Nominal Rewriting ⋆

Maribel Fernández and Murdoch J. Gabbay

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

Abstract

Nominal rewriting is based on the observation that if we add support for alpha-equivalence to first-order syntax using the nominal-set approach, then systems with binding, including higher-order reduction schemes such as lambda-calculus beta-reduction, can be smoothly represented.

Nominal rewriting maintains a strict distinction between variables of the object-language (*atoms*) and of the meta-language (*variables* or *unknowns*). Atoms may be bound by a special abstraction operation, but variables cannot be bound, giving the framework a pronounced first-order character, since substitution of terms for variables is not capture-avoiding.

We show how good properties of first-order rewriting survive the extension, by giving an efficient rewriting algorithm, a critical pair lemma, and a confluence theorem for orthogonal systems.

Key words: Binders, α-conversion, first and higher-order rewriting, confluence.

⋆ This work has been partially funded by EPSRC grant EP/C517148/1 "Rewriting Frameworks".

Email address: Maribel.Fernandez@kcl.ac.uk, Murdoch.Gabbay@gmail.com (Maribel Fernández and Murdoch J. Gabbay).

Contents

1 Introduction

This is a paper about rewriting in the presence of α-conversion. 'Rewriting', or in full 'the framework of Term Rewriting Systems' (**TRS**), is a framework for specifying and reasoning about logic and computation. Usually, if the reader's favourite formal system can be described by syntax trees (also called **terms**), then any notion of dynamics (deduction and evaluation for example) can probably be captured by a suitable collection of rewrite rules. For example (we are more formal later):

(1) Assume 0-ary term-formers (**constants**) S and K, and a binary term-former $\circ$ which we write infix. Then the rewrite rules

$$((\mathsf{S} \circ X) \circ Y) \circ Z \to (X \circ Y) \circ (X \circ Z) \quad \text{and} \quad (\mathsf{K} \circ X) \circ Y \to X$$

define a rewrite system for **combinatory algebra** (or **combinatory logic**) [6]. X, Y, and Z are **unknowns**, which can be instantiated to any term. So from this rewrite system we may deduce $(\mathsf{K} \circ \mathsf{S}) \circ \mathsf{K} \to \mathsf{S}$ and indeed $(\mathsf{K} \circ X) \circ Y \to X$.

(2) Assume constants *very*, *damn*, and *whitespace*, and rewrite rules

$$\mathit{very} \to \mathit{damn} \quad \text{and} \quad \mathit{damn} \to \mathit{whitespace}.$$

This implements a rewrite system due to Mark Twain. [1]

However, in the presence of binding, a notion of rewriting on pure abstract syntax trees is not as useful as we might like. For example, here are informal descriptions of the α- and η-reduction rules of the untyped λ-calculus [6]:

$$\begin{array}{ll} \lambda x.s \to \lambda y.s[x \mapsto y] & \text{if } y \notin \mathit{fv}(s) \\ \lambda x.(sx) \to s & \text{if } x \notin \mathit{fv}(s) \end{array}$$

Note the *freshness* side-conditions on the right: $\mathit{fv}(s)$ denotes the set of free variables of s.

The β-reduction rule $(\lambda x.s)t \to s[x \mapsto t]$ raises some issues too: we need to define the capture-avoiding substitution $s[x \mapsto t]$, and this involves more freshness side-conditions.

These rules introduce nondeterminism and make rewriting conditional. Experience also shows that they are difficult to reason about and pose specific implementation problems.

[1] Substitute "damn" every time you're inclined to write "very"; your editor will delete it and the writing will be just as it should be. – Mark Twain

One answer is to take rules creating these issues, for example α, as *equalities* on terms. We say that 'names and binding are relegated to the meta-level'. A lot of effort has gone into developing systems along these lines. Several notions of rewriting modulo an equational theory have been developed, but none that can deal specifically with α-equivalence (where equivalence classes of terms are infinite). Combinatory Reduction Systems (**CRS**) [31], Higher-order Rewrite Systems (**HRS**) [33], Expression Reduction Systems (**ERS**) [30,29], and the rewriting calculus [13,14], combine first-order rewriting with a notion of bound variable, and rewriting rules work on α-equivalence classes of terms. In these systems the λ-calculus can be defined as a particular rewrite system with one binder: λ-abstraction.

A benefit of such systems is that non-capture-avoiding substitution can be implemented using capture-avoiding substitution. For example, the effect of $(\lambda a.X)[X \mapsto a]$ (where here substitution for X does not avoid capture) can be obtained in a higher-order system as $(\lambda a.fa)[f \mapsto \lambda b.b]$. Thus, we do not lose any power by taking terms up to α-equivalence (which forces all substitution to be capture-avoiding) so long as we also take them up to β-equivalence.

Another approach is to bite the bullet and specify everything to do with α, β, η, and so on, completely explicitly. To make this more manageable, substitution is introduced as a term-former, which does at least make reasoning on these equivalences susceptible to term-level inductions on syntax and so on. Explicit substitution systems have been defined for the λ-calculus (e.g. [1,32,15]) and more generally for higher-order rewrite systems (e.g. [35,9]) with the aim of specifying the higher-order notion of substitution as a set of first-order rewrite rules. In most of these systems variable names are replaced by de Bruijn indices to make easier the explicitation of α-conversion, at the expense of readability. Explicit substitution systems that use names for variables (see for example [23,8]) either restrict the rewriting mechanism to avoid cases in which α-conversion would arise (for instance using weak reduction, or closed reduction) or use Barendregt's convention (all bound variables in a term have fresh names, different from the free variables; and this property is assumed to be maintained by reduction) to avoid addressing the problem of α-conversion.

In this paper we present a framework for rewriting based on a different way of slicing these issues. We maintain a strict distinction between object-level variables (we write them a, b, c and call them **atoms**), which can be abstracted but behave similarly to constants (whence the 'nominal', for 'name' in "nominal rewriting systems", henceforth abbreviated to **NRS**) — and meta-level variables (we write them X, Y, Z and may call them **unknowns**), which are first-order in that there are no binders for them and substitution does not avoid capture of free atoms (we may refer to our system as 'first-order', as opposed to higher-order systems which quotient terms up to α-equivalence, and for which substitution is capture-avoiding and may involve β-reductions).

Our approach is based on the work reported in [36,24,41]. We deal with α-conversion using a small logic for deriving a relation 'are α-equivalent', in a syntax-directed manner (this gives us the great benefit that we can reason by induction on syntax and/or on the *derivation* that two terms are α-equivalent); we deal with the *freshness* side-conditions mentioned above by introducing an explicit **freshness relation** between atoms and unknowns, written as $a\#X$ (read "a is fresh for X"; in fact, we write $a\#t$ where t is any term, but this is derived by simple induction on syntactic structure). Then, as we shall see, β-, η- and other similar reduction rules can be easily defined, as rewrite rules.

We can see nominal terms as first-order terms, with a definition of α-equality which explicitly supports our intuitive notions of 'meta-variable' and 'freshness condition'. It combines many of the conveniences of higher-order techniques (smooth handling of β- and similar reduction rules) with the syntax-directed simplicity of first-order techniques (a simple notion of substitution, decidable unification).

Consistent with previous work on nominal logic and unification [36,24,41], we call atoms the names that can be bound and reserve the words variable and unknown for the identifiers that cannot be bound (known as variables and metavariables respectively in CRSs and ERSs, or bound and free variables in HRSs). We leave implicit the dependencies between variables and names as it is common practice in informal presentations of higher-order reductions. More precisely, variables in NRSs have arity zero, as in ERSs (but unlike ERSs, substitution of atoms for terms is not a primitive notion in NRSs). For example, the β-reduction rule and the η-expansion rule of the λ-calculus are written as:

$$\begin{array}{lll} & app(\lambda([a]M), N) & \to subst([a]M, N) \\ a\#X \;\vdash & X & \to \lambda([a]app(X, a)) \end{array}$$

where the substitution in the β-rule is a term-former, which has to be given meaning by rewrite rules.

To summarise, the main contributions of this paper are:

(1) We formulate a notion of rewriting on nominal terms which behaves as first-order rewriting but uses matching modulo α-conversion. In particular, substitution remains a first-order notion: we deal with α-conversion without introducing meta-substitutions and β-reductions in our meta-language (in contrast with standard notions of higher-order rewriting, which rely on meta-substitutions and/or β-reductions in the substitution calculus). Consequently in some cases we need freshness assumptions in terms and rewrite rules; these will be taken into account in the matching algorithm. We use **nominal matching** [41] to rewrite terms.

Selecting a nominal rewrite rule that matches a given term is an NP-

complete problem in general [12]. However, by restricting to **closed** rules we can avoid the exponential cost: nominal matching is polynomial in this case. Closed rules are, roughly speaking, rules which preserve abstracted atoms during reductions; in particular, closed rules do not contain free atoms, hence their name. CRSs, ERSs, and HRSs impose similar conditions on rules, by definition (ERSs impose a condition on the substitution used to match a left-hand side, which corresponds to our notion of closed rules). Closed rules are very expressive: [2] see [21] for an encoding of CRSs using closed nominal rules. Translations between CRSs, HRSs and ERSs have already been defined (see [37]).

(2) We prove a Critical Pair Lemma which ensures that closed nominal rewriting rules which do not introduce critical pairs are locally confluent, and a confluence result for orthogonal systems (i.e. NRSs where rules have linear left-hand sides without superpositions). Similar results have been proved for CRSs and HRSs (see [31,33]).

Related work First-order rewriting systems and the λ-calculus provide two useful notions of terms and reduction, and both formalisms have been used as a basis to develop specification and programming languages. However, in both cases the expressive power is limited (although for different reasons): first-order rewrite systems do not provide support to define binding operators, and there are useful operations which cannot be encoded in the λ-calculus (see for instance [5], where it is shown that the rules for surjective pairing cannot be encoded in the λ-calculus). These observations motivated the search for more general formalisms combining the power of first-order term rewriting with the binding capabilities of the λ-calculus. Our work can be seen as part of this effort. In the rest of the introduction we will compare nominal rewrite systems with the rewrite systems with binders that have been defined previously.

(1) Algebraic λ-calculi, which can be typed [10,11,28,3,4] or untyped [17], combine the β-reduction rule of the λ-calculus with a set of term rewriting rules. They use capture-avoiding substitution in β-reductions, and first-order matching and substitution for term rewriting rules. They are more expressive than either the λ-calculus or first-order rewriting, but it was observed that properties of the latter formalisms are not automatically inherited by the combination. The papers cited above use types, or syntactical conditions in the rewrite rules, or both, to characterise subsystems that preserve confluence or termination for instance.

Algebraic λ-calculi can be defined as nominal rewrite systems, but as for the λ-calculus, the notion of capture-avoiding substitution has to be

[2] An argument could be made that they are *the* correct notion of nominal rewrite rule, though we have a weaker well-behavedness condition we call **uniformity**, which is also relevant.

explicitely defined using nominal rewrite rules.

(2) Higher-order rewriting systems (e.g. CRSs [31], HRSs [33], ERSs [30,29], HORSs [37]) extend first-order rewriting to include binders using higher-order substitutions and higher-order matching. In contrast with algebraic λ-calculi (which use first-order matching), binders are allowed in left-hand sides of higher-order rewrite rules. NRSs are related to these since nominal rules may also have binders in left-hand sides, however, nominal rewriting does not use higher-order matching; instead, it relies on nominal matching (which takes care of α-equality). Although there is no 'official' definition of higher-order rewrite rule, it is generally acknowledged that CRSs, ERSs and HRSs (although using different presentations) define a canonical higher-order rewrite format (see [31]). The subclass of *closed* NRSs is also canonical in this sense. However, NRSs are more expressive than standard higher-order rewrite systems, we will give examples.

