

Nominal Matching and Alpha-Equivalence* (Extended Abstract)

Christophe Calvès and Maribel Fernández

King's College London, Department of Computer Science,
Strand, London WC2R 2LS, UK

`Christophe.Calves@kcl.ac.uk` `Maribel.Fernandez@kcl.ac.uk`

Abstract. Nominal techniques were introduced to represent in a simple and natural way systems that involve binders. The syntax includes an abstraction operator and a primitive notion of name swapping. Nominal matching is matching modulo α -equality, and has applications in programming languages and theorem proving, amongst others. In this paper we describe efficient algorithms to check the validity of equations involving binders, and also to solve matching problems modulo α -equivalence, using the nominal approach.

Keywords: Binders, α -equivalence, matching, nominal terms.

1 Introduction

The notion of a binder is ubiquitous in computer science. Programs, logic formulas, and process calculi, are some examples of systems that involve binding. Program transformations and optimisations, for instance, are defined as operations on programs, and therefore work uniformly on α -equivalence classes. To formally define a transformation rule acting on programs, we need to be able to distinguish between free and bound variables, and between meta-variables of the transformation rule and variables of the object language. We also need to be able to test for α -equivalence, and we need a notion of matching that takes into account α -equivalence.

Nominal techniques were introduced to represent in a simple and natural way systems that include binders [7,10,11]. The nominal approach to the representation of systems with binders is characterised by the distinction, at the syntactical level, between *atoms* (or object-level variables), which can be abstracted (we use the notation $[a]t$, where a is an atom and t is a term), and *meta-variables* (or just variables), which behave like first-order variables but may be decorated with atom permutations. Permutations are generated using swappings (e.g. $(a\ b) \cdot t$ means swap a and b everywhere in t). For instance, $(a\ b) \cdot \lambda[a]a = \lambda[b]b$, and $(a\ b) \cdot \lambda[a]X = \lambda[b](a\ b) \cdot X$ (we will introduce the notation formally in the next section). As shown in this example, permutations suspend on variables. The idea is that when a substitution is applied to X in $(a\ b) \cdot X$, the permutation will be

* This work has been partially funded by an EPSRC grant (EP/D501016/1 “CANS”).

applied to the term that instantiates X . Permutations of atoms are one of the main ingredients in the definition of α -equivalence for nominal terms.

Nominal terms [12] can be seen as trees, built from function symbols, tuples and abstraction term-constructors; atoms and variables are leaves. We can define by induction a *freshness relation* $a\#t$ (read “the atom a is fresh for the term t ”) which roughly corresponds to the notion of a not occurring unabstracted in t . Using freshness and swappings we can inductively define a notion of α -equivalence of terms. Nominal unification is the problem of deciding whether two nominal terms can be made α -equivalent by instantiating their variables. It is a generalisation of the unification problem for first-order terms [1], and has the same applications in rewriting [5], logic programming [3], etc. Urban, Pitts and Gabbay [12] showed that nominal unification is decidable, and gave an algorithm to find the most general solution to a nominal unification problem, if one exists.

In this paper we study a simpler version of the problem —nominal matching— that has applications in functional programming, rewriting and meta programming amongst others. In a matching problem $s \approx_\alpha t$, t is *ground* (i.e., it has no variables), or, more generally, it has variables that cannot be instantiated.¹ When the term t is ground we say that the matching problem is *ground*. The left-hand side of a matching problem $s \approx_\alpha t$ is called a *pattern*, and may have variables occurring multiple times. When each variable occurs at most once in patterns we say that the matching problem is *linear*. We present an efficient algorithm that can be used to solve both linear and non-linear matching problems modulo α , as well as ground and non-ground problems. An algorithm to test α -equivalence of nominal terms (ground or non-ground) can be easily derived.

The complexity of the algorithms depends on the kind of problem to be solved; it is given in the table below:

Case	Alpha-equivalence	Matching
Ground	linear	linear
Non-ground and linear	log-linear	log-linear
Non-ground and non-linear	log-linear	quadratic

We have implemented the algorithms using OCAML [9], the implementation is available from: www.dcs.kcl.ac.uk/staff/maribel/CANS. We give sample benchmarks in the Appendix (Section A), for more details see the website above.

In functional programming applications, matching problems are ground and in this case our algorithm is linear in time, as indicated in the first line of the table above. To our knowledge, this is the only available nominal matching algorithm with this complexity.

