Matching and alpha-equivalence check for nominal terms

Christophe Calvès, Maribel Fernández*

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

A R T I C L E   I N F O

Article history:
Received 31 October 2008
Received in revised form 8 March 2009
Available online 27 October 2009

Keywords:
Binders
Alpha-equivalence
Matching
Nominal terms

A B S T R A C T

Nominal terms generalise first-order terms by including abstraction and name swapping
constructs. α-equivalence can be easily axiomatised using name swappings and a freshness
relation, which makes the nominal approach well adapted to the specification of systems
that involve binders. Nominal matching is matching modulo α-equivalence and has
applications in programming languages, rewriting, and theorem proving. In this paper, we
describe efficient algorithms to check the validity of equations involving binders and to
solve matching problems modulo α-equivalence, using the nominal approach.

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 transfor-
mation 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 [11,19,25].
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) · t means swap a and b everywhere in t). For instance,
(a b) · λ[a]a = λ[b]b, and (a b) · λ[a]X = λ[b](a b) · 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) · X, the
permutation will be 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 [26] can be seen as trees, built from function symbols, tuples, and abstractions; atoms and variables
are leaves. On nominal terms, 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. Since t may contain variables, in order to deduce a # t we
might need to use assumptions a # X, so freshness is derived in a context of freshness assumptions: we write freshness
judgements as Δ ′ ⊢ a # t. The α-equivalence relation is also defined with respect to a freshness context. For instance, we can deduce

* Corresponding author.
E-mail address: Maribel.Fernandez@kcl.ac.uk (M. Fernández).

© 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.jcss.2009.10.003
Nominal unification is the problem of deciding whether two nominal terms can be made \( \alpha \)-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 \([9]\) and programming \(\text{see for example } [4]\). Urban, Pitts and Gabbay \([26]\) showed that nominal unification is decidable, and gave an algorithm to find the most general solution to a nominal matching or unification problem, if one exists. Implementations of this algorithm are available, for instance, nominal unification is used in \(\alpha\)-Prolog \([4]\). A naive implementation, representing terms as trees, is exponential. A polynomial implementation using graphs is described in \([3]\).

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. Various versions of matching modulo \(\alpha\)-equivalence are used in functional programming languages that provide constructs for declaring and manipulating abstract syntax trees involving names and binders (e.g., FreshML \([20,25,22]\) and CoML \([21]\), see also the design of ML\(\alpha\) \([15]\)).

In a matching problem \( s \approx_\alpha t \) 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 \(\alpha\), as well as ground and non-ground problems. We also show that an algorithm to test \(\alpha\)-equivalence of nominal terms (ground or non-ground) can be easily derived.

A naive approach to the implementation of matching modulo \(\alpha\) gives a quadratic algorithm. This is because to match two terms involving binders we may need to rename bound variables on-the-fly. For instance, in \(\lambda x. (\lambda x.x) x\) we need to distinguish between the different occurrences of \( x \) in the body. A naive translation to De Bruijn notation (or to a term satisfying Barendregt’s “hygienic” naming convention, where one uses different names for each bound variable) has a similar cost. To obtain efficient algorithms, we use nominal techniques relying on swappings instead of renamings. Our algorithms are based on two main ideas. The first one is that in a nominal matching problem \( s \approx_\alpha t \) 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 \(\alpha\), as well as ground and non-ground problems. We also show that an algorithm to test \(\alpha\)-equivalence of nominal terms (ground or non-ground) can be easily derived.

A naive approach to the implementation of matching modulo \(\alpha\) gives a quadratic algorithm. This is because to match two terms involving binders we may need to rename bound variables on-the-fly. For instance, in \(\lambda x. (\lambda x.x) x\) we need to distinguish between the different occurrences of \( x \) in the body. A naive translation to De Bruijn notation (or to a term satisfying Barendregt’s “hygienic” naming convention, where one uses different names for each bound variable) has a similar cost. To obtain efficient algorithms, we use nominal techniques relying on swappings instead of renamings. Our algorithms are based on two main ideas. The first one is that in a nominal matching problem \( s \approx_\alpha t \) 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 \(\alpha\), as well as ground and non-ground problems. We also show that an algorithm to test \(\alpha\)-equivalence of nominal terms (ground or non-ground) can be easily derived.

\[ 
a \# X, a \# Y \leftarrow a \# f(X, Y, [a]Z) 
\]
\[ 
a \# X, b \# X \leftarrow [a]X \approx_\alpha [b]X 
\]

We have implemented the algorithms using OCAML \([18]\); we give benchmarks in Section 6. The implementation is available from \url{www.dcs.kcl.ac.uk/staff/maribel/CANS}.

In functional programming applications, matching problems are ground and in this case our algorithm is linear in time in the size of the problem, as indicated in the first line of the table above. To our knowledge, this is the only available implementation of nominal matching with this complexity. For non-ground matching problems, if the pattern is linear we have a log-linear algorithm; if there are variables in the right-hand side and the pattern is non-linear, then our matching algorithm has a quadratic worst case.

We have used the nominal matching algorithm to implement nominal rewriting \([9]\), and are currently deploying the algorithms in a rewriting tool that can be used to specify equational theories involving binders (see \([10,7]\)), and to evaluate functions working on data structures with binders.

The complexity of nominal unification is still an open problem. A related problem (unification of higher-order patterns \([16]\)) is linear \([23]\) but a quadratic factor exists between the size of a nominal problem and its representation as a higher-order pattern problem, see \([14,5]\). Note that due to this quadratic factor, our algorithms, even in the case of non-ground and non-linear matching problems, are linear in the size of the problem represented in higher-order abstract syntax.

This paper is organised as follows: In Section 2 we define nominal syntax and we recall the transformation rules given by Urban, Gabbay and Pitts \([26]\) to solve nominal problems. Section 3 describes our algorithms to solve nominal matching problems and to check the validity of \(\alpha\)-equality constraints. In Section 4 we discuss the data structures used to implement these algorithms. Section 5 analyses the complexity of the algorithms, and Section 6 gives some benchmarks obtained with our implementation. Section 7 presents an algorithm to implement nominal rewriting. Finally, we conclude the paper in Section 8.

\(^1\) Although in the case of first-order terms variables that cannot be instantiatied can be treated as constants, this is not the case for nominal terms since variables may have suspended permutations.
2. Background

In this section, we recall the syntax of nominal terms, define $\alpha$-equivalence and matching problems, and recall the rewriting rules given in [26] to solve them.

Let $\Sigma$ be a denumerable set of function symbols $f, g, \ldots; \mathcal{X}$ a denumerable set of variables $X, Y, \ldots$; and $\mathcal{A}$ a denumerable set of atoms $a, b, \ldots$. We assume that $\Sigma, \mathcal{X}$, and $\mathcal{A}$ are pairwise disjoint. In the intended applications, variables will be used to denote meta-level variables (unknowns), and atoms will be used to represent object-level variables.

A swapping is a pair of (not necessarily distinct) atoms, written $(a \ b)$. Permutations $\pi$ are lists of swappings, generated by the grammar:

$$\pi ::= \text{id} | (a \ b) \circ \pi$$

where $\text{id}$ is the identity permutation. We write $\pi^{-1}$ for the permutation obtained by reversing the list of swappings in $\pi$.

We denote by $\pi \circ \pi'$ the permutation containing all the swappings in $\pi$ followed by those in $\pi'$.

A pair of a permutation $\pi$ and a variable $X$ is called a suspension, written $\pi \cdot X$; we say that $\pi$ is suspended on $X$.

Nominal terms, or just terms for short, over $\Sigma, \mathcal{X}$, and $\mathcal{A}$, are trees built from atoms, suspensions, tuples, abstractions, and function applications. More precisely, terms are generated by the grammar:

$$s, t ::= a | \pi \cdot X | (s_1, \ldots, s_n) | [a]s | ft$$

We follow standard notational conventions, omitting brackets and abbreviating $\text{id} \cdot X$ as $X$ when there is no ambiguity.

A term is ground if it has no variables; $V(t)$ denotes the set of elements of $\mathcal{X}$ that occur in $t$. For example, $\lambda[a]a$ is a ground term, and $V(\lambda[a](X, X)) = \{X\}$. We refer the reader to [26,9] 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:

$$\text{id} \cdot t = t \quad \text{and} \quad ((a \ b) \circ \pi) \cdot t = (a \ b) \cdot (\pi \cdot t)$$

where

$$(a \ b) \cdot a = b \quad (a \ b) \cdot b = a$$
$$(a \ b) \cdot c = c \quad \text{if } c \notin \{a, b\}$$
$$(a \ b) \cdot (\pi \cdot X) = ((a \ b) \circ \pi) \cdot X \quad (a \ b) \cdot (f t) = f(a \ b) \cdot t$$
$$(a \ b) \cdot [n]t = [(a \ b) \cdot n](a \ b) \cdot t \quad (a \ b) \cdot (t_1, \ldots, t_n) = ((a \ b) \cdot t_1, \ldots, (a \ b) \cdot t_n)$$

A substitution is generated by the grammar:

$$\sigma ::= \text{id} | [X \mapsto s]$$

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 $\sigma$ by induction:

$$t \text{id} = t \quad \text{and} \quad t[X \mapsto s]\sigma = (t[X \mapsto s])\sigma$$

where

$$a[X \mapsto s] = a \quad (t_1, \ldots, t_n)[X \mapsto s] = (t_1[X \mapsto s], \ldots, t_n[X \mapsto s])$$
$$([a]t)[X \mapsto s] = [a](t[X \mapsto s]) \quad (ft)[X \mapsto s] = f(t[X \mapsto s])$$
$$(\pi \cdot X)[X \mapsto s] = \pi \cdot s \quad (\pi \cdot Y)[X \mapsto s] = \pi \cdot Y$$

The predicate $\#$ specifies a freshness relation between atoms and terms, and $\approx_\alpha$ denotes alpha-equivalence. Constraints have the form $a \# t$ or $s \approx_\alpha t$. 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 $s$. In the absence of variables, $a \# s$ and $s \approx_\alpha t$ are structural properties (to check $a \# s$ we just check that every $a$ in $s$ occurs under an abstractor), but in the presence of variables both predicates may depend on assumptions $a \# X$ about what will get substituted for the variables.