(3) Although NRSs were not designed as explicit substitution systems, they are at an intermediate level between standard higher-order rewriting systems and their explicit substitution versions (e.g. [35,9]), which implement in a first-order setting the substitution operation together with α-conversions using de Bruijn indices. Compared with the latter, NRSs are more modular: a higher-order substitution is decomposed into a first-order substitution and a separate notion of α-equality (a design idea borrowed from Fresh-ML [38]). Also, from a (human) user point of view, it is easier to use systems with variable names than systems with indices. The disadvantage is that nominal rewriting is not just first-order rewriting, therefore we cannot directly use all the results and techniques available for first-order rewriting. However, nominal rewriting turns out to be sufficiently close to first-order rewriting that it shares many of its desirable and convenient properties: efficient matching, a critical pair lemma, and a confluence result for orthogonal systems.

(4) Hamana's Binding Term Rewriting Systems (BTRS) [27] also extend first-order rewriting to include binders and α-equality, but use a de Bruijn notation. The main difference with Nominal Rewriting is that BTRSs use a containment relation that indicates which free atoms occur in a term (as opposed to a freshness relation which indicates that an atom does not occur free in a term). Not surprisingly, the notions of renaming and variants play an important rôle in BTRSs, as do swappings and equivariance in NRSs. In other words, when free atoms occur in rules, we have to consider all the (infinite) variants that can be obtained by renaming the free atoms. Selecting a rewrite rule that matches a given term is then NP, but we have characterised a class of NRSs for which matching is efficient and we conjecture the BTRS-matching algorithm is efficient in this case too.

(5) NRSs, with their use of freshness contexts in rules, could be seen as a form of conditional rewriting systems (see [7]), albeit with matching modulo α. Takahashi's λ-calculus with conditional rules [39] is a closely

related formalism: it is a higher-order rewriting framework in the sense that rules can include binders (there is a distinction between object-level and meta-level variables). As in NRSs, substitution of metavariables for terms is not capture-avoiding, but terms are defined as α-equivalence classes of trees, and capture-avoiding substitution is a primitive notion as in ERSs. The requirement that rule schemes are closed under capture-avoiding substitution means that only closed NRSs can be expressed. On the other hand, the conditions on the rewrite rules in conditional λ-calculus systems are arbitrary (not just freshness predicates as in NRSs). We discuss in Section 9 extensions of NRS which can deal with more general contexts in rules.

This paper is an updated and extended version of [21]. Other work exists extending nominal rewriting in various ways [19] and considering types [20].

Overview of the paper Section 2 presents nominal signatures, terms and substitutions. In Section 3 we define α-equivalence of nominal terms and give an algorithm to check it, which is used as a basis to design a unification algorithm in Section 4. Rewriting is defined in Section 5. In Section 6 we define uniform systems (a class of nominal rewriting systems which are well-behaved with respect to α-equivalence), and prove the Critical Pair Lemma for uniform rules. In Section 7 we prove that orthogonality is a sufficient condition for confluence of uniform rewriting. A further restriction on rewrite rules is used in Section 8 to obtain an efficient implementation of nominal rewriting. In Section 9 we briefly discuss some extensions of nominal rewriting (with sorts, and with more expressive contexts for rewrite rules). We conclude the paper in Section 10.

2 Nominal terms

2.1 Signatures and terms

A **nominal signature** Σ is a set of **term-formers** typically written f (though we do try to give them suggestive names in examples). For instance, a nominal signature for a fragment of ML has term-formers:

$$\texttt{app} \quad \texttt{lam} \quad \texttt{let} \quad \texttt{letrec}$$

and a nominal signature for the π-calculus is given by

$$\texttt{in} \quad \texttt{out} \quad \texttt{par} \quad \texttt{rep} \quad \nu$$

In order to define operations over the syntax of ML for instance, we will need:

- The notion of a term containing unknown terms represented by variables which can be instantiated, so that we can represent a schema of rewrites by a single rewrite rule. We call these **(meta-level) unknowns**.
- The notion of an object-level variable symbol of the ML language, so that we can directly represent the variable structure of the object language, and define variable lookup, open terms, patterns, and λ-abstractions. We call these **(object-level) variable symbols** or **atoms**.

Of course, we would need the same if we wanted to model First-Order Logic, the λ-calculus, or any other syntactic system which mentions variable symbols.

Fix a countably infinite set $\mathcal{X}$ of **term variables** X, Y, Z; these represent meta-level unknowns. Also, fix a distinct countably infinite set $\mathcal{A}$ of **atoms** a, b, c, n, x; these represent object-level variable symbols. Consistent with later notation for terms, we write $a \equiv a$ and $X \equiv X$ to denote syntactic identity of unknowns and atoms. We assume that Σ, $\mathcal{X}$ and $\mathcal{A}$ are pairwise disjoint.

A **swapping** is a pair of atoms, which we write $(a\ b)$. **Permutations** π are lists of swappings, generated by the grammar:

$$\pi ::= \mathtt{Id} \mid (a\ b)\pi.$$

We usually omit the last `Id` when we write the list of swappings that define a permutation. We call `Id` the **identity permutation**. We call a pair of a permutation π and a variable X a **moderated variable** and write it $\pi{\cdot}X$. We say that π is **suspended** on X. We may write π^{-1} for the permutation obtained by reversing the list of swappings in π. For example if $\pi = (a\ b)(b\ c)$ then $\pi^{-1} = (b\ c)(a\ b)$ and $\pi^{-1}(a) = c$. We denote by $\pi \circ \pi'$ the permutation containing all the swappings in π followed by those in π'.

Remark: $\pi{\cdot}X$ represents an unknown term X in which some atoms must be renamed (e.g. by some α-equivalence taking place elsewhere in the term) — but because we do not yet know what X is, the renaming sits *suspended* outside. When we come to define substitution of terms for unknowns $[X{\mapsto}s]$, we shall see that permutations 'unsuspend' and go into s. For example, using the notation we introduce in a moment, $(\pi{\cdot}X)[X{\mapsto}(Y,Y)] \equiv (\pi{\cdot}Y, \pi{\cdot}Y)$. $*$

Definition 1 ***Nominal terms****, or just* ***terms*** *for short, are generated by the grammar:*

$$s, t ::= a \mid \pi{\cdot}X \mid (s_1, \ldots, s_n) \mid [a]s \mid (f\ t)$$

Terms are called respectively **atoms**, **moderated variables** (or just **variables** for short), **tuples**, **abstractions** and **function applications**.

Note that X is not a term, but $\mathtt{Id}{\cdot}X$ is. We abbreviate $\mathtt{Id}{\cdot}X$ as X when there is no ambiguity. In the clause for tuples we call n the **length** of the tuple. If $n = 0$ we have the **empty tuple** $()$. We omit the brackets when n is 1, if there is no ambiguity. If f is applied to the empty tuple we may write $f()$ as just f.

We write $V(t)$ for the set of variables occuring in t. **Ground terms** are terms without variables, that is $V(t) = \emptyset$. A ground term may still contain atoms, for example a is a ground term and X is not.

An abstraction $[a]t$ is intended to represent t with a bound, as in the syntax $\lambda a.t$ (from the λ-calculus) and $\nu a.P$ (from the π-calculus). Accordingly we call occurrences of a **abstracted** (or bound) and unabstracted occurrences **unabstracted** (or free). We do *not* work modulo α-conversion of abstracted atoms, so syntactic identity $\equiv$ is *not* modulo α-equivalence; for example, $[a]a \not\equiv [b]b$. α-equivalence $\approx_\alpha$ is a logical notion constructed on top of $\equiv$ using a notion of context which we shall define soon.

Example 2 Recalling the signature for ML mentioned previously, the following are nominal terms (and the last one is ground):

$$\mathtt{app}(\mathtt{lam}([a]a), X) \quad (\mathtt{lam}([a]\mathtt{lam}([b]a)), Y) \quad \mathtt{let}([a]a, a)$$

We define the following sugar:

- Sugar $\mathtt{app}(s, t)$ to $s\ t$.
- Sugar $\mathtt{lam}([a]s)$ to $\lambda[a]s$.
- Sugar $\mathtt{let}([a]s, t)$ to let $a = t$ in s.
- Sugar $\mathtt{letrec}([f]([a]t, s))$ to let $fa = t$ in s.

There is nothing to stop us writing $\mathtt{app}([a]s)$ if we like, because we have, for simplicity, introduced no notion of arity or *sort system*. We discuss this later in Section 9, see also [20].

2.2 Substitution and swapping

Substitutions (of variables for terms) are used to instantiate unknowns and to represent matching or unification solutions. In systems with binders, substitution is not so easy to define because it should avoid capture of bound variables, so α-conversions may be needed. α-conversion is in turn traditionally defined by using a 'simpler' kind of substitution — renaming, that is, substitution of atoms by atoms — where an atom is replaced by a *fresh* one. Instead of using renamings, nominal techniques define α-equivalence and freshness using *swappings*. Intuitively, a swapping $(a\ b)$ is a special kind of first-order substitution

which replaces simultaneously a by b and b by a in the syntax of the term, suspending on variables. Swappings have better commutation properties (with substitutions and with α-equivalence) than renamings — no side conditions are required. We will return to this point later.

As discussed $\pi{\cdot}X$ represents an unknown term with some swappings waiting to happen. This is reflected in the definition of the action of permutations and substitutions on terms below, denoted $\pi{\cdot}t$ and $t[X{\mapsto}s]$ respectively.

Definition 3 (Permutation) *The action of a permutation π on a term t is defined by induction on the number of swappings in π:*

$$\mathtt{Id}{\cdot}t = t \qquad (a\ b)\pi{\cdot}t = (a\ b){\cdot}(\pi{\cdot}t)$$

where

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

Here, a, b, c, n are any pairwise different atoms.

For example, $(a\ b){\cdot}\lambda[a]\lambda[b]abX = \lambda[b]\lambda[a]ba(a\ b){\cdot}X$.

Note that although $(a\ b)$ and $(b\ a)$ are different swappings, they have the same action on terms. We will show (see Lemma 19) that two permutations with the same action are logically undistinguishable,.

Definition 4 (Substitution) *A **substitution** is generated by the grammar*

$$\sigma ::= \mathtt{Id} \mid [X{\mapsto}s]\sigma.$$

We write substitutions postfix and write $\circ$ for composition: $t(\sigma\circ\sigma') \equiv (t\sigma)\sigma'$.

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

σ acts on terms elementwise in the natural way:

$$t\mathtt{Id} \equiv t \qquad t[X{\mapsto}s]\sigma \equiv (t[X{\mapsto}s])\sigma.$$

Remark: There is no primitive notion of substitution of a term for an atom in Nominal Rewriting, since some languages have variables which do not represent terms (e.g. the π-calculus, which only has variable-for-variable renaming,

or a language with global state which may have location variables and a notion of generating new location variables, but no notion of replacing one location by another). Various different kinds of substitution on atoms can be efficiently implemented by rewrite rules — but substitution of terms for *unknowns* X, Y, Z is primitive since we need it to express matching and thus rewriting. $*$

Note that $t[X{\mapsto}s]$ really does replace every X in t by s in a completely unimaginative way — in particular $([a]t)[X{\mapsto}s] \equiv [a](t[X{\mapsto}s])$ does *not* avoid capture of a in s by the abstraction — except for the clause $(\pi{\cdot}X)[X{\mapsto}s] \equiv \pi{\cdot}s$, where a suspended permutation becomes 'active' and acts on s. Permutations act top-down and accumulate on moderated variables whereas substitutions act on the variable symbols in the moderated variables. These observations are the core of the next lemma.

