial* means that the critical pair is trivial, and $--$ means there is no overlap. We omit $B\beta FX$, since there are no overlaps.

Lemma 5.1. *The final expressions in Fig. 7 can be joined under the union of all constraints.*

Proof. The constraints are: $A\#\lambda A'.F\ A', A'\#F$. this implies $A\#F$ by a case analysis whether A and A' are equally instantiated or not. This in turn implies $F' \sim (A\ A')\cdot F'$.

Before we prove that the non-trivial A-A-overlap can be joined, we look at a seemingly trivial overlap. It is reducing $(M \gg\!= F) \gg\!= G$ in two ways using (A) on the top: One result is $M \gg\!= (\lambda A.(F\ A) \gg\!= G)$ with $A\#F, G$, and another one is $M \gg\!= (\lambda B.(F\ B) \gg\!= G)$ with $B\#F, G$. The essential step in showing $\equiv_{\{M,F,G\}}$ of the resulting constrained expressions is to argue that $\lambda A.(F\ A) \gg\!= G \sim \lambda B.(F\ B) \gg\!= G$ by arguing that $(F\ A) \gg\!= G \sim (A\ B)\cdot(F\ B) \gg\!= G$, where the freshness constraints show that this reasoning is correct.

The pair arising from the proper overlap of the associativity rule (A) with itself needs a check if the final expressions are equivalent under the union of the constraints, which requires a bit more of computations.

Lemma 5.2. *The final expressions in the A-A diagram in Fig. fig:monad-joins-B are joinable.*

Proof. The freshness constraints in the A-A-diagram in Fig. 6 are as follows:

$$\begin{array}{ll} \text{left:} & A\#F, G\ A'\#G, G'\ B'\#F\ B'\#\lambda A'.G\ A' \gg\!= G' \\ \text{right:} & A\#F, G\ A'\#G, G'\ B\#G'\ B\#\lambda A.F\ A \gg\!= G \quad C\#G, G' \end{array}$$

We can derive the following further constraints:

(1) (right): $B\#F, G$: If $\gamma(B) = \gamma(A)$, then this follows from $A\#F, G$.
If $\gamma(B) \neq \gamma(A)$, then it follows from $B\#\lambda A.F\ A \gg\!= G$.
(2) (left) $B'\#G, G$': If $\gamma(B') = \gamma(A')$, then this follows from $A\#G, G'$.
If $\gamma(B) \neq \gamma(A')$, then it follows from $B'\#\lambda A'.G\ A' \gg\!= G'$.

Now we have to verify the following equivalence:

$$\begin{array}{c} M \gg\!= (\lambda B'.F\ B' \gg\!= (\lambda A'.G\ A' \gg\!= G') \\ \equiv_{\{A,A',F,G,G',M\}} \\ M \gg\!= (\lambda B.F\ B \gg\!= (\lambda C.G\ C \gg\!= G')) \end{array}$$

This is equivalent to verifying:

$$(\lambda B'.F\ B' \gg\!= (\lambda A'.G\ A' \gg\!= G')) \equiv_{\{A,A',F,G,G',M\}} (\lambda B.F\ B \gg\!= (\lambda C.G\ C \gg\!= G'))$$

Using the given and derived constraints, and since $B, B'\#F, G, G'$, this is equivalent to verifying:

$$F\ B' \gg\!= (\lambda A'.G\ A' \gg\!= G') \equiv_{\{A,A',F,G,G',M\}} F\ B' \gg\!= (\lambda (B\ B')\cdot C.(G\ (B\ B')\cdot C) \gg\!= G'))$$

A case analysis shows the remaining parts:

(1) If $\gamma(A') = \gamma((B\ B')\cdot C)$, then this is correct by a direct decomposition.
(2) If $\gamma(A') \neq \gamma((B\ B')\cdot C)$, then after an application of $(A'\ \ (B\ B')\cdot C))$, and due to $A'\#G, G'$, and $B, B'\#G, G'$ a direct decomposition shows equality.

