32nd International Workshop on Unification (UNIF 2018), Informal Proceedings
Oxford, 7th July, 2018

Rewriting with Generalized Nominal Unification

Yunus D. K. Kutz[1] and Manfred Schmidt-Schauß[2*]

[1] Goethe-University Frankfurt am Main, Germany, kutz@ki.informatik.uni-frankfurt.de
[2] Goethe-University Frankfurt am Main, Germany, schauss@ki.informatik.uni-frankfurt.de

Abstract

We consider rewriting, critical pairs and confluence tests on rewrite rules using nominal notation. Computing critical pairs is done using nominal unification, and rewriting using nominal matching. The progress is that we permit atom variables in the notation and in the unification algorithm, which generalizes previous approaches using usual nominal unification

Keywords: nominal unification, atom variables, nominal rewriting, Knuth-Bendix criterion,

1 Introduction

The goal of this paper is to demonstrate the expressive power of nominal unification with atom variables [14] also in applications, where we consider rewriting and critical pairs ala Knuth-Bendix [7] in a higher-order language with alpha-equivalence and nominal modeling, where in the nominal unification 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 generalizes the approach in [5, 2]. The application of nominal unification with atom variables avoids guessing of (dis-)equality of atom, which is necessary not only as a pre-procedure by the previous uses of nominal unification in rewriting, but also in rewriting sequences in every single rewriting step.

Nominal techniques [10, 9] support machine-oriented reasoning on the syntactic level for higher-order languages and support alpha-equivalence. An algorithm for nominal unification was first described in [17], which outputs unique most general unifiers. More efficient algorithms are given in [3, 8], also exhibiting a quadratic algorithm. The approach is also used in higher-order logic programming [4] and in automated theorem provers like nominal Isabelle [15, 16]. Nominal unification was generalized to permit also atom variables [14] where also in the generalization, unique most general unifiers are computed, while the decision problem is NP-complete.

As an extended example, illustrating also the ideas and potentials of the nominal modelling and unification in rewriting, in particular with atom variables, we consider the monad laws [18]. An informal explanation is that monads are an implementation of sequential actions, as extension of 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 the next action, written in lambda notation as $(a_2\ v)$. Besides the operational behavior, there is a set of monad laws, describing the desired behavior of monadic combination as an equational theory (see below). Hamana[6] used second-order unification to show confluence. However, second-order unification is undecidable, and thus the extension of this idea will in general lead to undecidable algorithmic questions. Thus, we use (decidable) nominal unification with atom-variables to obtain the same result, however, for a finer notion of unification and of equivalence.
We will use the following encoding: `return` is a function symbol of arity 0, `app` $\gg\!=$ are function

*supported by the Deutsche Forschungsgemeinschaft (DFG) under grant SCHM 986/11-1.

symbols of arity 2, where we write $\gg\!=$ as infix, and app as juxtaposition. A, B, C denote atom-variables, and other upper-case letters expression-variables.

The three monad laws are encoded as follows:

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

Additionally we add η-reduction and as a very weak version of β-reduction, we add $B\beta$. Note that $B\beta$ is a consequence of the monad laws as equations (w.r.t. α-equivalence), see Fig. 1.

$$\begin{array}{llll} (\eta) & A\#F & \vdash (\lambda A.F\ A) & \to \quad F \\ (B\beta) & A\#F,G & \vdash (\lambda A.(F\ A) \gg\!= G)\ X & \to \quad (F\ X) \gg\!= G \end{array}$$

Note that the combined rewriting system is terminating, which is a prerequisite for applying the technique of local confluence for showing confluence of rewrite system.

Our rewrite system is higher-order, but we only use nominal unification for computing the critical pairs and nominal matching for rewriting, where we permit atom variables in every case.

There are the following critical pairs, using nominal unification with atom variables:

1. $(B\beta)$ in (η).
2. (Id_l) in $(B\beta)$.
3. (Id_l) equal to Id_r.
4. (Id_l) in A.
5. (Id_r) in $(B\beta)$.
6. (Id_r) in A.
7. A in A

The critical pairs (2), (3), and (5) are trivial or easily joinable. The remaining ones are treated separately (see Fig. 1). The pair arising from the overlap of the associativity rule with itself needs a check if two freshness environment-expression pairs are equivalent (which indeed they are), which is done comparing the set of ground instances, respecting the freshness constraints.