We are currently deploying the algorithms in a rewriting tool that can be used to specify equational theories including binders in the nominal style (see [6,4]), and to evaluate functions working on data structures that include binding. In future, we hope to be able to extend the implementation techniques discussed in this paper to solve nominal unification problems. The complexity of nominal unification is still an open problem.

¹ These variables may have suspended permutations.

2 Background

Let Σ be a denumerable set of **function symbols** f, g, \dots ; \mathcal{X} be a denumerable set of **variables** X, Y, \dots (representing meta-level variables); and \mathcal{A} be a denumerable set of **atoms** a, b, \dots (representing object-level variables). We assume that Σ , \mathcal{X} and \mathcal{A} are pairwise disjoint. A **swapping** is a pair of (not necessarily distinct) atoms, written $(a\ b)$. **Permutations** π are lists of swappings, generated by the grammar: $\pi ::= \text{ld} \mid (a\ b) \circ \pi$. We call **ld** the **identity permutation** and write π^{-1} for the permutation obtained by reversing the list of swappings in π . We denote by $\pi \circ \pi'$ the permutation containing all the swappings in π followed by those in π' . A pair $\pi \cdot X$ of a permutation π and a variable X is called a **suspension**.

Nominal terms, or just **terms** for short, over $\Sigma, \mathcal{X}, \mathcal{A}$ are generated by the grammar: $s, t ::= a \mid \pi \cdot X \mid (s_1, \dots, s_n) \mid [a]s \mid f\ t$

A term is **ground** if it has no variables; $V(t)$ denotes the set of elements of \mathcal{X} that occur in t . We refer the reader to [12,5] for more details and examples of nominal terms.

We can apply permutations and substitutions on terms, denoted $\pi \cdot t$ and $t[X \mapsto s]$ respectively. Permutations act top-down and accumulate on variables whereas substitutions act on variables. More precisely, $\pi \cdot t$ is defined by induction: $\text{ld} \cdot t = t$ and $((a\ b) \circ \pi) \cdot t = (a\ b) \cdot (\pi \cdot t)$, where

$$\begin{aligned} (a\ b) \cdot a &= b & (a\ b) \cdot b &= a & (a\ b) \cdot c &= c \text{ if } c \notin \{a, b\} \\ (a\ b) \cdot (\pi \cdot X) &= ((a\ b) \circ \pi) \cdot X & (a\ b) \cdot (f\ t) &= f(a\ b) \cdot t \\ (a\ b) \cdot [n]t &= [(a\ b) \cdot n](a\ b) \cdot t & (a\ b) \cdot (t_1, \dots, t_n) &= ((a\ b) \cdot t_1, \dots, (a\ b) \cdot t_n) \end{aligned}$$

In the sequel we abbreviate $\text{ld} \cdot X$ as X when there is no ambiguity.

A **substitution** is generated by the grammar: $\sigma ::= \text{ld} \mid [X \mapsto s]\sigma$. We write substitutions postfix and write \circ for composition of substitutions: $t(\sigma \circ \sigma') = (t\sigma)\sigma'$. We define the instantiation of a term t by a substitution σ by induction: $t\ \text{ld} = t$ and $t[X \mapsto s]\sigma = (t[X \mapsto s])\sigma$ where

$$\begin{aligned} a[X \mapsto s] &= a & (t_1, \dots, t_n)[X \mapsto s] &= (t_1[X \mapsto s], \dots, t_n[X \mapsto s]) \\ ([a]t)[X \mapsto s] &= [a]t[X \mapsto s] & (ft)[X \mapsto s] &= f(t[X \mapsto s]) \\ (\pi \cdot X)[X \mapsto s] &= \pi \cdot s & (\pi \cdot Y)[X \mapsto s] &= \pi \cdot Y \end{aligned}$$

Constraints have the form: $a\#t$ or $s \approx_\alpha t$, where $\#$ is the **freshness** relation between atoms and terms, and \approx_α denotes **alpha-equality**. A set Pr of constraints is called a **problem**. Intuitively, $a\#t$ means that if a occurs in t then it must do so under an abstractor $[a]$. For example, $a\#b$, and $a\#[a]a$ but not $a\#a$. We sometimes write $a, b\#s$ instead of $a\#s, b\#s$, or write $A\#s$, where A is a set of atoms, to mean that all atoms in A are fresh for s .