Lemma 5 *Substitution and permutation commute:* $\pi{\cdot}(s\sigma) \equiv (\pi{\cdot}s)\sigma$.

Proof By induction on s: The property is trivial for atoms since they are not affected by substitutions. For moderated variables the property follows directly from the definition of substitution. The cases of tuples and function applications are easily dealt with by the induction hypotheses. The only interesting case is abstraction: $(\pi{\cdot}[a]s)\sigma \equiv ([\pi{\cdot}a](\pi{\cdot}s))\sigma$ by Definition 3, and $([\pi{\cdot}a]\pi{\cdot}s)\sigma \equiv [\pi{\cdot}a]((\pi{\cdot}s)\sigma)$ by Definition 4. Now by induction we obtain $[\pi{\cdot}a]\pi{\cdot}(s\sigma)$ which is $\pi{\cdot}[a](s\sigma)$ by Definition 3. $*$

Remark: In contrast with the Substitution Lemma [6], in the lemma above we do not need to compose substitutions and there are no free variable side-conditions. $*$

3 Alpha-equivalence

The notion of 'fresh variable' plays an important rôle in the definition of α-equivalence. We will introduce a **freshness** predicate $\#$ and an **alpha-equality** predicate $\approx_\alpha$:

- $a\#t$ intuitively 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 $a\#s, b\#s$.
- $s \approx_\alpha t$ intuitively means that s and t are α-equivalent.

Syntactic equality $s \equiv t$ is a structural (rather than logical) fact. Intuitively, in the absence of unknowns $a\#s$ and $s \approx_\alpha t$ are also structural facts — to check $a\#s$ for example we just check that every a in s occurs under an abstractor. However, in the presence of unknowns both predicates may depend

on assumptions $a\#X$ about what will get substituted for the unknowns (the simplest example: we may derive $a\#X$ if we assume $\ldots a\#X$). Formally, we define $\#$ and $\approx_\alpha$ by a logical system. We use $\#$ in the definition of $\approx_\alpha$, which expresses the 'freshness side-conditions' mentioned in the introduction.

Definition 6 ***Constraints*** *are generated by the grammar*

$$P, Q, C ::= a\#t \mid s \approx_\alpha t$$

and specified by a system of natural-deduction rules as follows (a, b are any pair of distinct atoms):

$$\frac{}{a\#b}\ (\#\mathbf{ab}) \qquad \frac{a\#s}{a\#fs}\ (\#\mathbf{f}) \qquad \frac{a\#s_1\ \cdots\ a\#s_n}{a\#(s_1,\ldots,s_n)}\ (\#\mathbf{tup})$$

$$\frac{}{a\#[a]s}\ (\#\mathbf{absa}) \qquad \frac{a\#s}{a\#[b]s}\ (\#\mathbf{absb}) \qquad \frac{\pi^{\text{-}1}\cdot a\#X}{a\#\pi\cdot X}\ (\#\mathbf{X})$$

To define $\approx_\alpha$ we use the ***difference set*** *of the two permutations:*

$$ds(\pi,\pi') \stackrel{\text{def}}{=} \left\{ n \;\middle|\; \pi\cdot n \neq \pi'\cdot n \right\}$$

In the rules defining $\approx_\alpha$ below, $ds(\pi,\pi')\#X$ denotes the set of constraints: $\{n\#X \mid n \in ds(\pi,\pi')\}$.

$$\frac{}{a \approx_\alpha a}\ (\approx_\alpha\mathbf{a}) \qquad \frac{ds(\pi,\pi')\#X}{\pi\cdot X \approx_\alpha \pi'\cdot X}\ (\approx_\alpha\mathbf{X})$$

$$\frac{s_1 \approx_\alpha t_1\ \cdots\ s_n \approx_\alpha t_n}{(s_1,\ldots,s_n) \approx_\alpha (t_1,\ldots,t_n)}\ (\approx_\alpha\mathbf{tup}) \qquad \frac{s \approx_\alpha t}{fs \approx_\alpha ft}\ (\approx_\alpha\mathbf{f})$$

$$\frac{s \approx_\alpha t}{[a]s \approx_\alpha [a]t}\ (\approx_\alpha\mathbf{absa}) \qquad \frac{(b\ a)\cdot s \approx_\alpha t \quad b\#s}{[a]s \approx_\alpha [b]t}\ (\approx_\alpha\mathbf{absb})$$

Remark: Rule $(\approx_\alpha\mathbf{absb})$ is equivalent to a rule with premisses $s \approx_\alpha (a\ b)\cdot t \quad a\#t$ $*$

Example 7 We can derive $a\#((a\ b)\cdot X, (b\ c)\cdot Y)$ from assumptions $a\#Y, b\#X$, using the fact that $(b\ c)\cdot a \equiv a$.

$$\frac{\dfrac{b\#X}{a\#(a\ b)\cdot X}\ (\#\mathbf{X}) \qquad \dfrac{a\#Y}{a\#(b\ c)\cdot Y}\ (\#\mathbf{X})}{a\#((a\ b)\cdot X, (b\ c)\cdot Y)}\ (\#\mathbf{tup})$$

We can also derive $a\#(X,[a]Y)$ from $a\#X$, and $a\#f\ a$ from $a\#a$:

$$\dfrac{a\#X \quad \dfrac{}{a\#[a]Y}\,(\#\mathbf{absa})}{a\#(X,[a]Y)}\,(\#\mathbf{tup}) \qquad \dfrac{a\#a}{a\#fa}\,(\#\mathbf{f})$$

Also, we can deduce $(a\ b)\cdot X \approx_\alpha X$ from assumptions $a\#X$ and $b\#X$, and we also have as expected $[a]a \approx_\alpha [b]b$ and, using sugar from the end of §2.1, we can prove $\lambda[f]\lambda[x]fxX \approx_\alpha \lambda[x]\lambda[f]xfX$ provided $f\#X$ and $x\#X$ (assuming f and x are atoms):

$$\dfrac{a\#X \quad b\#X}{(a\ b)\cdot X \approx_\alpha X}\,(\approx_\alpha\mathbf{X}) \qquad \dfrac{\dfrac{}{b \approx_\alpha b}\,(\approx_\alpha\mathbf{a}) \qquad \dfrac{}{b\#a}\,(\#\mathbf{ab})}{[a]a \approx_\alpha [b]b}\,(\approx_\alpha\mathbf{absb})$$

$$\dfrac{\dfrac{}{x \approx_\alpha x}\,(\approx_\alpha\mathbf{a}) \quad \dfrac{\dfrac{}{f \approx_\alpha f}\,(\approx_\alpha\mathbf{a}) \quad \dfrac{x\#X \quad f\#X}{(x\ f)\cdot X \approx_\alpha X}\,(\approx_\alpha\mathbf{X})}{f(x\ f)\cdot X \approx_\alpha fX}\,(\approx_\alpha\mathbf{f},\approx_\alpha\mathbf{tup})}{x(f(x\ f)\cdot X) \approx_\alpha x(fX)}\,(\approx_\alpha\mathbf{f},\approx_\alpha\mathbf{tup}) \qquad \dfrac{}{x\#[x]f(xX)}\,(\#\mathbf{absa})}{\dfrac{\lambda[f]x(f(x\ f)\cdot X) \approx_\alpha \lambda[f]x(fX)}{[f]\lambda[x]f(xX) \approx_\alpha [x]\lambda[f]x(fX)}\,(\approx_\alpha\mathbf{f},\approx_\alpha\mathbf{tup},\approx_\alpha\mathbf{absa}) \qquad \dfrac{}{x\#\lambda[x]f(xX)}\,(\#\mathbf{f})}\,(\approx_\alpha\mathbf{absb})}{\lambda[f]\lambda[x]f(xX) \approx_\alpha \lambda[x]\lambda[f]x(fX)}\,(\approx_\alpha\mathbf{f},\approx_\alpha\mathbf{tup})$$

Here we use the fact that $ds((a\ b),\mathtt{Id}) = \{a,b\}$. We recall that $\lambda[f]\lambda[x]f(xX)$ is actually $\mathtt{lam}([f]\mathtt{lam}([x]\mathtt{app}(f,\mathtt{app}(x,\mathtt{Id}\cdot X))))$, where $\mathtt{lam}$ is applied to a tuple with one element. We write several rules used together as (**rule1**,**rule2**,...).

Definition 8 *We call constraints of the form $a\#a$ and $a\#X$ **reduced**, and write Δ,∇,Γ for sets of reduced constraints, we may call them **contexts**. If there are no constraints of the form $a\#a$ in Δ we say it is **consistent**.*

Intuitively, an assumption $a\#a$ can never be true; we could add a rule in the system to reflect this fact:

$$\dfrac{a\#a}{P}\ \text{for any } P$$

See [19] for a presentation with this rule. On the other hand, $a\#X$ might be true if we instantiate X sensibly (to b, say, but not to a).

Definition 9 (Problems and entailment) *A set Pr of constraints will be called a **problem**. We write $\Delta \vdash Pr$ when proofs of P exist for all $P \in Pr$, using the derivation rules above and elements of the context Δ as assumptions. We say that Δ **entails** Pr. If $\Delta \vdash P$ because $P \in \Delta$ we say Δ **trivially entails** P, or that the derivation is **trivial**.*

Remark: In contrast with higher-order systems, here α-equivalence is axiomatised instead of being built into the syntactic equality. This might seem inefficient, because to decide α-equivalence in the front end (the syntax we use to reason about terms) we must do an explicit proof. However, this is an illusion, since we are free in nominal rewriting, as with any other framework, to choose the back end (the underlying representation) wisely so it is efficient for the manipulations we intend to carry out. When we define nominal matching later, we will build the α-equivalence into the matching algorithm; but by making the calculation of α-equivalence explicit in the front end, we lost nothing and gain useful proof principles. $*$

We give below the specification of an algorithm to check the validity of freshness and α-equality constraints.

3.1 An algorithm to check constraints

The rules above decompose syntax and the part above the line in a rule is always strictly simpler than the part below the line. This is not completely obvious for the second rule for abstractions $\dfrac{s \approx_\alpha (a\ b)\cdot t \quad a\#t}{[a]s \approx_\alpha [b]t}$ until we recall that $(a\ b)\cdot t$ is not itself a term but sugar for a term with a swapped with b and (if there are unknowns X in t), $(a\ b)$ suspended on X. The depth of this term is strictly less than that of $[b]t$. Based on this observation, we give below an algorithm to check constraints. We refer to [18] for a description of an efficient implementation.

The algorithm is specified as a set of simplification rules acting on problems. We will later extend these rules to solve matching problems between left-hand sides of rewrite rules and terms; in anticipation we use l for terms in the simplification rules for $\approx_\alpha$.