The next example, also motivating the use of atom-variables, is a rule in the concurrent calculus CHF [12]. It permits rewriting $\texttt{let}\ y = c\ x_1 \dots x_n\ \texttt{in}\ C[y] \to \texttt{let}\ y = c\ x_1 \dots x_n\ \texttt{in}\ C[c\ x_1 \dots x_n]$, and can be applied to $C_1[\texttt{let}\ z = c\ y_1 \dots y_n\ \texttt{in}\ C_2[z]]$ even if the y_i are not pairwise different variables, which is in contrast to usual nominal rewriting using atoms instead of atom variables, since a unique unifier covers all possibilities of equal/unequal atoms.

2 Nominal Rewriting

We first introduce some notation [14].

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:

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

(1)
$$\begin{array}{ccc} A\#F,G & & \\ C\#(\lambda A.(F\ A) \gg\!= G)\ X & \xrightarrow{\eta} & (\lambda A.(F\ A) \gg\!= G)\ X \\ (\lambda C.(\lambda A.(F\ A) \gg\!= G)\ X)\ C & & \\ \downarrow B\beta & & \downarrow B\beta \\ \lambda C.((F\ X) \gg\!= G)\ C & \xrightarrow{\eta} & ((F\ X) \gg\!= G) \end{array}$$

(4)
$$\begin{array}{ccc} A\#F,G & & \\ (\texttt{return}\ X \gg\!= F) \gg\!= G & \xrightarrow{A} & \texttt{return}\ X \gg\!= (\lambda A.F\ A \gg\!= G) \\ \downarrow Id_l & & \downarrow Id_l \\ F\ X \gg\!= G & \xleftarrow{B\beta} & (\lambda A.F\ A \gg\!= G)\ X \end{array}$$

(6)
$$\begin{array}{ccc} A\#G & & \\ (M \gg\!= \texttt{return}) \gg\!= G & \xrightarrow{A} & M \gg\!= (\lambda A.\texttt{return}\ A \gg\!= G) \\ \downarrow Id_r & & \downarrow Id_l \\ M \gg\!= G & \xleftarrow{\eta} & M \gg\!= \lambda A.G\ A \end{array}$$

(7)
$$\begin{array}{ccc} A\#F,G & & \\ A'\#G,G' & \xrightarrow{A} & ((M \gg\!= (\lambda A.F\ A \gg\!= G)) \gg\!= G' \\ ((M \gg\!= F) \gg\!= G) \gg\!= G' & & \downarrow A \\ \downarrow A & & B\#\lambda A.F\ A \gg\!= G \\ (M \gg\!= F) \gg\!= (\lambda A'.G\ A' \gg\!= G') & & B\#G' \\ & & M \gg\!= (\lambda B.((\lambda A.F\ A \gg\!= G)\ B \gg\!= G')) \\ \downarrow A & & \downarrow B\beta \\ & & M \gg\!= (\lambda B.(F\ B \gg\!= G) \gg\!= G') \\ & & \downarrow A \\ B'\#F & & C\#G,G' \\ B'\#\lambda A'.G\ A' \gg\!= G' & \overset{\alpha-inst}{\longleftrightarrow} & \\ M \gg\!= (\lambda B'.F\ B' \gg\!= (\lambda A'.G\ A' \gg\!= G')) & & M \gg\!= (\lambda B.F\ B \gg\!= (\lambda C.G\ C \gg\!= G')) \end{array}$$

Figure 1: Joining the nontrivial critical pairs of Monad Theory

where λ is a binder for atoms. The basic constraint $a\#e$ is valid if a is not free in e and a set of constraints ∇ is valid if all constraints are valid.

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:*

$$\frac{}{a \sim a} \quad \frac{\forall i : e_i \sim e'_i}{(f\ e_1 \dots e_{ar(f)}) \sim (f\ e'_1 \dots e'_{ar(f)})} \quad \frac{e \sim e'}{\lambda a.e \sim \lambda a.e'} \quad \frac{a\#e' \wedge e \sim (a\ b)\cdot e'}{\lambda a.e \sim \lambda b.e'}$$

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 a context C, we have $e_1 \sim e_2$ implies $C[e_1] \sim C[e_2]$.

Let $\mathcal{S}$ be a 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_{AS} with atom-variables is:

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

where π is a permutation and $\emptyset$ denotes the identity. Note that we permit nested permutations. The expression $((\pi{\cdot}A)\ (\pi'{\cdot}A'))$ is a single nested swapping. We assume that permutation application is done as simplification if possible. The inverse π^{-1} of a permutation $\pi = sw_1{\cdot}\dots{\cdot}sw_n$ with swappings sw_i is the expression $sw_n{\cdot}\dots{\cdot}sw_1$.
$AtVar(e)$ are the atom-variables contained in e, $ExVar(e)$ the expression-variables contained in e and $Var(e) = AtVar(e) \cup ExVar(e)$.