Formally, we define $\#$ and $\approx_\alpha$ inductively, by a system of axioms and rules, using $\#$ in the definition of $\approx_\alpha$ (see below). We write $ds(\pi, \pi') \# X$ as an abbreviation for $[n \cdot X | n \in ds(\pi, \pi')]$, where $ds(\pi, \pi') = \{n | \pi \cdot n \neq \pi' \cdot n\}$ is the set of atoms where $\pi$ and $\pi'$ differ (i.e., their difference set). Below $a, b$ are any pair of distinct atoms.
2.1. Checking and solving constraints

A locally confluent and terminating relation is also confluent, by Newman’s Lemma [17].

Below we define reduction relations (also called rewriting relations) on terms or problems. If \( \Rightarrow \) is a reduction relation, we will denote by \( \Rightarrow^* \) its transitive and reflexive closure. Irreducible terms are called normal forms. The following properties of reduction relations will be useful later.

A rewriting relation is:

- terminating if there are no infinite reduction chains;
- normalising if for all \( t \) there is some \( t' \) such that \( t \Rightarrow t' \) and \( t' \) is irreducible;
- locally confluent if, for each peak \( t \Rightarrow u \) and \( t \Rightarrow v \), there is a term \( s \) such that \( u \Rightarrow s \) and \( v \Rightarrow s \);
- confluent if for all \( t, u, v \) such that \( t \Rightarrow u \) and \( t \Rightarrow v \), there is a term \( s \) such that \( u \Rightarrow s \) and \( v \Rightarrow s \).

A locally confluent and terminating relation is also confluent, by Newman’s Lemma [17].

### 2.1. Checking and solving constraints

The following set of simplification rules from [26], acting on problems, where \( a, b \) denote any pair of distinct atoms, can be used to check the validity of \( \alpha \)-equality constraints.

\[
\begin{align*}
\frac{a \# b}{\alpha} & \quad \frac{a \# s}{\alpha} & \quad \frac{a \# s_1 \ldots a \# s_n}{\alpha} \\
\frac{a \# f s}{\alpha} & \quad \frac{a \# s}{\alpha} & \quad \frac{\pi^{-1} \cdot a \# X}{\alpha} \\
\frac{a \# [a] s}{\alpha} & \quad \frac{a \# s}{\alpha} & \quad \frac{\pi^{-1} \cdot a \# X}{\alpha}
\end{align*}
\]

We remark that \( \approx_{\alpha} \) is indeed an equivalence relation (see [26] for more details).

Below we define reduction relations (also called rewriting relations) on terms or problems. If \( \Rightarrow \) is a reduction relation, we will denote by \( \Rightarrow^* \) its transitive and reflexive closure. Irreducible terms are called normal forms. The following properties of reduction relations will be useful later.

A rewriting relation is:

- terminating if there are no infinite reduction chains;
- normalising if for all \( t \) there is some \( t' \) such that \( t \Rightarrow t' \) and \( t' \) is irreducible;
- locally confluent if, for each peak \( t \Rightarrow u \) and \( t \Rightarrow v \), there is a term \( s \) such that \( u \Rightarrow s \) and \( v \Rightarrow s \);
- confluent if for all \( t, u, v \) such that \( t \Rightarrow u \) and \( t \Rightarrow v \), there is a term \( s \) such that \( u \Rightarrow s \) and \( v \Rightarrow s \).

A locally confluent and terminating relation is also confluent, by Newman’s Lemma [17].

These rules generate a reduction relation on problems: \( Pr \Rightarrow Pr' \) if \( Pr' \) is obtained from \( Pr \) by applying a simplification rule. To check constraints we proceed as follows: Given a problem \( Pr \), we apply the rules until we get an irreducible problem, i.e., a normal form. If only a set \( \Delta \) of constraints of the form \( a \# X \) are left, then the original problem is valid in the context \( \Delta \) (i.e., \( \Delta \vdash Pr \)), otherwise it is not valid.
For example, the problem \([a]X \approx_\alpha [b]X\) reduces to the set of constraints \(a \neq X, b \neq X\); therefore \(a \neq X, b \neq X \vdash [a]X \approx_\alpha [b]X\), as mentioned in the introduction. A problem such as \(X \approx_\alpha a\) is not valid since it is irreducible; however, in this case \(X\) can be made equal to \(a\) by instantiation (i.e., applying a substitution) and 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 also 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 \((\Delta, \sigma)\) of a freshness context and a substitution, obtained from the simplification rules above enriched with an instantiating rule labelled with substitutions:

\[
\pi \cdot X \approx\alpha u, \quad Pr \xrightarrow{X\to\pi^{-1}\cdot u} Pr[X \mapsto \pi^{-1} \cdot u]
\]

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.

As shown in [26], these rules define a sound and complete nominal matching algorithm, where the most general solution to the input problem is obtained by computing its normal form \(\Delta\) and by composing the substitutions used in the instantiation rules. We give an example and refer to [26] for more details.

**Example 1.** Since there is a reduction sequence:

\[
[a]X \approx_\alpha [b]b \implies X \approx\alpha a, \quad a \neq b \quad \xrightarrow{a \leftarrow b} \quad a \neq b \implies \emptyset
\]

the most general solution of \([a]X \approx_\alpha [b]b\) is \((\emptyset, [X \mapsto a])\).

Although in the case of first-order terms we can, without loss of generality, restrict ourselves to matching problems with ground right-hand sides (it is sufficient to consider all variables as constants), this is not the case for nominal problems: when terms contain variables we may need to compute difference sets of permutations.

In the following sections we describe an algorithm that can be used to check \(\alpha\)-equivalence, and to solve nominal matching problems (linear or non-linear), based on the rules given above. We discuss the data structures used in its implementation, and analyse its complexity.

### 3. A modular algorithm to check \(\alpha\)-equivalence and solve matching constraints

Polynomial implementations of nominal unification [3] rely on the use of lazy permutations: permutations are only pushed down a term when this is needed to apply a transformation rule, and then, they are only pushed one level down the term. Since lazy permutations may grow (they accumulate), in order to obtain an efficient algorithm we will compose the swappings eagerly. The key idea is to work with a single current permutation, represented by an environment.

The algorithms to check \(\alpha\)-equivalence constraints and to solve matching problems (linear or non-linear) will be built in a modular way. The core module is composed of four phases and is common to both algorithms; only the final phase will be specific to matching or \(\alpha\)-equivalence.

#### 3.1. Environments

We begin by introducing the notion of an environment. Environments will be associated to terms and used to store a permutation and a set of atoms.

**Definition 1.** Let \(s\) and \(t\) be terms, \(\pi\) be a permutation, and \(A\) be a finite set of atoms. An environment \(\xi\) is a pair \((\pi, A)\). We denote by \(\xi_s\) the permutation (resp. \(\xi_A\) the set of atoms) of an environment. We write \(s \approx\alpha \xi \cdot t\) to represent \(s \approx\alpha \xi_{\pi \cdot t}, \xi_A \neq t\) and call \(s \approx\alpha \xi \cdot t\) an environment constraint.

For example, the environment constraint \(X \approx\alpha \xi \cdot b\) where \(\xi_{\pi} = (a \cdot b)\) and \(\xi_A = \{a\}\) represents the problem \(X \approx\alpha (a \cdot b) \cdot b, a \neq b\).

**Definition 2.** An environment problem \(Pr\) is either \(\bot\) or has the form

\[
s_1 \approx\alpha \xi_1 \cdot t_1, \ldots, s_n \approx\alpha \xi_n \cdot t_n
\]

where \(s_i \approx\alpha \xi_i \cdot t_i\) \((1 \leq i \leq n)\) are environment constraints. We will sometimes abbreviate it as \((s_1 \approx\alpha \xi_1 \cdot t_1)^\odot\).

**Definition 3.** 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 the rule:

\[
s \approx\alpha \xi \cdot t \implies s \approx\alpha \xi_{\pi \cdot t}, \xi_A \neq t
\]
as many times as possible. We denote by \([Pr]\) the standard form of an environment problem \(Pr\). Thus:

\[
[\xi_s \triangleright_\alpha _{1} \triangleright t_1, \ldots, s_n \triangleright_\alpha _{n} \triangleright t_n] = s_1 \triangleright_\alpha _{(\xi_1)_{\pi} \cdot t_1}, (\xi_1)_{A} \# t_1, \ldots, s_n \triangleright_\alpha _{(\xi_n)_{\pi} \cdot t_n}, (\xi_n)_{A} \# t_n
\]

This rule is terminating because it consumes \(\triangleright\) each time, without creating any. There are no superpositions, so the system is locally confluent and because it terminates it is confluent [17]. Therefore the standard form of an environment problem exists and is unique.

**Definition 4.** 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.

As a consequence of Definition 4, two equivalent environment problems have the same set of solutions.