Definition 10 (Simplification rules for problems) *Here a, b denote any pair of distinct atoms, $\pi\cdot X$ denotes a moderated variable, and f a term-former.*

$$
\begin{aligned}
a\#b, Pr &\Longrightarrow Pr \\
a\#fs, Pr &\Longrightarrow a\#s, Pr \\
a\#(s_1, \ldots, s_n), Pr &\Longrightarrow a\#s_1, \ldots, a\#s_n, Pr \\
a\#[b]s, Pr &\Longrightarrow a\#s, Pr \\
a\#[a]s, Pr &\Longrightarrow Pr \\
a\#\pi\cdot X, Pr &\Longrightarrow \pi^{-1}\cdot a\#X, Pr \qquad\qquad \pi \not\equiv \mathtt{Id}
\end{aligned}
$$

$$a \approx_\alpha a, Pr \Longrightarrow Pr$$
$$(l_1, \ldots, l_n) \approx_\alpha (s_1, \ldots, s_n), Pr \Longrightarrow l_1 \approx_\alpha s_1, \ldots, l_n \approx_\alpha s_n, Pr$$
$$fl \approx_\alpha fs, Pr \Longrightarrow l \approx_\alpha s, Pr$$
$$[a]l \approx_\alpha [a]s, Pr \Longrightarrow l \approx_\alpha s, Pr$$
$$[b]l \approx_\alpha [a]s, Pr \Longrightarrow (a\ b){\cdot}l \approx_\alpha s, a\#l, Pr$$
$$\pi{\cdot}X \approx_\alpha \pi'{\cdot}X, Pr \Longrightarrow ds(\pi, \pi')\#X, Pr$$

These rules define a reduction relation on problems: We write $Pr \Longrightarrow Pr'$ when Pr' is obtained from Pr by applying a simplification rule, and we write $\stackrel{*}{\Longrightarrow}$ for the transitive and reflexive closure of $\Longrightarrow$.

These rules 'run the derivation rules in reverse', in no particular order. In view of the example derivations above, we leave the reader to check that the following hold:

$$a\#(X, [a]Y) \stackrel{*}{\Longrightarrow} a\#X \qquad a\#fa \stackrel{*}{\Longrightarrow} a\#a$$
$$a\#((a\ b){\cdot}X, (b\ c){\cdot}Y) \stackrel{*}{\Longrightarrow} b\#X, a\#Y,$$

and that the following hold:

$$(a\ b){\cdot}X \approx_\alpha X \stackrel{*}{\Longrightarrow} a\#X, b\#X \qquad [a]a \approx_\alpha [b]b \stackrel{*}{\Longrightarrow} \{\}$$
$$\lambda[f]\lambda[x]f(xX) \approx_\alpha \lambda[x]\lambda[f]x(fX) \stackrel{*}{\Longrightarrow} x\#X, f\#X.$$

Intuitively, if a constraint can be reduced to the emptyset (as in the second example for $\approx_\alpha$ above) then the predicate holds. If after simplifying a problem as much as possible, still some constraints remain, then we will need those as assumptions to derive the constraints in the problem. We will formalise these observations below.

Lemma 11 *The relation $\Longrightarrow$ is confluent (i.e. $Pr \stackrel{*}{\Longrightarrow} Pr_1$ and $Pr \stackrel{*}{\Longrightarrow} Pr_2$ implies that there exists Pr_3 such that $Pr_1 \stackrel{*}{\Longrightarrow} Pr_3$ and $Pr_2 \stackrel{*}{\Longrightarrow} Pr_3$) and strongly normalising (i.e. the simplification process terminates).*

Proof By Newman's Lemma [34] we need only show termination, because the simplification rules do not overlap (there are no critical pairs). The rules form a hierarchical system in the sense of [22], from which it follows that if the first group of rules is terminating (it is, since the rules decrease the size of the problem, defined as the multiset of sizes of individual constraints) and non-duplicating (it is, since no terms are duplicated) and does not use in the right-hand side any symbol defined in the second group (i.e. equality $\approx_\alpha$; it doesn't), then if the rules for the equality symbol satisfy the general

recursive scheme, then the whole system is terminating. The general recursive scheme requires that recursive calls in right-hand sides use strict subterms of the left-hand side arguments, and this is the case (permutations are ignored). ∗

As a consequence, the simplification rules define a function from problems to their unique normal forms. We write $\langle Pr \rangle_{nf}$ for the normal form of Pr, and $\langle P \rangle_{nf}$ for $\langle \{P\} \rangle_{nf}$, i.e. the result of simplifying it as much as possible.

The following technical properties are direct corollaries of confluence of $\Longrightarrow$:

Corollary 12 *$\langle Pr \cup Pr' \rangle_{nf} = \langle Pr \rangle_{nf} \cup \langle Pr' \rangle_{nf}$, and as a corollary if $Pr \subseteq Pr'$ then $\langle Pr \rangle_{nf} \subseteq \langle Pr' \rangle_{nf}$.*

Proof The algorithm for determining $\langle Pr \cup Pr' \rangle_{nf}$ works elementwise on the elements of $Pr \cup Pr'$, and is confluent by Lemma 11. ∗

We will say that an equality constraint $u \approx_\alpha v$ is **reduced** when one of the following holds:

- u and v are distinct atoms. For example $a \approx_\alpha b$ is a reduced equality.
- u and v are applications with different term-formers (e.g. $ft \approx_\alpha gs$).
- u and v are two different variables. So $\pi{\cdot}X \approx_\alpha \pi'{\cdot}Y$ is reduced, but $\pi{\cdot}X \approx_\alpha \pi'{\cdot}X$ is not, for any π and π'.
- u and v have different term constructors at the root. For example $[a]s \approx_\alpha (t, t')$, $X \approx_\alpha ft$, and $a \approx_\alpha \pi{\cdot}X$, are all reduced equalities.

Recall from §3 that we call a freshness constraint $a\#s$ **reduced** when it is of the form $a\#a$ or $a\#X$ (i.e. when $s \equiv a$ or $s \equiv X$). We call the first **inconsistent** and the second **consistent**.

Say a problem Pr is **reduced** when it consists of reduced constraints, and **inconsistent** when $\langle Pr \rangle_{nf}$ contains an inconsistent element — so Pr is inconsistent if and only if $\langle Pr \rangle_{nf}$ is, if and only if $a\#a \in \langle Pr \rangle_{nf}$ for some a.

Lemma 13 *Pr is reduced if and only if $Pr = \langle Pr \rangle_{nf}$.*

Proof We check the simplification rules above, and the definition of a reduced constraint, and see that a simplification rule applies to a constraint, if and only if that constraint is *not* reduced. ∗

Corollary 14 (Characterisation of normal forms) *(1) $\langle a\#s \rangle_{nf}$ is a context Δ, as defined in Section 3.*

Δ need not be consistent. For example $\langle a\#a \rangle_{nf} = \{a\#a\}$ is an inconsistent context.

(2) $\langle s \approx_\alpha t \rangle_{nf}$ is of the form $\Delta \cup Contr \cup Eq$ where Δ is a set of consistent

reduced freshness constraints (that is, a consistent context), Eq is a set of reduced equality constraints, and Contr is a set of inconsistent reduced freshness constraints.

Any of Δ, Contr, and Eq, may be empty.

(3) $\langle Pr \rangle_{nf}$ *is of the form* $\Delta \cup Contr \cup Eq$ *which is as above.*

Proof Direct consequence of the previous lemma.

*

So now we know Pr simplifies to a unique normal form $\langle Pr \rangle_{nf}$, and we have a good idea of the structure of $\langle Pr \rangle_{nf}$.

The rest of this subsection addresses the question: How do logical entailment $\Gamma \vdash Pr$ and $\langle Pr \rangle_{nf}$ interact?

Lemma 15 *(1) Assume* $Pr \stackrel{*}{\Longrightarrow} Pr'$. *Then* $\Gamma \vdash Pr$ *if and only if* $\Gamma \vdash Pr'$.
(2) $\Gamma \vdash Pr$ *if and only if* $\Gamma \vdash \langle Pr \rangle_{nf}$.

Proof

(1) There are 12 simplification rules which could have been applied in a step $Pr \Longrightarrow Pr'$ so there are precisely 12 cases to consider. Each corresponds precisely to one of the 12 syntax-directed derivation rules defining the entailment relation. The result follows by induction on the number of steps in the simplification $Pr \stackrel{*}{\Longrightarrow} Pr'$.
(2) An immediate consequence of the first part.

*

Lemma 16 *If* Γ *is consistent and* $\Gamma \vdash Pr$ *then* Pr *is consistent and moreover if it is in normal form then it does not contain equality constraints.*

Proof By a simple induction on derivations. *

Theorem 17 *Write* $\langle Pr \rangle_{nf} = \Delta \cup Contr \cup Eq$ *as described by Corollary 14.* $\Delta \vdash Pr$ *if and only if Contr and Eq are empty.*

Proof

By Lemma 15, $\Gamma \vdash Pr$ if and only if $\Gamma \vdash \Delta, Contr, Eq$. The result now follows easily by Lemma 16 because Δ is consistent (see Corollary 14). *

Corollary 18 *Let* Γ *and* Δ *be consistent contexts, and* Pr *and* Pr' *be any problems.*

(1) Correctness of the Algorithm: $\Gamma \vdash Pr$ *if and only if* $\langle Pr \rangle_{nf} = \Delta$ *(that*

is, Contr and Eq are empty) and $\Gamma \vdash \Delta$.

(2) *Cut: If* $\Gamma \vdash \Delta$ *and* $\Gamma, \Delta \vdash \Psi$*, then* $\Gamma \vdash \Psi$*. (This is, of course, a form of Cut rule.)*

In particular, if Γ *and* Δ *are both consistent and* $\Gamma \vdash \Delta$ *and* $\Delta \vdash \Psi$*, then* $\Gamma \vdash \Psi$.

(3) *If* $\Gamma \vdash Pr$ *and* $\Gamma, \langle Pr \rangle_{nf} \vdash Pr'$*, then* $\Gamma \vdash Pr'$.

Proof

(1) Suppose $\Gamma \vdash Pr$. By Theorem 17, $\langle Pr \rangle_{nf} = \Delta$ and by Lemma 15, $\Gamma \vdash \Delta$.
Conversely if $\langle Pr \rangle_{nf} = \Delta$ and $\Gamma \vdash \Delta$, by the same results $\Gamma \vdash Pr$.

(2) Suppose $\Gamma \vdash \Delta$. By the first part of this result, $\Gamma \vdash C$ for each $C \in \Delta$. Now we can paste into the derivation of $\Delta \vdash \Psi$ to obtain a derivation of $\Gamma \vdash \Psi$ as required.

(3) By the previous two parts.

*

3.2 Properties of # and $\approx_\alpha$

The following lemma indicates that use of permutations is **extensional**, in the sense that lists of swappings denoting the same permutation, are logically indistinguishable.

Lemma 19 *Suppose* ∇ *is a context. If* $ds(\pi, \pi') = \emptyset$ *then:*

(1) $\nabla \vdash \pi \cdot a \# t \iff \nabla \vdash \pi' \cdot a \# t$.
(2) $\nabla \vdash a \# \pi \cdot t \iff \nabla \vdash a \# \pi' \cdot t$.
(3) $\nabla \vdash \pi \cdot s \approx_\alpha t \iff \nabla \vdash \pi' \cdot s \approx_\alpha t$.
(4) $\nabla \vdash s \approx_\alpha \pi \cdot t \iff \nabla \vdash s \approx_\alpha \pi' \cdot t$.