The ground language of NL_{AS} is NL_a, i.e. a ground substitution replaces atom variables by atoms and expression variables by expression in NL_a.
A *freshness constraint* ∇ is a set (a Boolean conjunction) of constraints of the form $A\#e$. Note that constraints of the form $\pi{\cdot}A\#e$ are equivalent to $A\#\pi^{-1}{\cdot}e$, hence we omit them from the syntax. ∇ is valid under a ground substitution γ, if $\nabla\gamma$ is valid; this is written as: $\gamma \models \nabla$.

Definition 2.2. *A* rewrite rule *is of the form* $(\nabla, l \to r)$*, where* ∇ *is a freshness constraint and* l, r *are expressions of* NL_{AS} *and* l *is not an atom- nor an expression-variable, and* $ExVar(r) \subseteq ExVar(l)$*. The rewrite rule is also written as* $\nabla \vdash l \to r$*.*

We will illustrate the ideas for rewriting by two examples.

Example 2.3. *Let the rewrite rule (for garbage collection) be* $(\{A\#S'\}, \texttt{let}\ A = S\ \texttt{in}\ S' \to S')$*. Then a rewrite step on the ground expression* $(\texttt{let}\ x = a\ \texttt{in}\ \lambda b.b)$ *without any atom- nor expression-variables can be done as follows: We have to compute a nominal matcher of* $(\texttt{let}\ A = S\ \texttt{in}\ S') \preceq (\texttt{let}\ x = a\ \texttt{in}\ \lambda b.b)$*, which results in a substitution* $\sigma = \{A \mapsto x; S \mapsto a; S' \mapsto \lambda b.b\}$ *and the resulting freshness constraint* $(A\#S')\sigma$ *is valid, since* $x\#\lambda b.b$ *is valid. The resulting expression of the rewriting step is* $\lambda b.b$*.*

This form of application appears to be too restricted, since we also want to rewrite expressions containing atom- and expression-variables, perhaps restricted by freshness constraints. In doing so we gain the ability to rewrite a multitude of related ground expressions (all instances of the expression-constraint pair) and are able to join critical pairs. We generalize the example and permit atom- and expression-variables in the target expression.

Example 2.4. *We use the same rewrite rule as above:* $(\{A\#S'\}, \texttt{let}\ A = S\ \texttt{in}\ S' \to S')$*. Then let the abstract expression be* $(\texttt{let}\ B = S_3\ \texttt{in}\ S_4)$ *with the additional freshness constraint* $B\#S_4$*, which represents a set of ground expressions. Rewriting can informally be done as follows: we have to compute a matcher of* $(\texttt{let}\ A = S\ \texttt{in}\ S') \preceq (\texttt{let}\ B = S_3\ \texttt{in}\ S_4)$*. This is done by treating* B, S_3 *and* S_4 *like constants. The freshness constraint* $B\#S_4$ *is interpreted as a part of the description of the input.*

The matching substitution is $\sigma = \{A \mapsto B; S \mapsto S_3; S' \mapsto S_4\}$*. The freshness constraint* $\{A\#S'\}$ *of the rule is instantiated to* $\{B\#S_4\}$*, which is identical to the input constraint. Thus the result of rewriting is* $(S_4, \{B\#S_4\})$*.*

$$\text{(M1)}\ \frac{(\Gamma \uplus \{e \preceq e\}, \nabla, \theta)}{(\Gamma, \nabla, \theta)} \qquad \text{(M2)}\ \frac{(\Gamma \uplus \{\pi \cdot S \preceq e\}, \nabla, \theta)}{(\Gamma \uplus \{S \preceq \pi^{-1} \cdot e\}, \nabla, \theta)}$$

$$\text{(M3)}\ \frac{(\Gamma \uplus \{S \preceq e\}, \nabla, \theta)}{(\Gamma[e/S], \nabla[e/S], \theta \cup \{S \mapsto e\})} \qquad \text{(M4)}\ \frac{(\Gamma \uplus \{\pi_1 \cdot A \preceq \pi_2 \cdot B\}, \nabla, \theta)}{(\Gamma, \nabla \cup \{A =_\# \pi_1^{-1} \cdot \pi_2 \cdot B\}, \theta)}$$

$$\text{(M5)}\ \frac{(\Gamma \uplus \{(f\ e_1 \dots e_{ar(f)}) \preceq (f\ e'_1 \dots e'_{ar(f)})\}, \nabla, \theta)}{(\Gamma \uplus \{e_1 \preceq e'_1, \dots, e_{ar(f)} \preceq e'_{ar(f)}\}, \nabla, \theta)}$$

$$\text{(M6)}\ \frac{(\Gamma \uplus \{\lambda\pi_1 \cdot A_1.e_1 \preceq \lambda\pi_2 \cdot A_2.e_2\}, \nabla, \theta)}{(\Gamma \uplus \{e_1 \preceq ((\pi_1 \cdot A_1)\ (\pi_2 \cdot A_2)) \cdot e_2\}, \nabla \cup \{(A_1 \# \pi_1^{-1} \cdot (\lambda\pi_2 \cdot A_2.e_2))\}, \theta)}$$