The following set of simplification rules from [12], acting on problems, can be used to *check* the validity of α -equality constraints (below $ds(\pi, \pi')$ is an abbreviation for $\{n \mid \pi \cdot n \neq \pi' \cdot n\}$).

$$\begin{aligned}
& a\#b, Pr \implies Pr \\
& a\#fs, Pr \implies a\#s, Pr \\
& a\#(s_1, \dots, s_n), Pr \implies a\#s_1, \dots, a\#s_n, Pr \\
& a\#[b]s, Pr \implies a\#s, Pr \\
& a\#[a]s, Pr \implies Pr \\
& a\#\pi \cdot X, Pr \implies \pi^{-1} \cdot a\#X, Pr \quad \pi \neq \text{ld} \\
& a \approx_\alpha a, Pr \implies Pr \\
& (l_1, \dots, l_n) \approx_\alpha (s_1, \dots, s_n), Pr \implies l_1 \approx_\alpha s_1, \dots, l_n \approx_\alpha s_n, Pr \\
& fl \approx_\alpha fs, Pr \implies l \approx_\alpha s, Pr \\
& [a]l \approx_\alpha [a]s, Pr \implies l \approx_\alpha s, Pr \\
& [a]l \approx_\alpha [b]s, Pr \implies l \approx_\alpha (a \ b) \cdot s, a\#s, Pr \\
& \pi \cdot X \approx_\alpha \pi' \cdot X, Pr \implies ds(\pi, \pi')\#X, Pr
\end{aligned}$$

Given a problem Pr , we apply the rules until we get an irreducible problem, i.e. a **normal form**. If only a set Δ of constraints of the form $a\#X$ are left, then the original problem is **valid** in the context Δ (i.e., $\Delta \vdash Pr$), otherwise it is **not valid**. Thus, a problem such as $X \approx_\alpha a$ is not valid, since it is irreducible. However, X can be made equal to a by *instantiation* (i.e., applying a substitution); we say that this constraint can be **solved**. If we impose the restriction that in a constraint $s \approx_\alpha t$ the variables in t cannot be instantiated and the variables in left-hand sides are disjoint from the variables in right-hand sides, then we obtain a nominal **matching** problem. If we require s to be linear (i.e., each variable occurs at most once in s), we obtain a **linear** nominal matching problem.

A most general **solution** to a nominal matching problem Pr is a pair (Δ, σ) of a freshness context and a substitution, obtained from the simplification rules above, enriched with an **instantiating** rule labelled with substitutions:

$$\pi \cdot X \approx_\alpha u, Pr \xrightarrow{X \mapsto \pi^{-1} \cdot u} Pr[X \mapsto \pi^{-1} \cdot u]$$

Note that there is no need to do an occur-check because left-hand side variables are distinct from right-hand side variables in a matching problem.

3 The algorithm

The transformation rules given in Section 2 create permutations. Polynomial implementations of the nominal unification algorithm [2] rely on the use of **lazy permutations**: permutations are only pushed down a term when this is needed to apply a transformation rule. We will use this idea, but, since lazy permutations may grow (they accumulate), in order to obtain an efficient algorithm we will devise a mechanism to compose the swappings eagerly. The key idea is to work with a single *current* permutation, represented by an **environment**.

Definition 1. Let s and t be terms, π be a permutation and A be a finite set of atoms. An **environment** ξ is a pair (π, A) . We denote by ξ_π the permutation (resp. ξ_A the set of atoms) of an environment. We write $s \approx_\alpha \xi \diamond t$ to represent $s \approx_\alpha \xi_\pi \cdot t$, $\xi_A \# t$, and call $s \approx_\alpha \xi \diamond t$ an **environment constraint**.

Definition 2. An *environment problem* Pr is either \perp or has the form $s_1 \approx_\alpha \xi_1 \diamond t_1, \dots, s_n \approx_\alpha \xi_n \diamond t_n$, where $s_i \approx_\alpha \xi_i \diamond t_i$ ($1 \leq i \leq n$) are environment constraints. We will sometimes abbreviate it as $(s_i \approx_\alpha \xi_i \diamond t_i)_1^n$.

The problems defined in Section 2 will be called **standard** to distinguish them from environment problems (standard problems have no environments). The **standard form** of an environment problem is obtained by applying as many times as possible the rule: $s \approx_\alpha \xi \diamond t \implies s \approx_\alpha \xi_\pi \cdot t, \xi_A \# t$. We denote by $\llbracket Pr \rrbracket$ the standard form of an environment problem Pr .