Thus, we have shown that for $V = \{A, A', F, G, G', M\}$ the formula:

$$\forall V : \nabla_l \cup \nabla_r \implies (\lambda B'.F\ B' \gg\!= (\lambda A'.G\ A' \gg\!= G')) \sim (\lambda B.F\ B \gg\!= (\lambda C.G\ C \gg\!= G'))$$

holds. Lemma 4.18 then implies the $\equiv_V$-equivalence and the join.

Lemma 5.3. *The final expressions in the $B\beta$-$B\beta$-diagram are joinable.*

Proof. The proof is analogous to the proof in lemma 5.2

As a summary we obtain:

Theorem 5.4. *The monad axioms (Id_l), (Id_r) and (A) modulo α-equivalence have as completion the additional rewrite rules $(B\eta)$, $(B\beta)$ and $(B\beta FX)$.*
The rewrite system consisting of these 6 rewrite rules is terminating and confluent (as a ground rewriting system), and a decision algorithm for the word-problem of the rewrite-theory of monad axioms in NL_a, modulo α-equality.

Proof. This follows from our computations in this section, the join-diagrams in this section, in particular Lemma 5.2, and the correctness of the Knuth Bendix confluence test for NL_{AS} in Theorems 4.19,4.20.

6 Comparison of Our Approach with Nominal Rewriting with Atoms

Nominal rewriting was introduced by [7] as a way to define equivariant relations on NL_a similar to first order rewriting. We provide a description of this approach while using our notations. To distinguish it from our formalism, it will be referred to as equivariant rewriting.

Definition 6.1. *Let NL_{aS} be the nominal language built with atoms and expression-variables. Let $s, t \in NL_{aS}$ be two expressions and let ∇ be freshness environment on NL_{aS}-expressions. A rewrite judgement is defined as: $\nabla \vdash l \to r$.*

The semantics of the induced relation on NL_a can be defined as follows (note that several equivalent definitions exist).

1) The relation $\to$ on NL_a is equivariant, i.e. if $e_1 \to e_2$ holds, then $\pi \cdot e_1 \to \pi \cdot e_2$ holds as well for all permutations π on atoms.
2) For all ground substitutions γ for which $\nabla\gamma$ is valid, $s\gamma \to t\gamma$ holds.

During rewriting the first condition is "hidden" in an equivariant matching procedure. That is, rather than trying to match two NL_{aS} constrained expressions, $(\nabla, l) \preceq (\Delta, e)$ with only a substitution and a freshness environment, one can also use a permutation on atoms π to make the two expressions equal.

Specifically, this means finding a triple (∇', θ, π) s.t. $\nabla' \vDash l^\pi\theta \sim e$ and $\Delta \vDash \nabla^\pi\theta \cup \nabla'$, where l^π denotes the application on π only on atoms – not on expression-variables.

As a result, the atoms in such a rewrite rule gain a variable like character. As a matter of fact, one could define an equivalent matching procedure in NL_{aAS} – different from the one used in this paper.

To do that, one would map every atom a_i on the left side of the matching equation $(\nabla, l) \preceq (\Delta, t)$ to an atom-variable A_i and utilize the additional freshness constraints $\nabla' = \{A_i \# A_j \mid i, j \in \{1, \ldots, k\}, i < j\}$ to enforce, that any solution of the matching problem matches the atom-variables to different atoms. The part of the solution which matches atom-variables to atoms would then function like the permutation *pi* in equivariant matching, with the only difference of it being a bijection between atom-variables and atoms, rather than atoms and atoms.

This brings us to the first obvious difference of our formalism to the framework of [7], the usage of atom-variables rather than atoms.

Example 6.2. Consider a simple version of (cpcx) in the concurrent calculus CHF [20, 19] or in other functional programming calculi, formulated as a rewrite rule:

$$\{B \# A_i \mid i \in \{1, \ldots, k\}\} \vdash \texttt{let}\ B = c\ A_1 \ldots A_k\ \texttt{in}\ B \to\ \texttt{let}\ B = c\ A_1 \ldots A_k\ \texttt{in}\ c\ A_1 \ldots A_k$$

where we do not care about the equality/inequality of the variable names occurring in this context. To define an equivalent relation $\to$ on NL_a in equivariant rewriting, one would need to add a rule for each variant of equality/inequality of the atom-variables, yielding exponentially many rules.