Figure 2: Rules of *NomMatchAS*

2.1 Nominal Match

The goal of matching (in a first approximation) is to find out for given expressions e, e' whether there is a substitution θ such that $e\theta$ represents e'. Below we will use (Δ, e') as targets, i.e., expressions e' that are restricted by a freshness constraint Δ. The semantics of a pair (Δ, s) is a set of ground expressions: $\{s\sigma \mid s\sigma \text{ is ground}, \Delta\sigma \text{ is valid}\}$.

The basic components of a freshness constraint ∇ are single constraints $A\#e$. Certain basic constraints can be written in different notation or more explicitly: (i) $A_1 =_\# \pi \cdot A_2$ means equality and abbreviates $A_1 \# \lambda(\pi \cdot A_2).A_1$; (ii) $A_1 \neq_\# \pi \cdot A_2$ means disequality and is another way to write $A_1 \# \pi \cdot A_2$; and (iii) $A_1 \# \lambda\pi \cdot A_2.e$ could be written as a disjunction $(A_1 =_\# \pi \cdot A_2 \vee A_1 \# e)$ (but we will not do this explicitely).

The rules for computing a match $e \preceq e'$ (ignoring the Δ-constraints) in NL_{AS} are in Fig. 2. The rules operate on a triple (Γ, ∇, θ) of a set of match-equations Γ, freshness constraints ∇ and a substitution θ.

Definition 2.5. *A matcher for a matching problem $(\Delta, e) \preceq (\Delta', e')$ is a tuple (∇, θ) such that:*

- $\nabla \vDash e\theta \sim e'$, *i.e.* $\forall\gamma : \nabla\gamma$ *is valid and* $e\gamma, e'\gamma$ *ground* $\implies e\sigma\gamma \sim e'\gamma$
- *For every ground substitution γ with domain $AtVar(\Delta')$, such that $\gamma \models \Delta'$, there is an extension γ' of γ, such that $\gamma' \models \nabla \cup \Delta\sigma$. This means the following formula must hold: $\forall\mathcal{B}.\exists\mathcal{A}.(\Delta' \implies \nabla \cup \Delta\sigma)$, where $\mathcal{B} = Var(\Delta', e')$ and $\mathcal{A} = Var(\nabla \cup \Delta\sigma) \setminus \mathcal{B}$*

Let $FA(.)$ denote the free atom variables in a constraint or an expression.

Definition 2.6. *Let the input of NomMatchAS be (Δ, e) and (Δ', e'), where $FA(\Delta') \subseteq FA(e')$ must hold,*
The matching algorithm NomMatchAS starts with $\Gamma = (\{e \preceq e'\}, \Delta, \emptyset)$. Then it performs the rules in Fig. 2 until the triple is $(\emptyset, \nabla, \theta)$, i.e. Γ is empty.
If the process gets stuck, then there is no match.
If Γ is empty, then the second matching condition needs to be tested, i.e. the formula $\forall\mathcal{B}.\exists\mathcal{A}.(\Delta' \implies \nabla)$, where $\mathcal{B} = Var(\Delta', e')$ and $\mathcal{A} = Var(\nabla) \setminus \mathcal{B}$, must hold.

The condition $\forall\mathcal{B}.\exists\mathcal{A}.(\Delta' \implies \nabla)$ can be made algorithmic by only looking for equivalence relations on $\mathcal{B}$ (that may be induced by the substitutions γ). I.e. for every equivalence relation $\sim$

on $\mathcal{B}$, let $EQ(\sim)$ be the freshness constraint that exactly describes the equations and disequations (of atom variables) for $\sim$: Then $(\Delta' \cup EQ(\sim)) \implies (\nabla \cup EQ(\sim))$ must be valid, which can be checked in polynomial time.

Proposition 2.7. *The complexity of the final test of the matching algorithm is in* Π_2^P.