This rule is terminating because it consumes a \diamond each time, without creating any. There is no critical pair so the system is locally confluent and because it terminates it is confluent [8]. Therefore the standard form of an environment problem exists and is unique, justifying the notation $\llbracket Pr \rrbracket$.

The **solutions** of an environment problem are the solutions of its standard form (see Section 2). A problem \perp has no solutions. Two environment problems are **equivalent** if their standard forms are equivalent, i.e., have the same solutions.

Standard problems are translated into environment problems in linear time: $s \approx_\alpha t$ is encoded as $s \approx_\alpha \xi \diamond t$ where $\xi = (\text{Id}, \emptyset)$ and $A \# t$ is encoded as $t \approx_\alpha \xi \diamond t$ where $\xi = (\text{Id}, A)$. In the sequel we restrict our attention to checking α -equivalence constraints and solving matching problems. In the latter case, in environment constraints $s \approx_\alpha \xi \diamond t$, the term t will not be instantiated and variables in s and t are disjoint. If right-hand sides t are ground terms, we will say that the problem is *ground*, and *non-ground* otherwise.

3.1 Core algorithm

The algorithms to check α -equivalence constraints and to solve matching problems will be built in a modular way. The core module is common to both algorithms; only the final phase will be specific to matching or α -equivalence. There are four phases in the core algorithm. We denote by \overline{Pr}^c the result of applying the core algorithm on Pr .

Phase 1. The input is an environment problem $Pr = (s_i \approx_\alpha \xi_i \diamond t_i)_i^n$, that we reduce using the following transformation rules, where a, b could be the same atom and in the last rule $\xi' = ((a \xi_\pi \cdot b) \circ \xi_\pi, (\xi_A \cup \{\xi_\pi^{-1} \cdot a\}) \setminus \{b\})$.

$$\begin{array}{l}
Pr, \quad a \quad \approx_\alpha \xi \diamond t \implies \begin{cases} Pr & \text{if } a = \xi_\pi \cdot t \text{ and } t \notin \xi_A \\ \perp & \text{otherwise} \end{cases} \\
Pr, (s_1, \dots, s_n) \approx_\alpha \xi \diamond t \implies \begin{cases} Pr, (s_i \approx_\alpha \xi \diamond u_i)_1^n & \text{if } t = (u_1, \dots, u_n) \\ \perp & \text{otherwise} \end{cases} \\
Pr, \quad f s \quad \approx_\alpha \xi \diamond t \implies \begin{cases} Pr, s \approx_\alpha \xi \diamond u & \text{if } t = f u \\ \perp & \text{otherwise} \end{cases} \\
Pr, \quad [a]s \quad \approx_\alpha \xi \diamond t \implies \begin{cases} Pr, s \approx_\alpha \xi' \diamond u & \text{if } t = [b]u \\ \perp & \text{otherwise} \end{cases}
\end{array}$$

The environment problems that are irreducible for the rules above will be called **phase 1 normal forms** or **ph1nf** for short.

Proposition 1 (Phase 1 Normal Forms). *The normal forms for phase 1 rules are either \perp or $(\pi_i \cdot X_i \approx_\alpha \xi_i \diamond s_i)_1^n$ where s_i are nominal terms.*

Phase 2. This phase takes as input an environment problem in ph1nf, and moves the permutations to the right-hand side. More precisely, given a problem in ph1nf, we apply the rule:

$$\pi \cdot X \approx_\alpha \xi \diamond t \implies X \approx_\alpha (\pi^{-1} \cdot \xi) \diamond t \quad (\pi \neq \text{ld})$$

where $\pi^{-1} \cdot \xi = (\pi^{-1} \circ \xi_\pi, \xi_A)$. Note that π^{-1} applies only to ξ_π here, because $\pi \cdot X \approx_\alpha \xi \diamond t$ represents $\pi \cdot X \approx_\alpha \xi_\pi \cdot t, \xi_A \# t$.

If the problem is irreducible (i.e., it is a normal form for the rule above), we say it is a **phase 2 normal form**, or **ph2nf** for short.