The set of reduction rules on standard problems given in Section 2 transforms a problem into an equivalent one (i.e., solutions are preserved, see [26]). Below we will give a set of transformation rules for environment problems, and we will prove that if there is a step of rewriting \(Pr \Rightarrow Pr'\) then \(Pr\) and \(Pr'\) are equivalent.

It is easy to translate standard problems into environment problems, as shown below.

**Definition 5.** The translation \(Env\) from standard problems into environment problems is inductive, with base cases:

\[
Env(s \triangleright_\alpha t) = s \triangleright_\alpha \xi \triangleright t \quad \text{where} \quad \xi = (Id, \emptyset) \quad \text{and}
\]

\[
Env(A \# t) = t \triangleright_\alpha \xi \triangleright t \quad \text{where} \quad \xi = (Id, A).
\]

This transformation is linear in time and in space. Therefore, from a (log-)linear algorithm solving environment problems we can derive a (log-)linear algorithm to solve standard problems.

In the following sections, we restrict our attention to checking \(\alpha\)-equivalence constraints and solving matching problems. In the latter case, in environment constraints \(s \triangleright_\alpha t\), 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.

### 3.2. Core algorithm

The core of the algorithm transforms an environment problem \((s_i \triangleright_\alpha (\xi_i \triangleright t_i))^n\) into an equivalent standard problem of the form \((X_i \triangleright_\alpha t_i)_{i \in I}, (A_j \# X_j)_{j \in J}\). There are four phases in the core algorithm. The first one reduces \(\triangleright_\alpha\) constraints, by propagating \(\xi_i\) over \(t_i\). The second phase eliminates permutations on the left-hand side of constraints, and the third phase reduces freshness constraints, also by propagating \(\xi_i\) over \(t_i\). Finally, the fourth phase computes the standard form of the resulting problem.

**Definition 6.** Let \(Pr\) be an environment problem. We denote by \(\overline{Pr}c\) the result of applying the core algorithm on \(Pr\).

Below we describe each phase of the core algorithm.

**Phase 1.** The input is an environment problem \(Pr = (s_i \triangleright_\alpha (\xi_i \triangleright t_i))^n\), which will be reduce by applying the following transformation rules.

\[
Pr, \quad a \quad \triangleright_\alpha \xi \triangleright t \quad \to \quad \begin{cases} 
Pr, & \text{if } a = \xi_{\pi} \cdot t \text{ and } t \notin \xi_{A} \\
\perp, & \text{otherwise}
\end{cases}
\]

\[
Pr, \quad (s_1, \ldots, s_m) \triangleright_\alpha \xi \triangleright t \quad \to \quad \begin{cases} 
Pr, \quad (s_1 \triangleright_\alpha \xi \triangleright u_1)_{1}^{m} & \text{if } t = (u_1, \ldots, u_m) \\
\perp, & \text{otherwise}
\end{cases}
\]

\[
Pr, \quad f \quad s \quad \triangleright_\alpha \xi \triangleright t \quad \to \quad \begin{cases} 
Pr, \quad s \triangleright_\alpha \xi \triangleright u & \text{if } t = f \quad u \\
\perp, & \text{otherwise}
\end{cases}
\]

\[
Pr, \quad [a]s \quad \triangleright_\alpha \xi \triangleright t \quad \to \quad \begin{cases} 
Pr, \quad s \triangleright_\alpha \xi' \triangleright u & \text{if } t = [b]_{u} \\
\perp, & \text{otherwise}
\end{cases}
\]

where \(\xi' = ((\alpha_{\pi} \cdot b) \circ \xi_{\pi}, (\xi_{A} \cup \{\xi_{\pi}^{-1} \cdot a\}) \setminus \{b\})\) in the last rule, and \(a, b\) could be the same atom.

We show below that the rules transform an environment problem into another equivalent environment problem.

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 \((\alpha_{\pi} \cdot X_i \triangleright_\alpha \xi_i \triangleright s_i)^n\) where \(s_i\) are nominal terms and \(n \geq 0\).
Proof. If the left-hand side of an $\approx_\alpha$ constraint is an atom, function application, tuple or abstraction, then the constraint can be reduced using the phase 1 rules. Note that if the problem is ground, the ph1nf is either $\bot$ or empty. □

**Proposition 2 (Correctness).** Let $Pr$ be an environment problem and assume $Pr \equiv Pr'$ using the phase 1 rules. Then $Pr$ and $Pr'$ are equivalent.

**Proof.** To prove that $Pr$ and $Pr'$ are equivalent we have to show that their standard forms have the same solutions. We prove this by showing that if a problem reduces to $\bot$ then it has no solutions, and that if $Pr \equiv Pr' \neq \bot$ then there is a problem $Q$ such that $\llbracket Pr \rrbracket \equiv Q$ using standard reduction rules (given in Section 2), and $Q$ is equivalent to $\llbracket Pr' \rrbracket$. Since standard reduction steps preserve solutions, this implies the equivalence of $Pr$ and $Pr'$. We distinguish cases according to the rule applied:

- $Pr, a \approx_\alpha \xi \circ t \to Pr'$. Then $\llbracket a \approx_\alpha \xi \circ t \rrbracket = a \approx_\alpha \xi \pi \cdot t$, $\xi A \neq t$ so if $t$ is not an atom or if $t$ is an atom but $a \neq \xi \pi \cdot t$ or $t \in \xi A$ then there is no solution.
  
  If $t$ is an atom $b$, $\xi \pi \cdot t = a$ and $t \notin \xi A$, then $Pr' = Pr$, and the reduction step can be simulated as follows:
  
  $\llbracket Pr \rrbracket, a \approx_\alpha \xi \pi \cdot t, \xi A \neq t \equiv \llbracket Pr \rrbracket, a \approx_\alpha a, (c \# b)_{c \in \xi A} \rightarrow \llbracket Pr \rrbracket, (c \# b)_{c \in \xi A} \rightarrow^* \llbracket Pr \rrbracket$

- $Pr, f \approx_\alpha \xi \circ t \to Pr'$. If $t$ is not a term of the form $f$ $u$ then there is no solution. If $t = f$ $u$ the step can be simulated by standard reduction steps as follows:
  
  $\llbracket Pr \rrbracket, f s \approx_\alpha \xi \pi \cdot f u, \xi A \neq f \equiv \llbracket Pr \rrbracket, s \approx_\alpha \xi \pi \cdot u, \xi A # u = \llbracket Pr, s \approx_\alpha \xi \pi \cdot u \rrbracket = \llbracket Pr' \rrbracket$

- $Pr, (s_1, \ldots, s_n) \approx_\alpha \xi \circ t \to Pr'$. If $t$ is not a tuple, there is no solution. If $t = (u_1, \ldots, u_n)$ the step can be simulated by standard steps as follows:
  
  $\llbracket Pr \rrbracket, (s_1, \ldots, s_n) \approx_\alpha \xi \pi \cdot (u_1, \ldots, u_n), \xi A \neq (u_1, \ldots, u_n)$
  
  $\rightarrow^* \llbracket Pr \rrbracket, (s_1 \approx_\alpha \xi \pi \cdot u_1) \bullet, (\xi A \neq u_1) \Rightarrow \llbracket Pr, (s_1 \approx_\alpha \xi \pi \cdot u_1) \rrbracket = \llbracket Pr' \rrbracket$

- $Pr, \{a\} \approx_\alpha \xi \circ t \to Pr'$. If $t$ is not an abstraction, there is no solution. If $t = \{b\} u$, where $a$ and $b$ denote two atoms not necessarily different, then we distinguish two cases:
  
  If $\xi \pi \cdot b = a$, then
  
  $\llbracket Pr \rrbracket, \{a\} s \approx_\alpha \xi \pi \cdot \{b\} u, \xi A \neq \{b\} u$
  
  $\rightarrow^* \llbracket Pr \rrbracket, s \approx_\alpha \xi \pi \cdot u, \xi A \lo (b) \neq u$
  
  $\rightarrow \llbracket Pr \rrbracket, s \approx_\alpha ((a \xi \pi \cdot b) \circ \xi \pi) \cdot u, (\xi A \cup \{\xi \pi^{-1} \cdot a\}) \\lo \{b\} \neq u = \llbracket Pr, s \approx_\alpha \xi \pi \cdot u \rrbracket = \llbracket Pr' \rrbracket$

  where $\xi' = ((a \xi \pi \cdot b) \circ \xi \pi, (\xi A \cup \{\xi \pi^{-1} \cdot a\}) \\lo \{b\})$. If $\xi \pi \cdot b \neq a$, then, since freshness is preserved by permutation (i.e., it is an equivariant relation, see [26,9]), we obtain:
  
  $\llbracket Pr \rrbracket, \{a\} s \approx_\alpha \xi \pi \cdot \{b\} u, \xi A \neq \{b\} u$
  
  $\rightarrow^* \llbracket Pr \rrbracket, s \approx_\alpha ((a \xi \pi \cdot b) \circ \xi \pi) \cdot u, \xi A \lo \{b\} \neq u, a \# (\xi \pi \cdot u)$
  
  $\rightarrow \llbracket Pr \rrbracket, s \approx_\alpha ((a \xi \pi \cdot b) \circ \xi \pi) \cdot u, (\xi A \cup \{\xi \pi^{-1} \cdot a\}) \\lo \{b\} \neq u = \llbracket Pr, s \approx_\alpha \xi \pi \cdot u \rrbracket = \llbracket Pr' \rrbracket$

  where $\xi' = ((a \xi \pi \cdot b) \circ \xi \pi, (\xi A \cup \{\xi \pi^{-1} \cdot a\}) \\lo \{b\})$. □

**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:

$\pi \cdot X \approx_\alpha \xi \circ t \to X \approx_\alpha (\pi^{-1} \cdot \xi) \circ t \quad (\pi \neq \text{Id})$

where $\pi^{-1} \cdot \xi = (\pi^{-1} \circ \xi, \xi A)$. Note that $\pi^{-1}$ applies only to $\xi A$ here, because $\pi \cdot X \approx_\alpha \xi \circ t$ represents $\pi \cdot X \approx_\alpha \xi \pi \cdot t$, $\xi A \neq t$.