Proof We consider the four logical equivalences in turn:

(1) We observe simply that $ds(\pi, \pi') = \emptyset$ precisely when $\pi \cdot a = \pi' \cdot a$ always, so $\pi \cdot a \equiv \pi' \cdot a$ and there is nothing more to prove.

(2) We work by induction on the derivation of $a \# \pi \cdot t$ from ∇. Note that because of the syntax-directed nature of the derivation rules, this is equivalent to working by induction on the syntax of t. We consider just a few cases.
- Suppose the derivation concludes in (#**X**), so $\nabla \vdash a \# \pi \cdot X$ and $\pi^{-1} \cdot a \# X \in \nabla$. By the same observation as before (i.e., $ds(\pi, \pi') = \emptyset$), $\pi^{-1} \cdot a \equiv \pi'^{-1} \cdot a$ and we are done.
- Suppose the derivation concludes in (#**absa**) so $\nabla \vdash a \# \pi \cdot ([b]s)$ and $\pi \cdot b \equiv a$. Since $\pi' \cdot b \equiv a$ we use (#**absa**) to build a derivation of $\nabla \vdash a \# \pi' \cdot [b] \cdot s$.

- Suppose the derivation concludes in ($\#\mathbf{absb}$) so $\nabla \vdash a\#\pi\cdot([b]s)$, $\pi\cdot b \equiv b' \not\equiv a$, and $\nabla \vdash a\#\pi\cdot s$ is derivable. By inductive hypothesis $\nabla \vdash a\#\pi'\cdot s$ is derivable, and we continue as in the previous case.

(3) We work by induction on the derivation of $\nabla \vdash \pi\cdot s \approx_\alpha t$. Again, we consider only a few cases.
- Suppose the derivation concludes in ($\approx_\alpha\mathbf{X}$), so we conclude $\nabla \vdash \pi \circ \tau\cdot X \approx_\alpha \tau'\cdot X$ from $\nabla \vdash a\#X$ for every $a \in ds(\pi \circ \tau, \tau')$. We now observe that $ds(\pi \circ \tau, \tau') = ds(\pi' \circ \tau, \tau')$ so we can write a derivation concluding in $\nabla \vdash \pi' \circ \tau\cdot X \approx_\alpha \tau'\cdot X$, from the same hypotheses.
- Suppose the derivation concludes in ($\approx_\alpha\mathbf{a}$). It suffices to observe that $\pi\cdot b = \pi'\cdot b$ always.
- Suppose the derivation concludes in ($\approx_\alpha\mathbf{absa}$), so we conclude $\nabla \vdash [\pi\cdot a]\pi\cdot s \approx_\alpha [\pi\cdot a]t$ and $\nabla \vdash \pi\cdot s \approx_\alpha t$ is also derivable. By inductive hypothesis $\nabla \vdash \pi'\cdot s \approx_\alpha t$ and since $\pi'\cdot a = \pi\cdot a$ we can extend this derivation with ($\approx_\alpha\mathbf{absa}$) to derive $\nabla \vdash [\pi'\cdot a]\pi'\cdot s \approx_\alpha [\pi\cdot a]t$.
- Suppose the derivation concludes in ($\approx_\alpha\mathbf{absb}$), so we conclude $\nabla \vdash [\pi\cdot a]\pi\cdot s \approx_\alpha [b]t$ where $b \not\equiv \pi\cdot a$, and $\nabla \vdash (b\ \pi\cdot a) \circ \pi\cdot s \approx_\alpha t$ and $\nabla \vdash b\#\pi\cdot s$ are also derivable.

  We now observe that $ds((b\ \pi\cdot a) \circ \pi, (b\ \pi'\cdot a) \circ \pi') = \emptyset$ (trivially, since $ds(\pi, \pi') = \emptyset$ and $\pi\cdot a = \pi'\cdot a$), and the result follows by the inductive hypothesis and by part 2 of this result.

(4) Much as for the previous case.

*

For example, the lemma above allows us to replace $\pi^{-1} \circ \pi$ with $\mathtt{Id}$ since $ds(\pi^{-1} \circ \pi, \mathtt{Id}) = \emptyset$.

Given a derivable judgement $\nabla \vdash P$, we can:

(1) instantiate unknowns ($\nabla \vdash P$ maps to $\nabla\sigma \vdash P\sigma$ in a suitable formal sense), and
(2) permute atoms ($\nabla \vdash P$ maps to $\nabla \vdash \pi\cdot P$).

We show that derivability is preserved by both. Note that for the case of permutation we only permute atoms in the conclusion P, not in the assumptions ∇.[3]

In the rest of this section we assume that we are working with a consistent context unless stated otherwise, in other words, we consider derivations $\nabla \vdash$

[3] The *intuition* is that in Definition 6 ($\#\mathbf{X}$) makes it clear that $a\#X$ if and only if $\pi'\cdot a\#\pi'\cdot X$, and Definition 3 makes it clear that permutations commute through all term-formers. Also note that renamings (possibly non-injective functions from atoms to themselves, e.g. 'replace a by b') do not satisfy these useful properties; it is essential to use swappings.

Pr where ∇ is a consistent context.

Lemma 20 *(1) If $\nabla \vdash a\#t$ then $\nabla \vdash \pi\cdot a\#\pi\cdot t$. Similarly if $\nabla \vdash s \approx_\alpha t$ then $\nabla \vdash \pi\cdot s \approx_\alpha \pi\cdot t$.*

This can be restated as follows: $\nabla \vdash Pr$ if and only if $\nabla \vdash \pi\cdot Pr$, where here the action of π is pointwise on all terms mentioned in Pr.

(2) $\nabla \vdash a\#\pi\cdot t$ if and only if $\nabla \vdash \pi^{-1}\cdot a\#t$, and similarly $\nabla \vdash \pi\cdot s \approx_\alpha t$ if and only if $\nabla \vdash s \approx_\alpha \pi^{-1}\cdot t$. (This turns out to be particularly useful.)

Proof The first part is by routine induction on derivations. We consider a few cases:

- Suppose the derivation concludes in (#**X**), so $\nabla \vdash a\#\tau\cdot X$ is derivable. It follows that $\tau^{-1}\cdot a\#X \in \nabla$. It is now easy to construct a derivation of $\nabla \vdash \pi\cdot a\#\pi\circ\tau\cdot X$, using (#**X**).
- Suppose the derivation concludes in (#**absb**), so $\nabla \vdash a\#[b]s$ is derivable. By the inductive hypothesis $\nabla \vdash \pi\cdot a\#\pi\cdot s$ is derivable, and we can also extend its derivation with (#**absb**).
- Suppose the derivation concludes in ($\approx_\alpha$**absa**), so $\nabla \vdash [a]s \approx_\alpha [a]t$ is derivable. By inductive hypothesis $\nabla \vdash \pi\cdot s \approx_\alpha \pi\cdot t$, and so

$$\nabla \vdash \pi\cdot[a]s \equiv [\pi\cdot a]\pi\cdot s \approx_\alpha [\pi\cdot a]\pi\cdot t \equiv \pi\cdot[a]t.$$

- Suppose the derivation concludes in ($\approx_\alpha$**absb**), so $\nabla \vdash [a]s \approx_\alpha [b]t$ is derivable. [4]

  By assumption $\nabla \vdash (b\ a)\cdot s \approx_\alpha t$ and $\nabla \vdash b\#s$ are derivable. By inductive hypothesis

$$\nabla \vdash \pi\circ(b\ a)\cdot s \approx_\alpha \pi\cdot t \quad \text{and} \quad \nabla \vdash \pi\cdot b\#\pi\cdot s$$

  are derivable.

  Now it is a fact that $ds(\pi\circ(b\ a), (\pi\cdot b\ \pi\cdot a)\circ\pi) = \emptyset$. Therefore, by Lemma 19

$$\nabla \vdash (\pi\cdot b\ \pi\cdot a)\circ\pi\cdot s \approx_\alpha \pi\cdot t \quad \text{and} \quad \nabla \vdash \pi\cdot b\#\pi\cdot s,$$

  are derivable, and so using ($\approx_\alpha$**absb**) we obtain

$$\nabla \vdash \pi\cdot[a]s \equiv [\pi\cdot a]\pi\cdot s \approx_\alpha [\pi\cdot b]\pi\cdot t \equiv \pi\cdot[b]t$$

  as required.

For the second part, we simply observe that $ds(\mathtt{Id}, \pi^{-1}\circ\pi) = \emptyset$ and use Lemma 19 and the first part. $*$

[4] In [41] the result which 'did' this case (Theorem 2.11) was proved by simultaneous induction with transitivity of $\approx_\alpha$. We do *not* need this, because we use Lemma 19. Lemmas 2.7 and 2.8 in [41] have more-or-less the same content, but the mention of equality in 2.8 makes it a little harder to use and forces the simultaneous induction.

We use this *technical lemma* in Lemma 22 below (a converse to this lemma is also true, see Lemma 34).

Lemma 21 *If* $\nabla \vdash a\#s$ *for each* $a \in ds(\pi, \pi')$*, then* $\nabla \vdash \pi{\cdot}s \approx_\alpha \pi'{\cdot}s$.

Proof We work by induction on the syntax of s for all π and π'.

(1) Suppose $s \equiv c$ for some atom c. Now either $c \in ds(\pi, \pi')$ or not; if $c \in ds(\pi, \pi')$ then $\nabla \vdash c\#c$ contradicting our assumption that ∇ is consistent. Otherwise, $\pi{\cdot}c \equiv \pi'{\cdot}c$ and there is nothing to prove.

(2) If $s \equiv \pi_1{\cdot}X$ we observe by group theory that $ds(\pi \circ \pi_1, \pi' \circ \pi_1) = ds(\pi, \pi')$, and we can use (#**X**).

(3) Suppose $s \equiv [a]s'$. Observe that either $a \in ds(\pi, \pi')$ or not.

(a) In the first case we construct a derivation as follows:

$$\dfrac{\dfrac{ds((\pi'{\cdot}a\ \pi{\cdot}a) \circ \pi, \pi')\#s}{\nabla \vdash (\pi'{\cdot}a\ \pi{\cdot}a) \circ \pi{\cdot}s \approx_\alpha \pi'{\cdot}s} \qquad \nabla \vdash \pi'{\cdot}a\#\pi{\cdot}s}{\nabla \vdash [\pi{\cdot}a]\pi{\cdot}s \approx_\alpha [\pi'{\cdot}a]\pi'{\cdot}s}\ (\approx_\alpha \mathbf{absb})$$

To explain the right-hand branch of the proof, we must do some basic group theory. Since $\pi{\cdot}a \not\equiv \pi'{\cdot}a$, also $a \not\equiv (\pi^{-1} \circ \pi'){\cdot}a$, so $\pi'{\cdot}a \not\equiv (\pi' \circ \pi^{-1} \circ \pi'){\cdot}a$, and finally

$$(\pi \circ \pi^{-1} \circ \pi') \cdot a \quad \not\equiv \quad (\pi' \circ \pi^{-1} \circ \pi') \cdot a.$$

Thus, $\pi^{-1} \circ \pi'{\cdot}a \in ds(\pi, \pi')$ and $\nabla \vdash \pi^{-1} \circ \pi'{\cdot}a\#s$ is derivable. Using the previous lemma $\nabla \vdash \pi'{\cdot}a\#\pi{\cdot}s$ is derivable as required.