Proposition 2 (Phase 2 Normal Forms). *Given a ph1nf problem, it has a unique normal form for the rule above, and it is either \perp or a problem of the form $(X_i \approx_\alpha \xi_i \diamond t_i)_1^n$, where the terms t_i are standard nominal terms.*

Phase 3. In the phases 1 and 2 we deal with \approx_α constraints. Phase 3 takes a ph2nf and simplifies freshness constraints, by propagating environments over terms. Since the input is a problem in ph2nf, each constraint has the form $X \approx_\alpha \xi \diamond t$. We reduce it with the following rewrite rules, which propagate ξ over t and deal with problems containing \perp (denoted $Pr[\perp]$):

$$\begin{aligned} \xi \diamond a &\implies \begin{cases} \xi_\pi \cdot a & a \notin \xi_A \\ \perp & a \in \xi_A \end{cases} \\ \xi \diamond f t &\implies f (\xi \diamond t) \\ \xi \diamond (t_1, \dots, t_j) &\implies (\xi \diamond t_i)_1^j \\ \xi \diamond [a]s &\implies [\xi_\pi \cdot a]((\xi \setminus \{a\}) \diamond s) \\ \xi \diamond (\pi \cdot X) &\implies (\xi \circ \pi) \diamond X \\ Pr[\perp] &\implies \perp \end{aligned}$$

where $\xi \setminus \{a\} = (\xi_\pi, \xi_A \setminus \{a\})$ and $\xi \circ \pi = ((\xi_\pi \circ \pi), \pi^{-1}(\xi_A))$.

These rules move environments inside terms, so formally we need to extend the definition of nominal term, to allow us to attach an environment at any position inside the term. We omit the definition of terms with suspended environments, and give just the grammar for the normal forms, which may have environments suspended only on variable leaves:

Definition 3. *The language of normal environment terms is defined by:*

$$T_\xi = a \mid f T_\xi \mid (T_\xi, \dots, T_\xi) \mid [a]T_\xi \mid \xi \diamond X$$

Proposition 3 (Phase 3 Normal Forms - ph3nf). *The normal forms for this phase are either \perp or $(X_i \approx_\alpha t_i)_1^n$ where $t_i \in T_\xi$.*

Phase 4. This phase computes the standard form of a ph3nf:

$$X \approx_\alpha C[\xi \diamond X'] \implies X \approx_\alpha C[\xi_\pi \cdot X'] , \xi_A \# X'$$

Proposition 4 (Phase 4 Normal Forms - ph4nf). *If we normalise a ph3nf using the rule above, we obtain either \perp or $(X_i \approx_\alpha u_i)_{i \in I}, (A_j \# X_j)_{j \in J}$ where u_i are nominal terms and I, J may be empty.*

The core algorithm terminates, and preserves the set of solutions. Since all the reduction rules, except the rule dealing with \perp , are local (i.e. only modify one constraint), the result of applying the core algorithm to a set of constraints is the union of the results obtained for each individual constraint (assuming $\perp, Pr \equiv \perp$).

3.2 Checking the validity of α -equivalence constraints

To check that a set Pr of α -equivalence constraints is valid, we first run the core algorithm on Pr and then reduce the result \overline{Pr}^c by the following rule:

$$(\alpha) \quad Pr , X \approx_\alpha t \implies \begin{cases} Pr , \text{supp}(\pi) \# X & \text{if } t = \pi \cdot X \\ \perp & \text{otherwise} \end{cases}$$

where $\text{supp}(\pi)$ is the **support** of π : $\text{supp}(\pi) = \{a \mid \pi \cdot a \neq a\}$.

Since this rule is terminating (each application consumes one \approx_α -constraint) and there are no critical pairs, it is confluent [8], therefore normal forms exist and are unique. $\overline{Pr}^{\approx_\alpha}$ denotes the normal form of \overline{Pr}^c by the rule above.

Proposition 5 (Normal Forms $\overline{Pr}^{\approx_\alpha}$). *$\overline{Pr}^{\approx_\alpha}$ is either \perp or $(A_i \# X_i)_1^n$.*

Proposition 6 (Correctness). *If $\overline{Pr}^{\approx_\alpha}$ is \perp then Pr is not valid. If $\overline{Pr}^{\approx_\alpha}$ is $(A_i \# X_i)_1^n$ then $\overline{Pr}^{\approx_\alpha} \vdash Pr$.*

If the left-hand sides of \approx_α -constraints in Pr are ground, then $\overline{Pr}^c = \overline{Pr}^{\approx_\alpha}$; rule (α) is not necessary in this case.