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 3 (Phase 2 normal forms).** Given a ph1nf problem, it has a unique normal form for the rule above, and it is either $\bot$ or a problem of the form $(X_i \approx_\alpha \xi_i \circ t_i)_{i=1}^n$, where the terms $t_i$ are standard nominal terms.

**Proof.** The rule is clearly terminating (each application consumes a permutation on the left) and it does not overlap with itself, therefore it is confluent [17]. This implies the unicity of normal forms. Given a ph1nf problem, its normal form cannot contain a suspension on the left-hand side. □
Proposition 4 (Correctness). $\pi \cdot X \approx \alpha \xi \odot t$ is equivalent to $X \approx \alpha (\pi^{-1} \cdot \xi) \odot t$.

Proof. We need to show that the standard forms of both problems are equivalent. Since $\|\pi \cdot X \approx \alpha \xi \odot t\| = \pi \cdot X \approx \alpha \xi \odot t$, $\xi A \neq t$, and $\|X \approx \alpha (\pi^{-1} \cdot \xi) \odot t\| = X \approx \alpha (\pi^{-1} \odot \xi) \cdot t$, $\xi A \neq t$, the result follows directly from the preservation of $\approx \alpha$ by permutations (see [26,9]). □

Phase 3. In the phases 1 and 2 we deal with $\approx \alpha$ 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 $X \approx \alpha \xi \odot t$. We reduce it with the following rewrite rules, which propagate $\xi$ over $t$ and deal with problems containing $\bot$ (denoted Pr[⊥]):

$$
\begin{align*}
\xi \odot a & \rightarrow \begin{cases} 
\xi \cdot a & a \neq \xi \\
\bot & a \in \xi 
\end{cases} \\
\xi \odot f t & \rightarrow f(\xi \odot t) \\
\xi \odot (t_1, \ldots, t_j) & \rightarrow (\xi \odot t_1)^j \\
\xi \odot [a]s & \rightarrow [\xi \cdot a](\xi \odot s) \\
\xi \odot (\pi \cdot X) & \rightarrow (\xi \odot \pi) \odot X \\
Pr[\bot] & \rightarrow \bot
\end{align*}
$$

where $\xi \setminus [a] = (\xi \cdot, \xi A \setminus [a])$ and $\xi \odot \pi = ((\xi \odot \pi), \pi^{-1}(\xi A))$.

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 7. The language of normal environment terms is defined by:

$$T_\xi = a | f T_\xi | (T_\xi, \ldots, T_\xi) | [a]T_\xi | \xi \odot X$$

If the problem is irreducible (i.e., it is a normal form for the rules above), we say it is a phase 3 normal form, or ph3nf for short.

Proposition 5 (Phase 3 normal forms — ph3nf). The normal forms for this phase are either $\bot$ or $(X_1 \approx \alpha t_1)^i$ where $t_1 \in T_\xi$.

Proof. By inspection of the rules, it is easy to see that environments move down, and suspend on variables. □

To give a semantics to the problems generated in phase 3, we extend the definition of a standard form (see Definition 2) as follows:

Definition 8 (Standard form). The standard form of a problem with suspended environments is obtained by normalising with the rewriting rule

$$s \approx \alpha C[\xi \odot t] \rightarrow s \approx \alpha C[\xi \cdot t], \xi A \neq t$$

where $s$ and $t$ are nominal terms.

Remark 1. The rule in Definition 3 is a particular case of this one, taking an empty context, that is, $C[\ ] = [\ ]$.

This rule is terminating and confluent, therefore normal forms are unique; we will use the notation $[[Pr]]$ as before.

Proposition 6 (Correctness). Let $Pr$ be a problem in ph2nf, and $Pr \implies Pr'$ using a phase 3 rule. Then $Pr$ and $Pr'$ are equivalent.

Proof. As before, it is sufficient to show that if $\bot$ is obtained then the problem has no solutions, and otherwise the standard forms of $Pr$ and $Pr'$ are equivalent. We show the three interesting cases (recall that standard reductions preserve solutions).

- Assume we have $s \approx \alpha C[\xi \odot a]$. Then its standard form is $[[s \approx \alpha C[\xi \cdot a]]]$. $\xi A \neq a$. Notice that $\bot$ can only be obtained if $a \in \xi A$, and in this case the standard form has no solutions.

- If $a \neq \xi A$ then $[[s \approx \alpha C[\xi \cdot a]]]$, $\xi A \neq a$ is equivalent to $[[s \approx \alpha C[\xi \cdot a]]]$.

- Assume we have $s \approx \alpha C[[\xi \odot [a]t]]$. Then its standard form is $[[s \approx \alpha C[[\xi \odot a][\xi \cdot t]]]]$, $\xi A \neq [a]t$ which is equivalent to $[[s \approx \alpha C[[\xi \odot a][\xi \cdot t]]]]$, $\xi A \setminus [a] \neq t$, which is the standard form of $s \approx \alpha C[[\xi \odot a][\xi \setminus [a]] \odot t]$. 

• Assume we have \( s \approx_{\alpha} C[\xi \odot (\pi \cdot X)] \). Then its standard form is \( \llbracket s \approx_{\alpha} C[(\xi \pi \circ \alpha) \cdot X] \rrbracket \). \( \xi \# \pi \cdot X \) and using standard reduction rules we obtain \( \llbracket s \approx_{\alpha} C[(\xi \pi \circ \alpha) \cdot X] \rrbracket, \pi^{-1}(\xi A) \# X \), equivalent to \( \llbracket s \approx_{\alpha} C[(\xi \circ \alpha) \odot X] \rrbracket \).

Phase 4. This phase computes the standard form of a ph3nf, using a particular case of the rule given in Definition 8:

\[
X \approx_{\alpha} C[\xi \odot X'] \quad \Longrightarrow \quad X \approx_{\alpha} C[\xi \pi \cdot X'], \xi \# X'
\]

Definition 9. We define the sets of variables on the left-hand (resp. right-hand) side of an \( \approx_{\alpha} \)-constraint as \( V_l(s \approx_{\alpha} t) = V(s) \) (resp. \( V_r(s \approx_{\alpha} t) = V(t) \)). We extend the notation to problems as follows: \( V_{l/2}(A \# t) = \emptyset \) and \( V_{l/2}(Pr_1, Pr_2) = V_{l/2}(Pr_1) \cup V_{l/2}(Pr_2) \).

Proposition 7 (Phase 4 normal forms — ph4nf). If we normalise a ph3nf using the rule above, we obtain either \( \bot \) 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.

Moreover, \( (X_i)_{i \in I} \subseteq V_l(Pr) \) and \( (X_j)_{j \in J} \subseteq V_r(Pr) \). Thus, if the right-hand sides of \( \approx_{\alpha} \)-constraints in \( Pr \) are ground, there are no freshness constraints in the ph4nf (because \( V_r(Pr) = \emptyset \)).

Proof. By Proposition 5, environments can only be suspended on variables in a ph3nf. So if the problem is irreducible by the rule above, it cannot contain any environment.

The variable property follows from the fact that the rules never move subterms of the left-hand side of an \( \approx_{\alpha} \)-constraint into the right-hand side, and all the freshness constraints generated involve subterms of the right-hand side.

Clearly, this phase preserves the set of solutions, it terminates and it does not raise \( \bot \), therefore if the resulting problem is \( \bot \) so was the ph3nf.

Since all the reduction rules, except the rule dealing with \( \bot \), 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 \( \bot, Pr = \bot \)):

\[
(s_i \approx_{\alpha} \xi_i \odot t_i)^c = (s_i \approx_{\alpha} \xi_i \odot t_i)^c_1
\]

Thus, without loss of generality we can consider the input of the core algorithm to be one single constraint \( s \approx_{\alpha} \xi \odot t \).

3.3. Checking the validity of \( \alpha \)-equivalence constraints

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

\[
(\alpha) \quad Pr, X \approx_{\alpha} t \quad \Longrightarrow \quad \begin{cases} Pr, \supp(\pi) \# X & \text{if } t = \pi \cdot X \\ \bot & \text{otherwise} \end{cases}
\]

where \( \supp(\pi) \) is the support of \( \pi : \supp(\pi) = \{a | \pi \cdot a \neq a\} \).

Since this rule is terminating (each application consumes one \( \approx_{\alpha} \)-constraint) and locally confluent, it is confluent [17], therefore normal forms exist and are unique.

Definition 10. We will denote by \( Pr^{\approx_{\alpha}} \) the normal form of \( Pr^c \) by the rule above.

Proposition 8 (Normal forms \( Pr^{\approx_{\alpha}} \)). \( Pr^{\approx_{\alpha}} \) is either \( \bot \) or \( (A_i \# X_i)^n \).

Proof. \( Pr^c \) is a ph4nf, so by Proposition 7 it is either \( \bot \) or \( (X_i \approx_{\alpha} t_i)^n, (A_j \# X_j)_{j \in J} \) where \( t_i \) are standard terms. While there are constraints \( X_i \approx_{\alpha} t_i \), the problem is reducible by \( (\alpha) \), and each \( \approx_{\alpha} \)-constraint is replaced by a set of freshness constraints.