Concerning the left-hand branch, we observe that the top line is not a real derivation rule but represents a use of the induction hypothesis, having verified (by some more basic group theory) that

$$ds((\pi'{\cdot}a\ \pi{\cdot}a) \circ \pi, \pi') \ = \ ds(\pi, \pi') \setminus \{a\}.$$

(b) In the second case ($a \notin ds(\pi, \pi')$) we write $b \equiv \pi{\cdot}a \equiv \pi'{\cdot}a$ and construct a derivation as follows:

$$\dfrac{\dfrac{ds(\pi, \pi')\#s}{\nabla \vdash \pi{\cdot}s \approx_\alpha \pi'{\cdot}s}\ (\text{Lemma 21})}{\nabla \vdash [b]\pi{\cdot}s \approx_\alpha [b]\pi'{\cdot}s}\ (\approx_\alpha \mathbf{absa})$$

(Here the topmost horizontal line actually represents a derivation which by Lemma 21 exists.)

(4) Other cases are similar and simpler.

*

Lemma 22 *Suppose ∇ and $\nabla\sigma$ are consistent.*

If $\nabla \vdash a\#t$ then $\langle\nabla\sigma\rangle_{nf} \vdash a\#(t\sigma)$. Similarly if $\nabla \vdash s \approx_\alpha t$ then $\langle\nabla\sigma\rangle_{nf} \vdash (s\sigma) \approx_\alpha (t\sigma)$. More generally, if $\nabla \vdash Pr$ then $\langle\nabla\sigma\rangle_{nf} \vdash Pr\sigma$.

Proof We work by induction on the derivation of $\nabla \vdash a\#t$ or $\nabla \vdash s \approx_\alpha t$. We consider a few cases:

- Suppose the derivation concludes with $(\#\mathbf{X})$ so $\nabla \vdash a\#\pi\cdot X$. It follows that $\pi^{-1}\cdot a\#X \in \nabla$. By Lemma 15 $\langle(\pi^{-1}\cdot a\#X)\sigma\rangle_{nf} \vdash (\pi^{-1}\cdot a\#X)\sigma$. Also, by Corollary 12 $\langle\pi^{-1}\cdot a\#X\sigma\rangle_{nf} \subseteq \langle\nabla\sigma\rangle_{nf}$.
  Therefore $\langle\nabla\sigma\rangle_{nf} \vdash \pi^{-1}\cdot a\#X\sigma$. By Lemma 20 $\langle\nabla\sigma\rangle_{nf} \vdash a\#\pi\cdot(X\sigma)$ and by Lemma 5 $\pi\cdot(X\sigma) \equiv (\pi\cdot X)\sigma$, and the result follows.
- Suppose the derivation concludes with $(\#\mathbf{absb})$ so $\nabla \vdash a\#[b]t$ and $\nabla \vdash a\#t$ are derivable. By inductive hypothesis $\langle\nabla\sigma\rangle_{nf} \vdash a\#t\sigma$ is derivable. We also observe that $[b](t\sigma) \equiv ([b]t)\sigma$ and the result follows.
- Suppose the derivation concludes with $(\approx_\alpha\mathbf{X})$ so $\nabla \vdash a\#X$ for each $a \in ds(\pi,\pi')$ and $\nabla \vdash \pi\cdot X \approx_\alpha \pi'\cdot X$ is derivable.
  By inductive hypothesis $\langle\nabla\sigma\rangle_{nf} \vdash a\#X\sigma$ for each $a \in ds(\pi,\pi')$. We now use the preceding *technical lemma*.
- Suppose the derivation concludes with $(\approx_\alpha\mathbf{absb})$ so $\nabla \vdash [a]s \approx_\alpha [b]t$, $\nabla \vdash (b\ a)\cdot s \approx_\alpha t$, and $\nabla \vdash b\#s$ are derivable. We can use the inductive hypothesis directly, once we recall Lemma 5 and observe that $((b\ a)\cdot s)\sigma \equiv (b\ a)\cdot(s\sigma)$.

*

Lemma 23 *(1) If $\nabla \vdash n\#s$ and $\nabla \vdash s \approx_\alpha t$ then $\nabla \vdash n\#t$.*
(2) If $\nabla \vdash s \approx_\alpha t$ and $\nabla \vdash t \approx_\alpha u$ then $\nabla \vdash s \approx_\alpha u$.

Proof For the first part (#) we work by induction on the syntax of s:

- If $s \equiv \pi\cdot X$ then by the syntax-directed nature of the rules for deriving $\nabla \vdash s \approx_\alpha t$, we see that the derivation must conclude in $(\approx_\alpha\mathbf{X})$ so $t \equiv \pi'\cdot X$ and $\nabla \vdash a\#X$ for every $a \in ds(\pi,\pi')$.
  Now suppose $\nabla \vdash n\#\pi\cdot X$. By Lemma 20, $\nabla \vdash \pi^{-1}\cdot n\#X$.
  If $n \notin ds(\pi,\pi')$ then $\nabla \vdash \pi'^{-1}\cdot n\#X$ and by Lemma 20, $\nabla \vdash n\#\pi'\cdot X$.
  If $n \in ds(\pi,\pi')$ then also $\pi'^{-1}\cdot n \in ds(\pi,\pi')$ so by the argument above, $\nabla \vdash \pi'^{-1}\cdot n\#X$ and by Lemma 20, $\nabla \vdash n\#\pi'\cdot X$.
- If $s \equiv [a]s'$ then there are two possibilities:

(1) $t \equiv [a]t'$ and the derivation concludes in $(\approx_\alpha\mathbf{absa})$. Then either $n \equiv a$ and we are done, or we can use the inductive hypothesis.
(2) $t \equiv [b]t'$ and the derivation concludes in $(\approx_\alpha\mathbf{absb})$. Then $\nabla \vdash (b\ a)\cdot s \approx_\alpha t$. We now work by cases according to whether $n = a$, $n = b$, or $n \neq a, b$, using Lemma 20.
(3) If $s \equiv (s_1,\ldots,s_n)$ then again by the syntax-directed nature of the rules, the derivation must conclude in $(\approx_\alpha\mathbf{tup})$ so $t \equiv (t_1,\ldots,t_n)$ and we use the

inductive hypothesis.

(4) Other cases are similar.

For the second part ($\approx_\alpha$), we sketch the proof semi-formally, but it can very easily be made completely formal in the style of the first part. We work by induction on the *size* of s (permutations are not counted in the size):

- If $\nabla \vdash \pi{\cdot}X \approx_\alpha \pi'{\cdot}X \approx_\alpha \pi''{\cdot}X$, the result follows by easy calculations to verify that $ds(\pi,\pi'') \subseteq ds(\pi,\pi') \cup ds(\pi',\pi'')$.
- If $\nabla \vdash [a]s' \approx_\alpha [a]t' \approx_\alpha [a]u'$ then it must be that $\nabla \vdash s' \approx_\alpha t' \approx_\alpha u'$ and we use the inductive hypothesis.
- If $\nabla \vdash [a]s' \approx_\alpha [b]t' \approx_\alpha [b]u'$ then $\nabla \vdash (b\ a){\cdot}s' \approx_\alpha t',\ b\#s',\ t' \approx_\alpha u'$. By the inductive hypothesis $\nabla \vdash (b\ a){\cdot}s' \approx_\alpha u'$ and the result follows.
- If $\nabla \vdash [a]s' \approx_\alpha [a]t' \approx_\alpha [b]u'$ then $\nabla \vdash s' \approx_\alpha t',\ (b\ a){\cdot}t' \approx_\alpha u',\ b\#t'$. By the inductive hypothesis and using Lemma 20, $\nabla \vdash (b\ a){\cdot}s' \approx_\alpha u'$ and by the previous part, $\nabla \vdash b\#s'$. The result follows.
- If $\nabla \vdash [a]s' \approx_\alpha [b]t' \approx_\alpha [c]u'$ then

  $$\nabla \vdash (b\ a){\cdot}s' \approx_\alpha t',\ b\#s',\ \ (c\ b){\cdot}t' \approx_\alpha u',\ c\#t'.$$

  It follows by induction that $\nabla \vdash (c\ b) \circ (b\ a){\cdot}s' \approx_\alpha u'$. Now $ds((c\ b) \circ (b\ a),(c\ a)) = \{b\}$ and $\nabla \vdash b\#s'$, so by Lemma 21 (the *technical lemma* above),

  $$\nabla \vdash (c\ a){\cdot}s' \approx_\alpha (c\ b) \circ (b\ a){\cdot}s' \approx_\alpha u'$$

  Now by inductive hypothesis we may complete the proof.
- Other cases are simple.

*

Fix a consistent context ∇ and say $\approx_\alpha$ is an **equivalence relation** when it is reflexive, transitive and symmetric (as usual). Say $\approx_\alpha$ is a **congruence** when it is an equivalence relation such that if $s \approx_\alpha t$ then $fs \approx_\alpha ft$, $[a]s \approx_\alpha [a]t$, $(\cdots s \cdots) \approx_\alpha (\cdots t \cdots)$, and $\pi{\cdot}s \approx_\alpha \pi{\cdot}t$.

Theorem 24 *$\approx_\alpha$ is an equivalence relation and a congruence in a consistent context.*

Proof Transitivity is by the previous lemma. Reflexivity is by an easy induction on syntax. Symmetry is slightly non-trivial.

We work by induction on the maximum of the sizes of s and t proving that if $\nabla \vdash s \approx_\alpha t$ then $\nabla \vdash t \approx_\alpha s$. Mostly this is easy, but ($\approx_\alpha$**absb**) causes difficulty because of its asymmetry. Suppose $\nabla \vdash [a]s \approx_\alpha [b]t$ is derived by ($\approx_\alpha$**absb**), so also $\nabla \vdash (b\ a){\cdot}s \approx_\alpha t$ and $\nabla \vdash b\#s$. By Lemma 19 also $\nabla \vdash (a\ b){\cdot}s \approx_\alpha t$, and by Lemma 20 $\nabla \vdash s \approx_\alpha (a\ b){\cdot}t$. We now use the inductive hypothesis to deduce $\nabla \vdash (a\ b){\cdot}t \approx_\alpha s$, and also Lemma 23 and Lemma 20 to deduce

$\nabla \vdash a\#t$. The result now follows.

Congruence is by induction: suppose $\nabla \vdash s \approx_\alpha t$. Then:

- $\nabla \vdash (u_1, \ldots, u_{k-1}, s, u_{k+1}, \ldots, u_n) \approx_\alpha (u_1, \ldots, u_{k-1}, t, u_{k+1}, \ldots, u_n)$ using $(\approx_\alpha \mathbf{tup})$.
- $\nabla \vdash [a]s \approx_\alpha [a]t$ using $(\approx_\alpha \mathbf{absa})$.
- $\nabla \vdash \pi{\cdot}s \approx_\alpha \pi{\cdot}t$ by Lemma 21.

*

If P is a freshness constraint $a\#u$ or equality constraint $u \approx_\alpha v$, write $P[X{\mapsto}s]$ for $a\#u[X{\mapsto}s]$ or $u[X{\mapsto}s] \approx_\alpha v[X{\mapsto}s]$ respectively.

Corollary 25 *Suppose Γ is consistent and $\Gamma \vdash s \approx_\alpha t$. Then:*

(1) $\Gamma \vdash P[X{\mapsto}s]$ is derivable if and only if $\Gamma \vdash P[X{\mapsto}t]$ is derivable.
(2) $\Gamma, \langle P[X{\mapsto}s]\rangle_{nf} \vdash Q$ is derivable if and only if $\Gamma, \langle P[X{\mapsto}t]\rangle_{nf} \vdash Q$ is derivable.