3.3 Solving Matching Problems

To solve a matching problem Pr , we first run the core algorithm on Pr and then if the problem is non-linear we normalise the result \overline{Pr}^c by the following rule:

$$(? \approx) \quad Pr , X \approx_\alpha s , X \approx_\alpha t \implies \begin{cases} Pr , X \approx_\alpha s , \overline{s \approx_\alpha t}^{\approx_\alpha} & \text{if } \overline{s \approx_\alpha t}^{\approx_\alpha} \neq \perp \\ \perp & \text{otherwise} \end{cases}$$

This rule is terminating: each reduction consumes at least one \approx_α -constraint, and the algorithm computing $\overline{Pr}^{\approx_\alpha}$ is also terminating. $\overline{Pr}^{? \approx}$ denotes a normal form of \overline{Pr}^c by the rule $(? \approx)$.

Proposition 7 (Normal Forms $\overline{Pr} \stackrel{?}{\approx}$). *If we normalise \overline{Pr}^c using the rule above, we obtain either \perp or $(X_i \approx_\alpha s_i)_1^n$, $(A_i \# X_i)_1^m$ where s_i are standard terms, all X_i in the equations $(X_i \approx_\alpha s_i)_1^n$ are different variables and $\forall i, j : X_i \notin V(s_j)$.*

Hence, a normal form $\overline{Pr} \stackrel{?}{\approx}$ is either \perp , or the coding of an idempotent substitution σ and a freshness context Δ .

Proposition 8 (Correctness). *$\overline{Pr} \stackrel{?}{\approx}$ is a most general solution of the matching problem Pr .*

4 Implementation

Coding the problem. Terms are represented as trees. We code $\xi \diamond t$ by attaching ξ to the root of t . Environment constraints are represented as pairs of trees, and freshness constraints as a pair of a set of atoms and a variable. Problems are represented as lists of constraints.

Avoiding environment creation in the core algorithm. Instead of running each phase in turn, we combine them to have a **local reduction** strategy: we fully reduce one constraint into ph4nf before reducing other constraints.

Each rule in the algorithm involves at most one environment, obtained either by copying or modifying another one. Instead of copying environments (in the case of tuples), we will share them. Updates in the *current* environment will, because of sharing, affect all the constraints where this environment is used. However, thanks to our local reduction strategy, none of these constraints will be reduced until the current constraint is in ph4nf (and then it will not use any environment). At this point, by reversing the environment to its original state, we can safely reduce the other constraints. Therefore, we keep track of the operations we made in the environment, fully reduce the current constraint, and then reverse the operations before reducing another constraint.

Permutations and sets. We code atoms as integers, and permutations (resp. sets) as *mutable arrays* or as *functional maps* of atoms (resp. booleans) indexed by atoms such that the value at the index a is the image of a by the permutation (resp. the boolean indicating whether a is in the set or not).

On one hand, mutable arrays have the advantage that they can be accessed and updated in constant time, but are destructive so we may need to copy them sometimes (an operation that is linear in time and space in their size). On the other hand, an access/update on functional maps is logarithmic in time, but since updates are not destructive we do not need to copy them. We will use either mutable arrays or functional maps depending on the kind of problem:

Case	Alpha-equivalence	Matching
Ground	mutable arrays	mutable arrays
Non-ground and linear	functional maps	functional maps
Non-ground and non-linear	functional maps	mutable arrays

Note that when the problem is ground, phase 4 is not needed in the core algorithm, and therefore we never need to display permutations or freshness constraints. Since in this case we only need to access and update the environment, arrays are more efficient. With linear, non-ground problems, we need phase 4, and the cost is quadratic using arrays, but log-linear using functional maps. We will discuss the non-linear case in Section 5.

Since we often need the inverse and the support of a permutation, when we create a permutation we compute at the same time its inverse and its support and keep them in the same tuple. This can be done in linear time with arrays and in log-linear time with maps.

Implementing the algorithms. The implementation of the core algorithm is essentially a traversal of the data structure representing the input problem Pr_0 , propagating the environment using the techniques above. The result is a list $\overline{Pr_0}^c$ of constraints in ph4nf. The α -equivalence algorithm then takes each \approx_α -constraint in the list $\overline{Pr_0}^c$ and reduces it with (α) . The matching algorithm applies the rule (\approx) : it traverses the list to take for each variable X the constraint $X \approx_\alpha s$ with minimal s (we define the size of s below), and store $S[X] = s$ in an array S indexed by variables. Then the algorithm traverses the list again applying the rule *Rl-Check-Subst* where $\overline{S[X] \approx_\alpha t}^{\approx_\alpha}$ is computed using arrays.

$$Pr, X \approx_\alpha t \implies \begin{cases} Pr, \overline{S[X] \approx_\alpha t}^{\approx_\alpha} & \text{if } \overline{S[X] \approx_\alpha t}^{\approx_\alpha} \neq \perp \\ \perp & \text{otherwise} \end{cases}$$

5 Complexity

Atoms are coded as integers, as explained above. Let M_{A_0} be the maximum atom in A_0 (the set of atoms occurring in the input problem Pr_0). Let $|t|_n$ be the number of nodes in the tree representing t . Finally, let $MV(t)$ be the multiset of the occurrences of variables in t .