The second difference, which is more subtle and at the same time semantically more meaningful, is which expressions can in principle be matched.

Example 6.3. Consider the η-expansion of the lambda calculus formulated as a rule in equivariant rewriting:

$$a \# S \vdash S \to \lambda a.\texttt{app}\ S\ a$$

Within the framework of equivariant rewriting, the matching problem

$$(\{a \# S, s\}) \preceq (\emptyset, S')$$

has no matcher, even though there always exists an atom b, s.t. $b \# S'$ holds. However,

$$(\{a \# S, s\}) \preceq (\{b \# S'\}, S')$$

has a matcher in $(\emptyset, \{S \mapsto S'\}, (a\ b))$.

In equivariant rewriting no new freshness constraints can be introduced, even if such an introduction would be semantically correct. This has the benefit, that the constrained implication check of the procedure, $\Delta \vDash \nabla^\pi\theta \cup \nabla'$, collapses to checking a subset property $\nabla^\pi\theta \cup \nabla \subseteq \Delta$ which in turn allows fixing the initial constraint set Δ. However, it introduces a mismatch between what can semantically be matched and what is procedurally matched. [8]. note that "this mismatch between nominal rewriting and nominal algebra could be solved by including fresh atom generation in the definition of a rewriting step". This is in fact what is happening during the rewriting step of the approach taken in this paper, with the benefit of being able to match things like η-expansion directly and the downside of having to reason about a changing freshness environment.

7 Conclusion

We have developed a nominal matching algorithm for constrained nominal expressions, and determined the complexity. We succeeded in formulating a variant of the Knuth Bendix confluence test for rewrite system based on our nominal language NL_{AS} with atom-variables, where the objects to be rewritten are constrained expressions. Thus we obtained a decidable criterion for testing confluence, which in the successful case, leads to a decidable α-equivalence check for theories on NL_{AS}, i.e. we obtain also decidability of word-problems modulo α-equivalence.

Thus, our method extends the rewriting and confluence check method of [6] by improving the treatment of (dis-)equality of atoms in a more systematic way.

We also investigated as an extended example the higher-order theory of monads, which illustrates the application of the Knuth Bendix confluence criterion. We also obtained a result for theory of monads:A confluent rewrite system for monads is constructed as a completion of the three defining rules. This is more fine-grained than the system in [10], which uses full beta-reduction.

Future work is to investigate whether also for (general) NL_{AS}-rewriting on constrained expressions a variant of the Knuth Bendix criterion for confluence can be constructed. Another direction of future work is to extend the Knuth Bendix criterion for nominal rewriting with atom-variables to rewriting modulo an equivalence relation.

Furthermore, we hope to extend the method to equational theories that are defined in more general ways, for example using descriptions of infinite sets of equations by context variables in rules, and applying the nominal unification algorithm as described in [22].

We also plan to implement a confluence tester for nominal term rewriting systems using our Knuth Bendix algorithm with atom-variables with atom-variables

A potential application are some reduction rules in the call-by-need calculus of [1] and also to the concurrent Haskell variant CHF [20, 21], like $\texttt{let}\ y = v\ \texttt{in}\ C[y] \to \texttt{let}\ y = v\ \texttt{in}\ C[v]$, where v is a value, or similar rules.

Bibliography

[1] Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. In *POPL'95*, pages 233–246, San Francisco, CA, 1995. ACM Press.

[2] Mauricio Ayala-Rincón, Maribel Fernández, Murdoch James Gabbay, and Ana Cristina Rocha-Oliveira. Checking overlaps of nominal rewriting rules. *ENTCS*, 323:39–56, 2016.

[3] Christophe Calvès and Maribel Fernández. A polynomial nominal unification algorithm. *Theor. Comput. Sci.*, 403(2-3):285–306, 2008.

[4] James Cheney. *Nominal Logic Programming*. PhD thesis, Cornell University, Ithaca, NY, August 2004.

[5] James Cheney. Equivariant unification. *JAR*, 45(3):267–300, 2010.