Proof

(1) The case of $P \equiv u \approx_\alpha v$ follows by the previous theorem. The case of $P \equiv a\#u$ follows again by the previous theorem, and by part 1 of Lemma 23.
(2) Observe that $\Gamma, \langle P[X{\mapsto}s]\rangle_{nf} \vdash P[X{\mapsto}s]$ by Lemma 15. Using the first part of this result, $\Gamma, \langle P[X{\mapsto}s]\rangle_{nf} \vdash P[X{\mapsto}t]$.

Now suppose $\Gamma, \langle P[X{\mapsto}t]\rangle_{nf} \vdash Q$. Then using Corollary 18 and the observation in the previous paragraph, we conclude that $\Gamma, \langle P[X{\mapsto}s]\rangle_{nf} \vdash Q$.

*

So equality *is* 'an equality' with respect to the simple logic given by $\#$, $\approx_\alpha$, and $\vdash$.

4 Unification

4.1 Definitions

As usual unification is about finding some substitution making two terms s and t equal; however, now the notion of equality is our 'logical' notion of $\approx_\alpha$.

Definition 26 (Unification problems) *A **unification problem** Pr is a problem as previously defined but replacing equality constraints $s \approx_\alpha t$ by **unification constraints** $s \;?\!\approx\!?\; t$.*

We recall from Corollary 14 that $\langle s \approx_\alpha t \rangle_{nf}$ is of the form $\Delta \cup Contr \cup Eq$ where Δ is a consistent freshness context, $Contr$ is a set of inconsistent freshness constraints, and Eq is a set of reduced equalities. For example, $\langle [a]X \approx_\alpha [b]Y \rangle_{nf} = \{b\#X,\ (b\ a)\cdot X \approx_\alpha Y\}$. Intuitively, a solution to $[a]X \;?\!\approx\!?\; [b]Y$ is any substitution σ such that $(b\ a)\cdot X\sigma \approx_\alpha Y\sigma$, and such that $b\#X\sigma$.

Definition 27 (Solution) *A **solution** to a unification problem Pr is a* pair *(Γ, σ) of a consistent context and a substitution such that:*

(1) $\Gamma \vdash Pr'\sigma$ where Pr' is obtained from Pr by changing unification predicates into equality predicates and $Pr'\sigma$ is the problem obtained by applying the substitution σ to the terms in Pr'.
*(2) $X\sigma \equiv X\sigma\sigma$ for all X (we say the substitution is **idempotent**).*

*If there is no such (Γ, σ) we say that Pr is **unsolvable**.*

*Write $\mathcal{U}(Pr)$ for the **set of unification solutions** to Pr.*

The condition of idempotence is not absolutely necessary, but it is technically convenient and since our algorithms (see the next subsection) generate only idempotent solutions, we lose nothing.

Solutions in $\mathcal{U}(Pr)$ can be compared using the following relation (we will show it is actually an ordering).

Definition 28 *Let Γ_1, Γ_2 be consistent contexts, and σ_1, σ_2 substitutions. Then $(\Gamma_1, \sigma_1) \leq (\Gamma_2, \sigma_2)$ when there exists some σ' such that*

$$\textit{for all } X, \quad \Gamma_2 \vdash X\sigma_1\sigma' \approx_\alpha X\sigma_2 \quad \textit{and} \quad \Gamma_2 \vdash \Gamma_1\sigma'.$$

If we want to be more specific, we may write $(\Gamma_1, \sigma_1) \leq_{\sigma'} (\Gamma_2, \sigma_2)$.

Lemma 29 *$\leq$ defines a partial order on $\mathcal{U}(Pr)$, call it the **instantiation ordering**.*

Proof Reflexivity is trivial. For transitivity, suppose $(\Gamma_1, \sigma_1) \leq_{\sigma_1'} (\Gamma_2, \sigma_2) \leq_{\sigma_2'} (\Gamma_3, \sigma_3)$. Then we know (writing slightly informally):

$$\Gamma_2 \vdash\ \Gamma_1\sigma_1',\ X\sigma_1\sigma_1' \approx_\alpha X\sigma_2 \quad \text{and} \quad \Gamma_3 \vdash\ \Gamma_2\sigma_2',\ X\sigma_2\sigma_2' \approx_\alpha X\sigma_3.$$

Since Γ_3 is consistent, $\Gamma_2\sigma_2'$ is consistent by Lemma 16. Since Γ_2 is consistent, $\Gamma_1\sigma_1'$ is consistent by the same result. We also know

$$\langle\Gamma_2\sigma_2'\rangle_{nf} \vdash \Gamma_1\sigma_1'\sigma_2', \quad X\sigma_1\sigma_1'\sigma_2' \approx_\alpha X\sigma_2\sigma_2'$$

by Lemma 22. Finally, we use Corollary 18 and Theorem 24 to deduce

$$\Gamma_3 \vdash \Gamma_1\sigma_1'\sigma_2', \quad X\sigma_1\sigma_1'\sigma_2' \approx_\alpha X\sigma_3$$

as required. *

A **least element** of a partially ordered set is one which is related to (we generally say **less than or equal to**) every other element of the set.

Definition 30 *A **principal** (or **most general**) solution to a problem Pr is a least element of $\mathcal{U}(Pr)$.*

4.2 Principal solutions

We will now show that every solvable unification problem has a principal idempotent solution. The algorithm is derived from the simplification rules from the previous section, enriched with **instantiating** rules, labelled with substitutions. The conditions in the instantiating rules are usually called **occurs check**.

$$\pi{\cdot}X \;{}_?{\approx}_?\; u, Pr \stackrel{X\mapsto\pi^{-1}\cdot u}{\Longrightarrow} Pr[X\mapsto\pi^{-1}\cdot u] \qquad (X \notin V(u))$$

$$u \;{}_?{\approx}_?\; \pi{\cdot}X, Pr \stackrel{X\mapsto\pi^{-1}u}{\Longrightarrow} Pr[X\mapsto\pi^{-1}\cdot u] \qquad (X \notin V(u))$$

Note that the instantiating rules above apply also in the case $Pr = \emptyset$. Also note that we do not solve freshness constraints by instantiation — for a problem like $X \approx_\alpha blah$ it is obvious we should instantiate X to $blah$, but there is no obvious most general instantiation making, say, $a\#X$ true (any more than there is an obvious instantiation making, say 'x is a natural number' true); any term such that a is fresh for X would do. This is why we always work in a freshness context.

The simplification and instantiation rules specify a **unification algorithm**: to solve a problem Pr, we will apply the rules until we obtain an irreducible problem. This algorithm is in essence the same as [41] although our presentation is slightly different.

Many different possible reduction paths exist for a given Pr, since it may have many formulae each of which is susceptible to some simplification. Neither are reductions necessarily confluent; for example,

$$\{a\#X,\ X\ {}_?{\approx}_?\ Y\} \stackrel{[X\mapsto Y]}{\Longrightarrow} \{a\#Y\} \quad \text{and} \quad \{a\#X,\ X\ {}_?{\approx}_?\ Y\} \stackrel{[Y\mapsto X]}{\Longrightarrow} \{a\#X\}.$$

However reductions do always terminate with some normal form, since at each step either a formula becomes smaller, or the number of variables in the problem is reduced by one. Also we shall see later that these normal forms *are* all equivalent in a natural and useful sense (see Lemma 33).

It will be useful to syntactically characterise normal forms; for this, we define reduced unification constraints:

Definition 31 *A unification problem $u \ {}_?{\approx}_?\ v$ is **reduced** when one of the following holds:*

- *u and v are distinct atoms. For example $a \ {}_?{\approx}_?\ b$ is reduced.*
- *Precisely one of u and v is a moderated variable and the other term mentions that variable (so the occurrence check in the instantiating rules fails). For example $\pi{\cdot}X \ {}_?{\approx}_?\ (X, Y)$ or $(X, Y) \ {}_?{\approx}_?\ \pi{\cdot}X$, but not $\pi{\cdot}X \ {}_?{\approx}_?\ \pi'{\cdot}X$ or $X \ {}_?{\approx}_?\ Y$.*
- *u and v are applications with different term-formers (e.g. $ft \ {}_?{\approx}_?\ gs$).*
- *u and v have different term constructors at the root and neither is a moderated variable. For example $[a]s \ {}_?{\approx}_?\ (t, t')$.*

*We may call all reduced unification constraints **inconsistent**.*

If Pr reduces to a normal form Pr' via some sequence of substitutions σ, write $\langle Pr\rangle_{sol}$ for the tuple (Pr', σ).

Lemma 32 (Unification normal forms) • *$\langle a\#s\rangle_{sol}$ is $(\langle a\#s\rangle_{nf}, \mathtt{Id})$.*
- *$\langle s \ {}_?{\approx}_?\ t\rangle_{sol}$ is a problem of the form $\Delta \cup Contr \cup Eq$ and a substitution, where Δ is a consistent freshness context, $Contr$ is an inconsistent freshness context and Eq is a set of inconsistent unification constraints.*
- *As a corollary, $\langle Pr\rangle_{sol}$ is a problem of the form $\Delta \cup Contr \cup Eq$ as above and a substitution.*

Proof Just as in Corollary 14. We check that the simplification *and instantiating* rules are applicable to any non-reduced unification or freshness constraint. *

If we write $\langle Pr\rangle_{sol} = (\Delta, \sigma)$ we presume that $Pr \stackrel{*}{\Longrightarrow} \Delta$ with substitution σ and $Contr$ and Eq are empty. In this case, we may call (Δ, σ) the **solution** of Pr (soon we shall show it actually *is* a solution to Pr, in the sense that $(\Delta, \sigma) \in \mathcal{U}(Pr)$).