Core algorithm. Below we analyse the cost of the algorithm.

Definition 4. *The size of the problem $s \approx_\alpha \xi \diamond t$, written $|s \approx_\alpha \xi \diamond t|$, is defined as $|s \approx_\alpha \xi \diamond t| = |s| + |\xi| + |t|$ where $|\xi| = 2 \times |\xi_\pi| + |\xi_A|$, $|\xi_\pi|$ (resp. $|\xi_A|$) is the size of the array/map representing it, and $|t|$ is defined by: $|a| = |X| = 1$, $|f t| = 1 + |t|$, $|(t_1, \dots, t_j)| = 1 + |t_1| + \dots + |t_j|$, $|[a]s| = 1 + |s|$ and $|\pi \cdot X| = 1 + |\pi|$.*

Proposition 9. *The core algorithm is linear in the size of the initial problem Pr_0 in the ground case, using mutable arrays. In the non-ground case, it is log-linear using functional maps and $\vartheta(|s \approx_\alpha t| + |M_{A_0}| \times |t|_n)$ using mutable arrays.*

The idea of the proof is that the core algorithm is essentially a traversal of the data structure representing the problem. Phases 1 to 3 are trivially linear with arrays and log-linear with functional maps. Phase 4 is done in $\vartheta(|M_{A_0}|)$ with functional maps, and $\vartheta(|M_{A_0}| \times |t|_n)$ with arrays.

Alpha-equivalence. To check the validity of an \approx_α -constraint, after running the core algorithm we have to normalise the problem using the rule (α) , as described in Section 3.2. If the right-hand sides of \approx_α -constraints are ground, the core algorithm is sufficient and it is linear. Otherwise, each application of the rule (α) requires to know the support of a permutation, which we do because supports are always created with permutations and maintained when they are updated. Thanks to the use of functional maps, the support is copied in constant time when the permutation is copied, therefore the algorithm is also log-linear in the size of the problem in the non-ground case.

Matching. The algorithm to solve matching constraints consists of the core algorithm, followed by a normalisation phase (using rule \approx , see Section 3.3) that deals with variables occurring multiple times in the pattern. In the case of linear matching this is not needed – the core algorithm is sufficient.

In Section 4 we discussed the implementation of the rule \approx using an array S indexed by variables and the rule *Rl-Check-Subst*. The construction of S requires the traversal of subterms of the term s and every term in the output of the core algorithm. This is done in time proportional to the size of the output of the core algorithm. At worst, the size is $|M_{A_0}| \times MV(t) + |s \approx_\alpha t|$ because phase 4 can add a suspended permutation and freshness constraints on every variable occurring in t . Therefore the output can be quadratic in the size of the input.

Then *Rl-Check-Subst* will compute $\overline{S[X_i]} \approx_\alpha t_i \approx_\alpha t_i$ for each constraint $X_i \approx_\alpha t_i$ in the result of the core algorithm. Phase 1 to 3 are linear in its size and phase 4 has a complexity $\mathcal{O}(|M_{A_0}| \times MV(t_i))$, hence at worst quadratic in time in the size of the input problem. The worst case complexity arises when phase 4 suspends permutations on all variables. On the other hand, if the input problem already has in each variable a permutation of size $|M_{A_0}|$ (i.e. variables are ‘saturated’ with permutations), then, since permutations cannot grow, the α -equivalence and matching algorithms are linear even using arrays. Note that the representation of a matching problem or an α -equivalence problem using higher-order abstract syntax does saturate the variables (they have to be applied to the set of atoms they can capture). The table below summarises the results:

Case	Alpha-equivalence	Matching
Ground	linear	linear
Non-ground and linear	log-linear	log-linear
Non-ground and non-linear	log-linear	quadratic

6 Conclusions

We described an algorithm to solve matching problems modulo α -equivalence which is linear time and space when the right-hand side terms are ground. Matching modulo α -equivalence has numerous applications, in particular, in the design of functional programming languages that provide constructs for declaring and manipulating abstract syntax trees involving names and binders, and as a basis for the implementation of nominal rewriting tools.