Proof. The quantifiers have the effect of adding an equivalence relation on the atom-variables. If the constraint Δ is instantiated with atom variables by a ground substitution σ, then very single freshness constraint $A\#e$ can be decided in polynomial time in the size of the constraint by simply checking $A\sigma\#e\sigma$. □

We are working on determining lower complexity bounds for a single rewriting step. The same techniques as in [14] permit to show:

Theorem 2.8. *NomMatchAS is sound and complete and computes at most one match.*

2.2 Rewriting and Overlap

We define nominal rewriting of expression with atom- and expression-variables on targets $(\Delta, C[s])$ where s is the sub-expression that is to be modified and Δ is a freshness constraint.

Definition 2.9. *Let $(\nabla, l \to r)$ be a rewrite rule and let $(\Delta, C[s])$ be the object to be rewritten, where we assume that $Var(\nabla, l \to r) \cap Var(\Delta, C[s]) = \emptyset$. (The condition can be achieved by a renaming of $\nabla, l \to r$.) A rewrite step is defined as follows:*

Let (∇', σ) be a nominal matcher of $(\nabla, l) \preceq (\Delta, s)$ computed with NomMatchAS and let $\nabla'' = \nabla' \wedge \nabla\sigma$.

Then the result of rewriting is $(\Delta \cup \nabla'', C[r\sigma])$.

Now we define overlap, join and critical pairs and the Knuth Bendix-criterion in our setting.

Definition 2.10. *An* overlap *of two (variable-disjoint) rewrite rules $(\nabla_1, l_1 \to r_1)$ and $(\nabla_2, l_2 \to r_2)$ is computed by the following algorithm. Select a non-variable position p in l_1, represented by a context C, such that $C[l_1'] = l_1$ and the hole of C is at expression-position p. Apply the unification algorithm in [14] to the equation $l_1' \doteq l_2$ and constraint $\nabla_1 \wedge \nabla_2$. If there is an overlap, then the (unique) result of the unification algorithm is a constraint and a substitution (∇', σ), where we assume that $Dom(\sigma) \cap Var(\nabla') = \emptyset$. The resulting overlap expression is $(l_1\sigma, \nabla')$.*

The critical pair *consists of the corresponding rewriting results: $((r_1\sigma, \nabla'), (C\sigma[r_2\sigma], \nabla'))$.*

For the final join, we have to check for the equivalence of targets (Δ_1, e_1) and (Δ_2, e_2). In general this cannot be done by purely syntactic means. A correct method is to compute whether these match each other also respecting all the freshness constraints. This test is decidable.

3 Conclusion

Future work is 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 [13].

A potential application are some reduction rules in the call-by-need calculus of [1] or in CHF [11, 12], like `let` $y = v$ `in` $C[y] \to$ `let` $y = v$ `in` $C[v]$, where v is a value, or similar rules.

Further applications are other higher-order theories of data structures like the monad theory.

References

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

[2] M. Ayala-Rincón, M. Fernández, M. J. Gabbay, and A. C. Rocha-Oliveira. Checking overlaps of nominal rewriting rules. *ENTCS*, 323:39–56, 2016.

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

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

[5] M. Fernández and A. Rubio. Nominal completion for rewrite systems with binders. In A. Czumaj, K. Mehlhorn, A. M. Pitts, and R. Wattenhofer, editors, *Proc. 39th ICALP Part II*, volume 7392 of *LNCS*, pages 201–213. Springer, 2012.

[6] M. 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.

[7] 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.

[8] J. Levy and M. Villaret. An efficient nominal unification algorithm. In C. Lynch, editor, *Proc. 21st RTA*, volume 6 of *LIPIcs*, pages 209–226. Schloss Dagstuhl, 2010.

[9] A. Pitts. Nominal techniques. *ACM SIGLOG News*, 3(1):57–72, Feb. 2016.

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

[11] D. Sabel and M. Schmidt-Schauß. A contextual semantics for concurrent Haskell with futures. In P. Schneider-Kamp and M. Hanus, editors, *Proc. 13th ACM PPDP 2011*, pages 101–112. ACM, 2011.

[12] M. Schmidt-Schauß and N. Dallmeyer. Space improvements and equivalences in a functional core language. *CoRR*, abs/1802.06498, 2018.

[13] M. Schmidt-Schauß and D. Sabel. Nominal unification with atom and context variables. In H. Kirchner, editor, *Proc. 3rd FSCD 2018)*. Schloss Dagstuhl, 2018. accepted for publication.

[14] M. Schmidt-Schauß, D. Sabel, and Y. Kutz. Nominal unification with atom-variables. *Journal of Symbolic Computation*, 2018. article in press.

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

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

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

[18] P. Wadler. Monads for functional programming. In J. Jeuring and E. 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.