Proposition 9 (Correctness). If \( Pr^{\approx_{\alpha}} = \bot \) then \( Pr \) is not valid. If \( Pr^{\approx_{\alpha}} = (A_i \# X_i)^n \) then \( Pr^{\approx_{\alpha}} \vdash Pr \).

Proof. The core algorithm preserves solutions, as a consequence of Propositions 2, 4, 6. Moreover, \( X \approx_{\alpha} t \) is valid (see Section 2) if and only if \( t = \pi \cdot X \) and \( \supp(\pi) \# X \), because \( ds(\pi, Id) = \supp(\pi) \). Hence \( (\alpha) \) is correct.

If the left-hand sides of \( \approx_{\alpha} \)-constraints in \( Pr \) are ground, then \( Pr^c = Pr^{\approx_{\alpha}} \); rule \( (\alpha) \) is not necessary in this case (by Proposition 7). Formally:

Proposition 10. Let \( Pr = (s_i \approx_{\alpha} \xi_i \odot t_i)^n \) where \( V(s_i) = \emptyset \) for \( 1 \leq i \leq n \). Then \( Pr^{\approx_{\alpha}} = Pr^c \).
Remark 2. The rule \((\alpha)\), when it does not involve \(\bot\), is also local (i.e., it only affects one constraint in the problem), so in practice we will implement two functions, one taking only \(s \approx_{\alpha} \xi \diamond t\) as input and computing \(s \approx_{\alpha} \xi \diamond t \approx_{\alpha}^\#\), and the other taking an environment problem as input and applying the first function on each constraint: \((s_1 \approx_{\alpha} \xi_i \diamond t_1) \approx_{\alpha}^\# = (s_1 \approx_{\alpha} \xi_i \diamond t_1 \approx_{\alpha})^\#\).

In the case of ground terms, our algorithm to check \(\alpha\)-equivalence relies only on the information stored in \(\xi\). Thus, it could be seen as an implementation of the inductive definition of ground \(\alpha\)-equality used in [12] (and attributed to Shankar [24]), which also relies on a list of pairs of names.

3.4. 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 \(\overrightarrow{Pr}^\approx\) by the following rule.

\[(\gamma\approx)\quad \overrightarrow{Pr}, \ X \approx_{\alpha} s, \ X \approx_{\alpha} t \quad \overrightarrow{\quad \overrightarrow{Pr}, \ X \approx_{\alpha} \overrightarrow{s}, \ X \approx_{\alpha} \overrightarrow{t} \quad \overrightarrow{\quad \bot \quad \overrightarrow{\quad \text{if } \overrightarrow{s} \approx_{\alpha} \overrightarrow{t} \neq \bot \quad \overrightarrow{\quad \text{otherwise}}}\]

This rule is terminating: each reduction consumes at least one \(\approx_{\alpha}\)-constraint since \(\overrightarrow{Pr}^\approx\) does not introduce \(\approx_{\alpha}\)-constraints (by Proposition 8) and is also terminating.

Definition 11. We denote by \(\overrightarrow{Pr}^\approx\gamma\) a normal form of \(\overrightarrow{Pr}^\approx\) by the rule \((\gamma\approx)\).

Proposition 11 (Normal forms \(\overrightarrow{Pr}^\approx\)). If we normalise \(\overrightarrow{Pr}^\approx\) using the rule above, we obtain either \(\bot\) or \((X_i \approx_{\alpha} t_i)^n\) where \(t_i\) are standard terms, all \(X_i\) in the equations \((X_i \approx_{\alpha} t_i)^n\) are different variables and \(V_i, j: X_i \notin V(t_j)\).

Proof. By Proposition 8, rule \((\gamma\approx)\) does not produce \(\approx_{\alpha}\)-constraints. The property then follows from Proposition 7, since \(V(t_i) \subseteq V_i(Pr), X_i \in V_i(Pr), \) and \(V_i(Pr) \cap V_j(Pr) = \emptyset\) in a matching problem.

A problem of the form \((X_i \approx_{\alpha} t_i)^n\) where all \(X_i\) are distinct variables and \(X_i \notin V(t_j)\) is the coding of an idempotent substitution \(\sigma\) defined by

\[\sigma(X) = \begin{cases} t_i & \text{if } \exists i \ X = X_i \\ X & \text{otherwise} \end{cases}\]

\((A_i \neq X_i)^n\) is a freshness context \(\Delta\). So the result of the algorithm is either \(\bot\) or a pair \((\sigma, \Delta)\) of a substitution and a freshness context.

Proposition 12 (Correctness). \(\overrightarrow{Pr}^\approx\gamma\) is a most general solution of the matching problem \(Pr\).

Proof. Rule \((\gamma\approx)\) is justified by the equivalence:

\[X \approx_{\alpha} s, \ X \approx_{\alpha} t \iff X \approx_{\alpha} s, \ s \approx_{\alpha} t\]

If \(\bot\) is raised, either:

- \(\overrightarrow{Pr}^\approx = \bot\) and the problem has no solution by correctness of the core algorithm.
- Rule \((\gamma\approx)\) raised \(\bot\), so \(s \approx_{\alpha} t \approx_{\alpha} = \bot\), i.e., \(s\) and \(t\) are not \(\alpha\)-equivalent, and the problem has no solution.

Note that if variables occur linearly in patterns (i.e., we have a linear matching problem), then the result of the core algorithm is already the solution of the problem. Formally:

Proposition 13. Let \(Pr = (s \approx_{\alpha} \xi \diamond t)\) where for all \(X \in V(s), X\) occurs only once in \(s\). Then \(\overrightarrow{Pr}^\approx = \overrightarrow{Pr}^\approx\).

Proof. If all variables \(X\) in \(s\) are linear, then \(\overrightarrow{Pr}^\approx\) contains only one equation for each \(X\) and rule \((\gamma\approx)\) cannot be applied.

4. Implementation

In this section we describe the implementation of the core algorithm, then the implementation of the final phases specific to \(\alpha\)-equivalence and matching respectively. Below, \(Pr_0\) will denote the input problem and \(A_0\) the set of atoms occurring in \(Pr_0\).
Coding the problem. Terms are represented as trees. We code $\xi \circ t$ by attaching $\xi$ to the root of $t$. In the following, we use $t$ also to denote the root of the tree representing $t$. 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.

Before showing how we code permutations, sets and environments, we show that it is sufficient to work with one single “current” environment: we never need to create or copy environments.

Avoiding environment creation in the core algorithm. We assume without loss of generality that the input of the core algorithm is a single constraint $s \approx_\alpha \xi \circ t$. We can see phase 1 and phase 3 as “environment propagation”: each rule either raises $\bot$ or moves $\xi$ one level down (i.e., towards the leaves of the problem). Instead of doing each phase separately we can combine them: to reduce the problem $s \approx_\alpha \xi \circ t$, we will apply phase 1 rules, and when we reach a variable on the left-hand side (i.e., we have obtained a constraint $\pi \cdot X \approx_\alpha \xi \circ t$), we will apply phase 2 directly to it, to obtain $X \approx_\alpha \xi' \circ t$. Then we will apply phase 3 on $\xi' \circ t$ and when we reach a variable (i.e., we have $\xi \circ X$) we will apply directly phase 4 to it to obtain its standard form. In this way, we have a strategy of reduction which deals with each constraint locally: this is one of the key ideas to avoid copying environments. In other words, 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. For example:

- The rule $(f \ s) \approx_\alpha \xi \circ (f \ t) \implies s \approx_\alpha \xi \circ t$ propagates $\xi$ on the right.
- The rule $[a]s \approx_\alpha \xi \circ [b]t \implies s \approx_\alpha \xi' \circ t$ updates $\xi$ to obtain $\xi'$.
- The rule $([s])_\alpha \approx_\alpha \xi \circ ([t])_\alpha \implies ([s])_\alpha \approx_\alpha \xi \circ ([t])_\alpha$ copies the environment into the elements in the tuple.
- Similarly for the rules in phase 3. The rule in phase 2 simply updates the environment, and phase 4 outputs the solution.

Instead of copying environments (in the case of tuples), we will share them, copying only pointers. 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.

We update environments in the algorithm by either composing the current permutation with another permutation $\pi$ (this operation can be reversed by composing again with $\pi^{-1}$), or by adding/removing an atom from the current set (this operation can be reversed by removing/adding it). Therefore, we need to keep track of the operations we made in the environment, fully reduce the current constraint, and then reverse the operations before reducing another constraint.

For example: the phase 3 rule $\xi \circ [a]s \implies [\xi \circ a][(\xi \setminus \{a\}) \circ s]$ removes $a$ from $\xi_A$ if $a$ is in $\xi_A$ (resp. does nothing if $a$ is not in $\xi_A$). After having fully reduced $[\xi \circ a][(\xi \setminus \{a\}) \circ s]$, we will restore $\xi$ to its original state by adding $a$ to $\xi_A$ (resp. do nothing). The only information we need to keep, to be able to restore $\xi$, is whether $a \in \xi_A$.

Permutations and sets. The rules given in Section 3 use several operations on permutations and sets:

- $a \in A$: membership test.
- $A \cup \{a\}$: add an atom to a set.
- $A \setminus \{a\}$: remove an atom from a set.
- $\pi \cdot a$: compute the image of an atom by a permutation.
- $\pi^{-1}$: compute the inverse of a permutation.
- $\text{supp}(\pi)$: compute the support of a permutation.