A Appendix: Benchmarks

The algorithms described above to check α -equivalence and to solve ground matching problems have been implemented in OCAML [9], using arrays. In Figure 1, we show benchmarks generated by building random solvable ground problems (i.e., problems that do not lead to \perp) and measuring the time taken by the α -equivalence and matching algorithms to give the result (marked as \diamond and $+$ in the graph)².

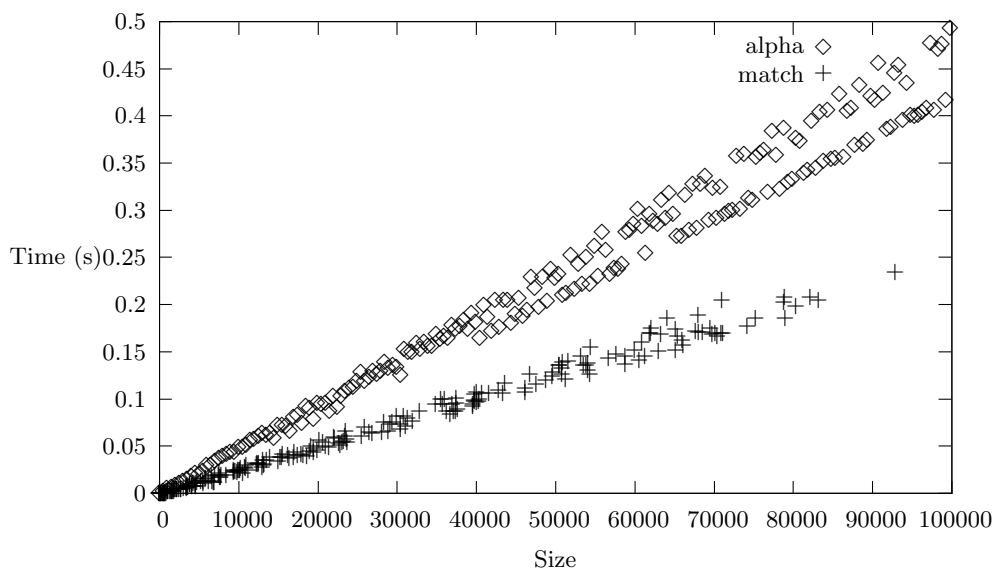


Fig. 1. Benchmarks

Acknowledgements We thank James Cheney, Jamie Gabbay, Andrew Pitts, François Pottier and Christian Urban for useful discussions on the topics of this paper.

References

1. F. Baader and W. Snyder. Unification Theory. *Handbook of Automated Reasoning*, volume I, chapter 8, 445–532. A. Robinson and A. Voronkov (eds.), Elsevier Science, 2001.
2. C. Calvès and M. Fernández. Implementing nominal unification. In *Proceedings of TERMGRAPH'06, 3rd International Workshop on Term Graph Rewriting, ETAPS 2006, Vienna*, Electronic Notes in Computer Science. Elsevier, 2006.

² The program is available from: www.dcs.kcl.ac.uk/staff/maribel/CANS

3. J. Cheney. α -Prolog: an interpreter for a logic programming language based on nominal logic. Software available from <http://homepages.inf.ed.ac.uk/jcheney/programs/aprolog/>
4. R. A. Clouston and A. M. Pitts. Nominal Equational Logic. In *Computation, Meaning and Logic. Articles dedicated to Gordon Plotkin*. L. Cardelli, M. Fiore and G. Winskel (eds.), volume 1496, *Electronic Notes in Theoretical Computer Science*, Elsevier, 2007.
5. M. Fernández and M. J. Gabbay. Nominal Rewriting. *Information and Computation* 205:917–965, 2007. Elsevier.
6. M. J. Gabbay and A. Mathijssen. Nominal Algebra. *Proceedings of the 18th Nordic Workshop on Programming Theory (NWPT'06)*, 2006.
7. M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2001.
8. M.H.A. Newman. On theories with a combinatorial definition of equivalence. *Annals of Mathematics*, 43(2):223–243, 1942.
9. OCAML, <http://caml.inria.fr/>
10. A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.
11. M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. In *Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP 2003)*, Sweden, pages 263–274. ACM Press, 2003.
12. C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. *Theoretical Computer Science*, 323:473 – 497, Elsevier, 2004.