[6] Maribel Fernández and Murdoch J. Gabbay. Nominal rewriting. *Information and Computation*, 205(6):917–965, jun 2007.

[7] Maribel Fernández, Murdoch J. Gabbay, and Ian Mackie. Nominal rewriting systems. In *Proceedings of the 6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '04, pages 108–119, New York, NY, USA, 2004. ACM.

[8] Maribel Fernández and Murdoch James Gabbay. Closed nominal rewriting and efficiently computable nominal algebra equality. In Karl Crary and Marino Miculan, editors, *Proceedings 5th International Workshop on Logical Frameworks and Meta-languages: Theory and Practice, LFMTP 2010, Edinburgh, UK, 14th July 2010.*, volume 34 of *EPTCS*, pages 37–51, 2010.

[9] Maribel Fernández and Albert Rubio. Nominal completion for rewrite systems with binders. In Artur Czumaj, Kurt Mehlhorn, Andrew M. Pitts, and Roger Wattenhofer, editors, *Proc. 39th ICALP Part II*, volume 7392 of *LNCS*, pages 201–213. Springer, 2012.

[10] Makoto Hamana. How to prove your calculus is decidable: practical applications of second-order algebraic theories and computation. *PACMPL*, 1(ICFP):22:1–22:28, 2017.

[11] Haskell-community. Haskell main website, 2019. `www.haskell.org`.

[12] D. Knuth and P. B. Bendix. Simple word problems in universal algebra. In J. Leech, editor, *Computational problems in abstract algebra*, pages 263–297. Pergamon Press, 1970.

[13] Jordi Levy and Mateu Villaret. An efficient nominal unification algorithm. In Christopher Lynch, editor, *Proc. 21st RTA*, volume 6 of *LIPIcs*, pages 209–226. Schloss Dagstuhl, 2010.

[14] C. Papadimitriou. *Computational Complexity*. Addison-Wesley,, Boston, MA, USA, 1994.

[15] Simon L. Peyton Jones. *Haskell 98 language and libraries: the Revised Report*. Cambridge University Press, 2003. `www.haskell.org`.

[16] Simon L. Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Proc. 23rd ACM POPL 1996*, pages 295–308. ACM, 1996.

[17] Andrew Pitts. Nominal techniques. *ACM SIGLOG News*, 3(1):57–72, February 2016.

[18] Andrew M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge University Press, New York, NY, USA, 2013.

[19] D. Sabel and M. Schmidt-Schauß. Conservative concurrency in Haskell. In *LICS'12*, pages 561–570. IEEE, 2012.

[20] David Sabel and Manfred Schmidt-Schauß. A contextual semantics for concurrent Haskell with futures. In Peter Schneider-Kamp and Michael Hanus, editors, *Proc. 13th ACM PPDP 2011*, pages 101–112. ACM, 2011.

[21] Manfred Schmidt-Schauß and Nils Dallmeyer. Space improvements and equivalences in a functional core language. *CoRR*, abs/1802.06498, 2018.

[22] Manfred Schmidt-Schauß and David Sabel. Nominal unification with atom and context variables. In Hélène Kirchner, editor, *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK*, volume 108 of *LIPIcs*, pages 28:1–28:20. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

[23] Manfred Schmidt-Schauß, David Sabel, and Yunus D. K. Kutz. Nominal unification with atom-variables. *J. Symb. Comput.*, 90:42–64, 2019.

[24] Christian Urban. Nominal techniques in Isabelle/HOL. *J. Autom. Reasoning*, 40(4):327–356, 2008.

[25] Christian Urban and Cezary Kaliszyk. General bindings and alpha-equivalence in nominal Isabelle. *Log. Methods Comput. Sci.*, 8(2), 2012.

[26] Christian Urban, Andrew M. Pitts, and Murdoch Gabbay. Nominal unification. In *17th CSL, 12th EACSL, and 8th KGC*, volume 2803 of *LNCS*, pages 513–527. Springer, 2003.

[27] Christian Urban, Andrew M. Pitts, and Murdoch J. Gabbay. Nominal unification. *Theor. Comput. Sci.*, 323(1–3):473–497, 2004.

[28] Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer, 1995.