Definition 12. An update operation in the environment consists of applying a swapping to a permutation, or adding/removing an atom from a set.

To be efficient, 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 (a new map is returned) we do not need to copy them.

We will use the same interface for mutable arrays and functional maps. In the case of mutable arrays, updates will be made directly in the array, destroying the previous value, and copying will create a new array. In the case of functional maps, updates will create new maps (in logarithmic time and constant space), and copying will only create a new pointer to the same map (a constant time operation). We will use either mutable arrays or functional maps depending on the kind of problem to be solved:
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 3.

Since we often need to know 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. Thus, permutations are represented as tuples of the actual permutation, its inverse and its support, all coded the same way, either as a mutable array or a functional map. Every time we update the permutation, we update as well its inverse and its support, which is done in constant time for arrays and logarithmic time for maps. Thanks to this technique we can always access the inverse and the support of a permutation in constant time.

### Implementing the algorithms.

The implementation of the core algorithm is essentially a traversal of the data structure representing the problem $Pr_0$, propagating the environment using the techniques above. The result is a list $Pr_0^c$ of constraints in ph4nf. The $\alpha$-equivalence algorithm then takes each $\approx_\alpha$-constraint in the list $Pr_0^c$ and reduces it with $(\alpha)$. The matching algorithm applies the rule $(\gamma \approx)$: it traverses the list to take for each variable $X$ the constraint $X \approx_s s$ 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:

$$Pr, X \approx_\alpha t \rightarrow \begin{cases} Pr, \ S[X] \approx_\alpha t \approx_\alpha & \text{if } S[X] \approx_\alpha t \approx_\alpha \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

where $S[X] \approx_\alpha t \approx_\alpha$ is computed using arrays.

If $\bot$ is raised the problem has no solution, otherwise let $Pr_t$ be the resulting problem. $(S, Pr_t)$ is a pair of a substitution and a freshness context, which is the most general solution to the input problem $Pr_0$.

## 5. Complexity

In this section we analyse the complexity of the core algorithm and the final $\alpha$-equivalence and matching phases.

Let $Pr_0 = s \approx_\alpha \xi \odot t$ be the input problem; $s$ and $t$ are coded as trees and $\xi$ is coded as a pair of a permutation and a set of atoms (permutations being a tuple of the actual permutation, its inverse, and its support), as discussed in Section 4. 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 $Pr_0$). Let $|t|_0$ be the number of nodes in the tree representing $t$. Let $P$ be the multiset of permutations in $s$, $t$, and $|\pi|_1$ be the size of the array representing $\pi \in P$ (or the size of the map if we are dealing with non-ground problems). Finally, let $MV(t)$ be the multiset of the occurrences of variables in $t$.

### Core algorithm.

Below we analyse the cost of the traversal of the data structure, and the cost of the operations involved in the rules.

#### Definition 13.

The size of a term $t$, written $|t|$, is defined as follows:

$$|a| = |X| = 1$$
$$|f t| = 1 + |t|$$
$$|(t_1, \ldots, t_j)| = 1 + |t_1| + \cdots + |t_j|$$
$$|\alpha| = 1 + |s|$$
$$|\pi \cdot X| = 1 + |\pi|$$

where $|\pi|$ is the size of the array/map representing $\pi$.

#### Definition 14.

The size of the problem $s \approx_\alpha \xi \odot t$, written $|s \approx_\alpha \xi \odot t|$, is defined as $|s \approx_\alpha \xi \odot t| = |s| + |\xi| + |t|$ where $|\xi| = 2 \times |\xi_0| + |\xi_A|$, $|\xi_0|$ (resp. $|\xi_A|$) is the size of the array/map representing $\xi_A$.

#### Proposition 14.

The core algorithm is linear in the size of the problem $Pr_0$ in the ground case, using mutable arrays. In the non-ground case it is log-linear using functional maps and $\vartheta(|s \approx_\alpha t| + |M_{A_0}| \times |t|_0)$ using mutable arrays.
Proof. There are three kinds of rules in the core algorithm:

- propagation rules, such as the rules in phase 1 and phase 3, except the rule $\xi \circ (\pi \cdot X) \implies (\xi \circ \pi) \circ X$;
- rules dealing with input permutations (i.e., permutations that occur in the initial problem $P_{\mathit{ini}}$), such as phase 2 rules and the rule $\xi \circ (\pi \cdot X) \implies (\xi \circ \pi) \circ X$;
- rules dealing with $\xi \circ X$ to compute standard form.

The propagation rules visit each node in the tree representation of $t$ at most once. They may apply at most one swapping (resp. adding/removing an atom) to modify the environment and the same again (resp. removing/adding this atom) to restore it. All these operations are done a constant number of times and in constant (resp. logarithmic) time with arrays (resp. maps). Hence, they take $\vartheta(|t_{\mathit{ini}}|)$ time with arrays and $\vartheta(\log(|t_{\mathit{ini}}|) \times |t_{\mathit{ini}}|)$ with maps.

Rules dealing with permutations, i.e., phase 2 rules and the rule $\xi \circ (\pi \cdot X) \implies (\xi \circ \pi) \circ X$, compose an input permutation $\pi$ and the environment. This takes $\vartheta(|\pi|)$ time per input permutation $\pi$ with arrays (resp. maps), as described in Section 4. They are applied at most once per input permutation, so take $\sum_{\pi \in P} \vartheta(|\pi|)$ in total with arrays and $\sum_{\pi \in P} \log(|\pi|) \times |\pi|$ with maps.

If the right-hand side terms are ground, only these two kinds of rules can be applied, so the algorithm is linear in the size of the problem. Otherwise, there’s also the rule dealing with $\alpha$-equivalence constraint. That is, when we reach an abstraction $\alpha \vdash b \in A$ on the current node of $t$, and is done in constant time with functional maps and $\vartheta(|M_{\alpha_{0}}|)$ time and space with arrays.

If the problem consists of more than one matching or $\alpha$-equivalence constraint, they can be solved independently. Thus, we have the algorithm on each of them and concatenate the results, obtaining again an algorithm of the same complexity.

Remark 3. Our algorithm checks $\approx_{\mathit{eq}}$-constraints and the freshness constraints that are generated, all in one traversal of the data structure representing the problem (relying on the environment to provide information about the freshness constraints generated). We have chosen to develop the algorithm in this way because it is simpler, but it is possible to solve the $\approx_{\mathit{eq}}$-constraints and the generated freshness constraints separately, while keeping the same complexity. Instead of using an environment with a permutation and a set of atoms, we can split the environment into two parts, one storing just a permutation and the other just a set of atoms. In phase 1, it is sufficient to reduce all $\approx_{\mathit{eq}}$-constraints without propagating any freshness constraints. That is, when we reach an abstraction $\alpha \vdash b \in A$, instead of adding $b$ to the set of atoms in the environment and propagating it, we just put a tag $a \#$ on the current node of $t$. At the end of the $\approx_{\mathit{eq}}$-reduction phase, an environment (consisting just of a set of atoms) will be attached to the root of $t$ and some nodes will be tagged with freshness constraints. It remains to propagate the environment with the set of atoms over $t$, adding $a$ to the set each time we reach a node tagged by $\#$.

To obtain an efficient algorithm it is important to avoid doing several freshness checks of the form $(a \# b)_{a \in A}$, which would require time proportional to $|A|$. Instead, we should do just one check $b \in_{\#} A$, which can be done in constant time using an array.

Alpha-equivalence. To check the validity of an $\approx_{\mathit{eq}}$-constraint, after running the core algorithm we have to normalise the problem using the rule $(\alpha)$, as described in Section 3.3.

If the right-hand sides of $\approx_{\mathit{eq}}$-constraints are ground, the core algorithm is sufficient and it is linear (ground case). Otherwise, each application of the rule $(\alpha)$ 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 in which the result of the core algorithm is reduced using a rule that deals with variables occurring multiple times in the pattern (called $\gamma_{\approx}$ in Section 3.4). In the case of linear matching this rule is not needed — the core algorithm is sufficient.

In Section 4 we discussed the implementation of the rule $\gamma_{\approx}$ using an array $S$ indexed by variables and a rule which we called $\mathit{Rl-Check-Subst}$. The construction of $S$ requires the traversal 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 $|M_{\alpha_{0}}| \times \text{MV}(t) + |s| \approx_{\mathit{eq}} t_{1}$ 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 $\mathit{Rl-Check-Subst}$ will compute $S[X_{i} \approx_{\mathit{eq}} t_{i}] \approx_{\mathit{eq}}$ for each constraint $X_{i} \approx_{\mathit{eq}} t_{i}$ in the result of the core algorithm. Phase 1 to 3 are linear in its size and phase 4 has a complexity $\vartheta(|M_{\alpha_{0}}| \times \text{MV}(t_{4}))$, hence at worst quadratic in size in the input of the problem. The worst case complexity arises when phase 4 suspends permutations on all variables, making the output terms bigger than the input ones. Since permutations are bounded in size, in the worst case (an input problem with no permutations but many variables and abstractions), the output of phase 4 may be quadratic in the size of the input.
problem. 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 $\alpha$-equivalence and matching algorithms are linear even using arrays.

Note that the representation of a matching problem or an $\alpha$-equivalence problem using higher-order abstract syntax does saturate the variables (they have to be applied to the set of atoms they can capture). Thus, our algorithms (whether for ground or non-ground, linear or non-linear problems) are all linear in time with respect to the size of the problem represented in higher-order abstract syntax. The table below summarises the results:

<table>
<thead>
<tr>
<th>Case</th>
<th>$\alpha$-equivalence</th>
<th>Matching</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ground</td>
<td>linear</td>
<td>linear</td>
</tr>
<tr>
<td>Non-ground and linear</td>
<td>log-linear</td>
<td>log-linear</td>
</tr>
<tr>
<td>Non-ground and non-linear</td>
<td>log-linear</td>
<td>quadratic</td>
</tr>
</tbody>
</table>

6. Benchmarks

The algorithms described above to check $\alpha$-equivalence and to solve ground matching problems have been implemented in OCAML [18], using arrays.

In Fig. 1, we show benchmarks generated by building random solvable ground problems (i.e., problems that do not lead to $\bot$) and measuring the time taken by the $\alpha$-equivalence and matching algorithms to give the result (marked as $\diamond$ and $+$ in the graph).²

The benchmarks suggest that for problems of similar size, the $\alpha$-equivalence algorithm takes more time than the matching algorithm. This may be explained by the fact that to check $\alpha$-equivalence we need to traverse the whole problem, whereas a full traversal is not always needed to solve a matching problem. Note that, unlike first-order matching, nominal matching might produce freshness constraints that have to be checked by traversing the term.

We give a concrete example below.

Let $s = [a]([a][a](g, X)), t = [b]([a][b](g, f \ [c]b)))$. We use integers to denote atoms and variables. In this case, the problem $s \approx_{\alpha} t$ contains three different atoms, which we write $a = \text{Atm}_1, b = \text{Atm}_2, c = \text{Atm}_3$, two different variables: $X = \text{Var}_1, Y = \text{Var}_2$, and two different function symbols: $g$ and $f$.

To solve the matching problem $s \approx_{\alpha} t$ we call the function match as follows (below we show the command and its result):

```bash
$ ./ltnm match \
  "[1](\text{Var} 2, [1][1]((g ()), \text{Var} 1))" \
  "[2](\text{Atm} 2, [1][2]((g ()), (f \ [3]\text{Atm} 2)))"
```

² The program is available from: www.dcs.kcl.ac.uk/staff/maribel/CANS.
Matching of ‘s’ and ‘t’ with :
-----------------------------
s = \{1\}(Var 2, \{1\}[1]((g ()), Var 1))
t = \{2\}(Atm 2, \{1\}[2]((g ()), (f [3] Atm 2)))

Result :
--------

Substitution :
-------------
\{ Var 2 -> Atm 1,
    Var 1 -> (f [3] Atm 1) \}

Freshness constraints :
----------------------
True

7. Application to rewriting

In this section, we recall the definition of nominal rewriting systems [9] and describe an algorithm for nominal rewriting using nominal matching.

7.1. Nominal rewriting

A nominal rewriting rule is a tuple \((\Delta_I, l, r)\), written \(\Delta_I \vdash l \rightarrow r\), where \(\Delta_I\) is a freshness context and \(l, r\) are nominal terms such that \(V(r, \Delta_I) \subseteq V(l)\).

Intuitively, in a rewriting rule, variables represent unknown terms and the rewrite relation is generated by instantiating the variables in the rules using substitutions. For example, the two rules

\[ a \# X \vdash (\lambda[a]X)Y \rightarrow X \quad \text{and} \quad a \# Y \vdash (\lambda[a]Y)X \rightarrow Y \]

have different syntax but should generate the same rewrite relation. Similarly, atoms in rules can be renamed: We write \(R^{(ab)}\) for that rule obtained by swapping the names \(a\) and \(b\) in the rule \(R\) throughout. For example, if \(R \equiv a \# X \vdash [a]X \rightarrow X\) then \(R^{(ab)} \equiv b \# X \vdash [b]X \rightarrow X\). In general, we write \(R^\pi\) for that rule obtained by applying the permutation \(\pi\) to the atoms in \(R\).

A set of rewrite rules is equivariant when it is closed under \((-)^{(ab)}\) for all atoms \(a\) and \(b\).

A nominal rewriting system \(\mathcal{R}\) is an equivariant set of nominal rewriting rules.

Nominal rewriting rules generate a rewrite relation on “nominal terms in context”. We denote a term \(t\) with a freshness context \(\Delta_t\) by \(\Delta_t \vdash t\). In order to apply a rewrite rule at the root position in \(t\), we use nominal matching. Not only \(l\) has to match \(t\), but also the constraints in \(\Delta_I\) have to be satisfied by \(\Delta_t\). For example, consider the rule \(a \# X \vdash [a](X, Y) \rightarrow (X, [a]Y)\) and let \([a](a, a)\) be the term to rewrite. In this case, even if \([a](X, Y)\) and \([a](a, a)\) match (with the substitution \(\sigma = [X \mapsto a, Y \mapsto a]\)) the rule cannot be applied because \(a\) is not fresh in \(X\sigma\).

Formally, given a nominal rewrite rule \(\Delta_I \vdash l \rightarrow r\) and a term \(t\) with freshness context \(\Delta_t\), we say that \(\Delta_t \vdash t\) matches the left-hand side \(l\) of the rule under the constraints \(\Delta_I\) if and only if the matching problem \(l \approx_{\Delta_I} t\) has a solution \(\sigma, \Delta =_{\Delta_I} \sigma\) and \(\Delta =_{\Delta_I} \sigma\) \(\cup\) \(\text{Env}(\sigma(\Delta_I)) =_{\Delta_{\Delta_I}} \Delta_t\), where \(\sigma(\Delta_I) = [a \# X\sigma \mid a \# X \in \Delta_I]\). The result of applying this rewrite rule to \(\Delta_t \vdash t\) is then the term \(\Delta_t \vdash \sigma\).

Note that since nominal rewriting systems are equivariant, in order to decide whether there exists a rule in the system that matches a term at a given position we need equivariant nominal matching (i.e., nominal matching with respect to variants of the rules obtained by changing the names of the atoms). However, nominal matching is sufficient if the rules are closed. Closed nominal rules are equivalent to the rules used in standard notions of higher-order rewriting (for instance, any Combinatory Reduction System [13] can be defined as a closed nominal rewriting system, see [8]). Equivariant nominal matching, which is an NP-complete problem in general [6], can be avoided for closed rules, simply by using a copy of the rewrite rule where all the variables and atoms are chosen to be different from the ones in the term to be rewritten (this is always possible because nominal rewriting systems are equivariant). We refer the reader to [9] for the formal definition and properties of closed nominal rewriting.
For example, the \( \eta \)-rule in the \( \lambda \)-calculus is usually written informally as a rule scheme
\[
\lambda x. Mx \rightarrow M \quad \text{if } x \notin \text{FV}(M)
\]
We can specify \( \eta \) using a nominal rewriting system defined by the (closed) rule
\[
a \# X \vdash \lambda([a]Ap(X, a)) \rightarrow X
\]
and then use this system to rewrite the term \( \lambda([b]Ap(a, b)) \) in an empty freshness context as follows:

1. We choose a fresh copy of the rewrite rule that does not use any variables or atoms occurring in the term to be rewritten. For instance, we take
\[
c \# X \vdash \lambda([c]Ap(X, c)) \rightarrow X
\]
2. We solve the matching problem:
\[
\lambda([c]Ap(X, c)) \approx_a \lambda([b]Ap(a, b))
\]
which has solution \( ([X \mapsto a], \emptyset) \). Since \( c \# a \) is valid, our term rewrites to the term \( a \), as expected.

7.2. A nominal rewriting algorithm: two improvements on the matching algorithm

To check whether \( \Delta_t \vdash t \) matches \( \Delta_t \vdash l \rightarrow r \) efficiently, we will use a modified version of our matching algorithm. Since checking \( \Delta \approx_u \cup \mathit{Env}(\sigma(\Delta_t)) \approx_u \subseteq \Delta_t \) can be expensive because of the set unions and \( \alpha \)-equivalence problems it involves, we will do these operations on-the-fly, directly in the matching algorithm.

For this, the first modification consists of computing \( \mathit{Env}(\sigma(\Delta_t)) \approx_u \) as we traverse the term, so that it will directly be in \( \Delta_\approx_u \).

Below we use the notation \( \Delta_X \) for the set of atoms in constraints relating to \( X \) in a freshness context \( \Delta \), more precisely:
\[
\Delta_X = \{ a \mid a \# X \in \Delta \}
\]

**Proposition 15.** If for each equation \( X \approx \alpha \xi \otimes t \) at the end of phase 2, we add the set \( \Delta^{-1}_\pi(\Delta_t X) = \{ \xi^{-1}_\pi \cdot a \mid a \# X \in \Delta_t \} \) to the current freshness set \( \xi_A \), this is equivalent to computing \( \mathit{Env}(\sigma(\Delta_t)) \approx_u \cup \Delta_\approx_u \).

In fact, it is sufficient to do this once for each distinct variable \( X \).

**Proof.** By Proposition 3, at the end of phase 2, the problem, if it has a solution, is of the form \( X_i \approx \alpha \xi_i \otimes t_i \). Let \( X_j \approx \alpha \xi_j \otimes t_j \) be one of these constraints. Notice that for each \( i \) such that \( X_i = X_j, \xi_i \pi \cdot t_i \) and \( \xi_j \pi \cdot t_j \) must be \( \alpha \)-equivalent for the problem to have a solution \( \sigma \). In addition to satisfying \( X_i \sigma \approx \alpha (\xi_j \pi \cdot t_j), \sigma \) must also satisfy \( \Delta_t X_j \# X_j \sigma \).

Since \( \Delta_t X_j \# X_j \sigma \) is equivalent to \( \Delta_t X_j \# \xi_j \pi \cdot t_j \), which is in turn equivalent to \( \xi_j^{-1}(\Delta_t X_j) \# t_j \), any solution \( \sigma \) of the original problem is also a solution of:
\[
X_j \approx \alpha (\xi_j \pi \cdot t_j, \xi_j \pi \cdot t_j)
\]
Furthermore, since \( \xi_j \pi \cdot t_j \approx \alpha \xi_j \pi \cdot t_j \), when \( X_i = X_j \), we also have
\[
\Delta_t X_j \# \xi_j \pi \cdot t_j \iff \Delta_t X_j \# \xi_j \pi \cdot t_j
\]
which means we only need to do it once for each distinct variable. \( \Box \)

The second improvement consists of computing \( \Delta \approx_u \subseteq \Delta_t \) directly without computing \( \Delta_\approx_u \). For this, in phase 4, when reaching \( \xi \otimes X \), instead of adding \( \xi_A \# X \) to \( \Delta_\approx_u \) we directly check if \( \xi_A \subseteq \Delta_t X \).

**Proposition 16.** In phase 4, when reaching \( \xi \otimes X \), checking whether \( \xi_A \subseteq \Delta_t X \) is equivalent to adding \( \xi_A \# X \) to \( \Delta_\approx_u \) and then checking \( \Delta \approx_u \subseteq \Delta_t \).

**Proof.** Phase 4 computes \( \Delta_\approx_u \) by generating constraints \( \xi_A \# X \) for each \( \xi \otimes X \) in the problem. Therefore, checking \( \Delta \approx_u \subseteq \Delta_t \) is equivalent to checking whether every \( \xi_A \# X \subseteq \Delta_t \), which is equivalent to \( \xi_A \subseteq \Delta_t X \). \( \Box \)

These two modifications transform the previous matching algorithm into an algorithm to match terms in context. The new algorithm consists of Phase 1 and 2 unchanged, then Phase 2’ performing the first modification, then Phase 3 unchanged and finally Phase 4’ performing the second modification.

**Remark 4.** Since we no longer need to copy freshness sets, we can always use mutable arrays to represent \( \xi_A \). Unfortunately we still need to copy permutations \( \xi_\pi \), so in the following we will use mutable arrays for sets, and functional maps for permutations. We will represent \( \Delta_t \) using an array indexed by variables, such that the element \( X \) in the array will contain the list of atoms that should be fresh for \( X \) according to \( \Delta_t \).
7.3. Complexity of nominal rewriting

We start by defining the size of a matching problem for terms in context.

**Definition 15.** The size of a matching problem \( Pr = \Delta t \vdash l \approx_{\Delta} \Delta t \vdash t \) is defined as
\[
|Pr| = |\Delta t| + |l| + |\Delta t| + |t|
\]
where \(|l|\) and \(|t|\) are computed as indicated in Definition 14, and for a freshness context \( \Delta \) we define \(|\Delta| = \Sigma_{X \in \Delta} |X|\).

**Proposition 17.** The total cost of the algorithm to solve the matching problem for terms in context:
\[
\Delta t \vdash l \approx_{\Delta} \Delta t \vdash t
\]
is at most
\[
\vartheta \left( (|\Delta t| + |l| + |t|) \times \log |A| + m \times |MVar(t)| \right)
\]
where \( A \) is the set of atoms occurring in the problem, \( m = \max_{X} (|\Delta_{t}X|) \) and \( MVar(t) \) is the multiset of variable occurrences in \( t \).

**Proof.** Phase 1, Phase 2 and Phase 3 are unchanged, their cost is the same as before.

Phase 2' adds \( \xi^{-1}_{\Delta} (\Delta_{t}X) \) to the current freshness set, once for each variable \( X \). Freshness sets being coded as mutable arrays and permutations as functional maps, it takes \( \vartheta (\log |A|) \) time to access each atom \( a \) in \( \Delta_{t}X \) (coded as a list of atoms), compute its image by \( \xi^{-1}_{\Delta} \) and add it to \( \xi_{A} \). The cost of Phase 2' is therefore \( \vartheta (|\Delta t| \times \log |A|) \).

Phase 4', when reaching \( \xi \triangleleft X \), rewrites it into \( \xi_{A} \cdot X \) and checks whether \( \xi_{A} \subseteq \Delta_{t}X \). The former is done in constant time thanks to the use of functional maps. The latter, if done in a naive way, would take time proportional to \( |A| \) since \( \xi_{A} \) is represented as an array of size \( |A| \). However, checking \( \xi_{A} \subseteq \Delta_{t}X \) is equivalent to checking \( |\Delta_{t}X \cap \xi_{A}| = |\{a \mid a \in \xi_{A}\}| \). We call this number \( s_{\xi_{A}} \). We can, as we did for permutations, compute this number when \( \xi_{A} \) is created and then update it in constant time on every update of \( \xi_{A} \). In this way, we only need to compute the number of atoms in \( \Delta_{t}X \) that are also in \( \xi_{A} \) and compare it with \( s_{\xi_{A}} \), which can be done in time proportional to \( \Delta_{t}X \) (because \( \Delta_{t}X \) is a list of atoms and \( \xi_{A} \) an array). Thus, for any \( X \), we can bound the cost by \( m \), so in the worst case the total cost is \( m \times MVar(t) \).

Summarising:

Phase 1: \( \vartheta (|l| \times \log |A|) \)

Phase 2: \( \vartheta (|l| \times \log |A|) \)

Phase 2': \( \vartheta (|\Delta t| \times \log |A|) \)

Phase 3: \( \vartheta (|t| \times \log |A|) \)

Phase 4': \( \vartheta (m \times |MVar(t)|) \)

7.3.1. Special cases

First-order matching is a particular case of nominal matching, and it is interesting to see that the rewriting algorithm specified above behaves as expected when the terms involved are ground or first-order.

**Proposition 18.** If \( t \) is ground or if \( \Delta t \) is empty, the complexity of the algorithm is at most \( \vartheta (|\Delta t| + |l| + |t|) \times \log |A|) \).

**Proof.** If \( t \) is ground, \( |MVar(t)| = 0 \) and if \( \Delta t \) is empty, \( m = 0 \).

**Proposition 19.** For first-order terms, where \( t \) is ground, the algorithm is linear in time.

**Proof.** In this case there are neither variables \((|MVar(t)| = 0)\) nor atoms \((|A| = 0)\).

7.4. Examples of nominal rewriting

We give below an example of reduction, using our implementation of the nominal rewriting algorithm, to illustrate the problem of matching terms in context.

Let \( t = [a][a][b][a,b] \) be the term to rewrite. First consider the rewriting rule
\[
b \# X \vdash [b][b \# X] \rightarrow [b]X
\]
In this case, there are two positions in \( t \) where we could try to apply the rule: at the root position, or in the subterm \([b][a,b] \). The latter does not match with \([b][b \# X] \), since \( a \) is a free atom but \( b \) is bound in the pattern. To rewrite the term at the root position we need to solve the matching problem: \( \vartheta \vdash t \approx_{\Delta} b \# X \vdash [b]X \).
Matching $t$ with $[b](b, X)$ gives the substitution $[X \mapsto [a](b,a)]$, but the freshness context $b \neq X$ is not satisfied, therefore the term cannot be rewritten with this rule.

Consider instead the rule $c \neq X \vdash [b](b,X) \rightarrow [b]X$. The matching problem $\emptyset \vdash t \approx_\alpha c \neq X \vdash [b](b,X)$ gives the substitution $[X \mapsto [a](b,a)]$, which satisfies the freshness context. The resulting term is $[b][a](b,a)$, as our nominal rewriting tool [2] shows (atoms and variables are coded as follows: $a = a : 1$, $b = a : 2$, $c = a : 3$, $X = v : 1$).

$./hnt$
Please enter the term to rewrite: $[a:1](a:1,[a:2](a:1,a:2))$

Term = $[a:1](a:1,[a:2](a:1,a:2))$
Path = /

$\Rightarrow$ Rewrite with $[a:3]\#v:1 \vdash [a:2](a:2,v:1) \rightarrow [a:2]v:1$

Term = $[a:2][a:1](a:2,a:1)$
Path = /

8. Conclusions

We described an algorithm to solve matching problems modulo $\alpha$-equivalence which is linear in time and space when the right-hand side terms are ground. Matching modulo $\alpha$-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. There is also a direct link between nominal matching and matching of higher-order patterns (see [5,14]) so efficient implementations of nominal matching also have applications in programming languages that are based on matching of higher-order patterns.

Our algorithm solves matching problems with linear or non-linear patterns, and with ground or non-ground right-hand sides. When the patterns are not linear, we use an $\alpha$-equivalence test. The matching and $\alpha$-equivalence algorithms rely on the use of an environment to represent mappings between atoms, and instead of renamings, we use swappings, which are calculated as the terms are traversed. To our knowledge, this is the most efficient algorithm available to solve matching problems modulo $\alpha$-equivalence.

We have also applied the matching algorithm to rewrite terms using nominal rewriting rules. In this case, we need to match terms with freshness contexts. We have shown that with two small modifications we can obtain an efficient nominal matching algorithm for terms in context.

The algorithms described in this paper have been implemented. We are currently deploying the matching algorithms in a nominal rewriting tool.

Acknowledgments

This work has been partially funded by the EPSRC grant “CANS” (EP/D501016/1). We thank James Cheney, Jamie Gabbay, Andrew Pitts, François Pottier and Christian Urban for useful discussions on the topics of this paper.

References