For example $\langle a\#a\rangle_{sol} = (\{a\#a\}, \mathtt{Id})$ and $\langle (X, [b]X, fX) \ {}_?{\approx}_?\ (a, [a]X, gX)\rangle_{sol} = (\{a\#a\}, \{b \ {}_?{\approx}_?\ a, fa \ {}_?{\approx}_?\ ga\}, [X{\mapsto}a])$. More examples are given in Figure 1; they are a quote from the 'Quiz' in [41]. Although the presentation is slightly different, the solutions are equivalent to the ones described in [41].

$$
\begin{array}{rl}
\lambda[a]\lambda[b](Xb) \;{}_?{\approx}_?\; \lambda[b]\lambda[a](aX) & \\
\xRightarrow{({}_?{\approx}_?\mathbf{f},{}_?{\approx}_?\mathbf{absb})} & \{\lambda[a](((b\ a){\cdot}X)a) \;{}_?{\approx}_?\; \lambda[a](aX),\ b\#\lambda[b](Xb)\} \\
\xRightarrow{({}_?{\approx}_?\mathbf{f},{}_?{\approx}_?\mathbf{absa},{}_?{\approx}_?\mathbf{f},\#\mathbf{f},\#\mathbf{absa})} & \{(b\ a){\cdot}X \;{}_?{\approx}_?\; a,\ a \;{}_?{\approx}_?\; X\} \\
\xRightarrow{X\mapsto b} & \{a \;{}_?{\approx}_?\; b\} \not\Longrightarrow \qquad \textit{Solution: None}
\end{array}
$$

$$
\begin{array}{rl}
\lambda[a]\lambda[b](Xb) \;{}_?{\approx}_?\; \lambda[b]\lambda[a](aY) & \\
\xRightarrow{({}_?{\approx}_?\mathbf{f},{}_?{\approx}_?\mathbf{absb})} & \{\lambda[a](((b\ a){\cdot}X)a) \;{}_?{\approx}_?\; \lambda[a](aY),\ b\#\lambda[b](Xb)\} \\
\xRightarrow{({}_?{\approx}_?\mathbf{f},{}_?{\approx}_?\mathbf{absa},{}_?{\approx}_?\mathbf{f},\#\mathbf{f},\#\mathbf{absa})} & \{(b\ a){\cdot}X \;{}_?{\approx}_?\; a,\ a \;{}_?{\approx}_?\; Y\} \\
\xRightarrow{X\mapsto b} & \{\ a \;{}_?{\approx}_?\; Y\} \\
\xRightarrow{[Y\mapsto a]} & \{\} \qquad \textit{Solution: } (\emptyset,\ [X{\mapsto}b, Y{\mapsto}a])
\end{array}
$$

$$
\begin{array}{rl}
\lambda[a]\lambda[b](bX) \;{}_?{\approx}_?\; \lambda[b]\lambda[a](aY) & \\
\xRightarrow{({}_?{\approx}_?\mathbf{f},{}_?{\approx}_?\mathbf{absb})} & \{\lambda[a](a((b\ a){\cdot}X)) \;{}_?{\approx}_?\; \lambda[a](aY),\ b\#\lambda[b](bX)\} \\
\xRightarrow{({}_?{\approx}_?\mathbf{f},{}_?{\approx}_?\mathbf{absa},{}_?{\approx}_?\mathbf{f},\#\mathbf{f},\#\mathbf{absa})} & \{a \;{}_?{\approx}_?\; a,\ (b\ a){\cdot}X \;{}_?{\approx}_?\; Y\} \\
\xRightarrow{({}_?{\approx}_?\mathbf{a}),\ Y\mapsto(b\ a){\cdot}X} & \{(b\ a){\cdot}X \;{}_?{\approx}_?\; (b\ a){\cdot}X\} \\
\xRightarrow{({}_?{\approx}_?\mathbf{X})} & \{\} \qquad \textit{Solution: } (\emptyset,\ [Y{\mapsto}(b\ a){\cdot}X])
\end{array}
$$

$$
\begin{array}{rl}
\lambda[a]\lambda[b](bX) \;{}_?{\approx}_?\; \lambda[a]\lambda[a](aY) & \\
\xRightarrow{({}_?{\approx}_?\mathbf{f},{}_?{\approx}_?\mathbf{absa})} & \{\lambda[b](bX) \;{}_?{\approx}_?\; \lambda[a](aY)\} \\
\xRightarrow{({}_?{\approx}_?\mathbf{f},{}_?{\approx}_?\mathbf{absb})} & \{a((b\ a){\cdot}X) \;{}_?{\approx}_?\; aY,\ a\#bX\} \\
\xRightarrow{({}_?{\approx}_?\mathbf{f},{}_?{\approx}_?\mathbf{a}),\ Y\mapsto(b\ a){\cdot}X} & \{(b\ a){\cdot}X \;{}_?{\approx}_?\; (b\ a){\cdot}X,\ a\#bX\} \\
\xRightarrow{({}_?{\approx}_?\mathbf{X},\#\mathbf{f},\#\mathbf{ab})} & \{a\#X\} \qquad \textit{Solution: } (\{a\#X\},\ [Y{\mapsto}(b\ a){\cdot}X])
\end{array}
$$

Fig. 1. Examples of the unification algorithm in action.

We now show that the unification algorithm checks whether a problem is solvable or not, and moreover it computes a principal, idempotent solution, if one exists. Thus the particular reduction path does not matter; we make some canonical but **arbitrary choice** and use it silently henceforth, for example we may talk about 'the normal form' of a problem.

Lemma 33 (Preservation of solutions)

If $Pr \Longrightarrow Pr'$ *using a simplification rule then* $\mathcal{U}(Pr) = \mathcal{U}(Pr')$.

Proof For simplicity suppose $Pr = \{P\}$ (i.e. it contains only one problem). Suppose $\Gamma \vdash P\sigma$. The derivation is syntax-directed and follows the simplification rules (see Definition 10), so it suffices to check the 12 simplification rules. All the cases are trivial, *except* for the final simplification rule for freshness and the final simplification rule for equality.

(1) Suppose $Pr = \{a\#\pi{\cdot}X\} \Longrightarrow Pr' = \{\pi^{-1}{\cdot}a\#X\}$. Suppose $\Gamma \vdash a\#(\pi{\cdot}X)\sigma$. Then we use part 2 of Lemma 20.
(2) Suppose $Pr = \{\pi{\cdot}X \ {}_?{\approx}_?\ \pi'{\cdot}X\} \Longrightarrow Pr' = \{a_1\#X, \ldots, a_n\#X\}$ where $ds(\pi, \pi') = \{a_1, \ldots, a_n\}$. Suppose $\Gamma \vdash \pi{\cdot}X\sigma \approx_\alpha \pi'{\cdot}X\sigma$. The result follows by Lemma 34 below.

*

This result is the converse of Lemma 21.

Lemma 34 *If $\nabla \vdash \pi{\cdot}s \approx_\alpha \pi'{\cdot}s$ then $\nabla \vdash a\#s$ for each $a \in ds(\pi, \pi')$.*

Proof By induction on the structure of s.

(1) If $s \equiv X$ then the derivation concludes in $(\approx_\alpha \mathbf{X})$. The result is immediate.
(2) If $s \equiv c$ then $\pi{\cdot}c \equiv \pi'{\cdot}c$ so $c \notin ds(\pi, \pi')$ and thus any $a \in ds(\pi, \pi')$ is not the same as c and the result follows using $(\#\mathbf{ab})$.
(3) If $s \equiv (t_1, \ldots, t_n)$ then the derivation must conclude in $(\approx_\alpha \mathbf{tup})$ and $\nabla \vdash \pi{\cdot}t_i \approx_\alpha \pi'{\cdot}t_i$ for $1 \leq i \leq n$. By the inductive hypothesis $\nabla \vdash a\#t_i$ for $1 \leq i \leq n$ and each $a \in ds(\pi, \pi')$. Finally we deduce $\nabla \vdash a\#(t_1, \ldots, t_n)$ for each $a \in ds(\pi, \pi')$ using $(\#\mathbf{tup})$.
(4) The case $s \equiv f(t)$ is similar.
(5) If $s \equiv [c]t$ then there are two cases:
   (a) If $\pi{\cdot}c \equiv \pi'{\cdot}c$ then the reasoning is much as for tuples above.
   (b) If $\pi{\cdot}c \not\equiv \pi'{\cdot}c$ (i.e. $c \in ds(\pi, \pi')$) then the derivation must conclude in $(\approx_\alpha \mathbf{absb})$, so $\nabla \vdash d'\#\pi{\cdot}t$ and $\nabla \vdash (d'\ d){\cdot}\pi{\cdot}t \approx_\alpha \pi'{\cdot}t$ where we set $d \equiv \pi{\cdot}c$ and $d' \equiv \pi'{\cdot}c$.

   By the inductive hypothesis $\nabla \vdash a\#t$ for each $a \in ds((d'\ d) \circ \pi, \pi')$. Now it is a fact that $ds((d'\ d) \circ \pi, \pi')$ is those atoms in $ds(\pi, \pi')$ not equal to c or $\pi^{-1}{\cdot}\pi'{\cdot}c$. Therefore by inductive hypothesis, $\nabla \vdash a\#t$ for each $a \in ds(\pi, \pi')$ not equal to c or $\pi^{-1}{\cdot}\pi'{\cdot}c$ and using $(\#\mathbf{absa})$ and/or $(\#\mathbf{absb})$, we have $\nabla \vdash a\#[c]t$ for the same a.

   We also have $\nabla \vdash \pi^{-1}{\cdot}\pi'{\cdot}c\#t$ by the above and using part 2 of Lemma 20, so $\nabla \vdash a\#[c]t$ for $a \equiv \pi^{-1}{\cdot}\pi'{\cdot}c$.[5] Then, $\nabla \vdash \pi{\cdot}c\#[c]t$ by $(\#\mathbf{absa})$.

*

[5] This is in $ds(\pi, \pi')$, since if $\pi{\cdot}\pi^{-1}{\cdot}\pi'{\cdot}c \equiv \pi'{\cdot}\pi^{-1}{\cdot}\pi'{\cdot}c$ then $\pi'{\cdot}c \equiv \pi'{\cdot}\pi^{-1}{\cdot}\pi'{\cdot}c$ so $\pi{\cdot}c \equiv \pi'{\cdot}c$, and this is not the case.

Theorem 35 *Let Pr be a unification problem, and suppose $\langle Pr\rangle_{sol} = (\Delta, \sigma)$. Then:*

(1) $(\Delta, \sigma) \in \mathcal{U}(Pr)$.
(2) Also $(\Delta, \sigma) \leq (\Delta', \sigma')$ *for all other* $(\Delta', \sigma') \in \mathcal{U}(Pr)$. *That is, the solution is also a* least *or* principal *solution.*

Proof We work by induction on the length of the reduction $Pr \stackrel{*}{\Longrightarrow} \langle Pr\rangle_{sol}$.

- Suppose Pr is in normal form. Then:

(1) Trivially $Pr = \Delta$ and $\sigma = \mathtt{Id}$, and equally trivially $\Delta \vdash Pr\mathtt{Id}$ and $\mathtt{Id}$ is idempotent.
(2) For any other $(\Delta', \sigma') \in \mathcal{U}(Pr)$ trivially σ' is such that $\Delta' \vdash \Delta\sigma$ and $\Delta' \vdash X\mathtt{Id}\sigma \approx_\alpha X\sigma$ for all X.

- Suppose $Pr \Longrightarrow Pr'$ by some non-instantiating simplification. Then using Lemma 33, we know that $\mathcal{U}(Pr) = \mathcal{U}(Pr')$. Both parts of the result follow by induction.
- Suppose $Pr \stackrel{\theta}{\Longrightarrow} Pr'\theta$ by an instantiating rule. So $Pr = \{\pi{\cdot}X \approx_\alpha u\} \cup Pr'$ where $\theta = [X{\mapsto}\pi^{-1}{\cdot}u]$ and $X \notin V(u)$. Suppose $\langle Pr'\theta\rangle_{sol} = (\Delta, \sigma)$, so that by construction $\langle Pr\rangle_{sol} = (\Delta, \theta \circ \sigma)$.

(1) It is easy to see that $\theta \circ \sigma$ is idempotent and by the first part of the inductive hypothesis $\Delta \vdash Pr'\theta\sigma$, that is, $(\Delta, \theta \circ \sigma) \in \mathcal{U}(Pr)$.
(2) Suppose $(\Delta', \sigma') \in \mathcal{U}(Pr)$. Then $\Delta' \vdash X\sigma' \approx_\alpha \pi^{-1}{\cdot}u\sigma'$ by part 2 of Lemma 20.

By part 1 of the technical lemma which follows (Lemma 36) and using its notation, $(\Delta', \theta{\circ}\sigma'') \in \mathcal{U}(Pr)$ where σ'' acts just like σ' only it maps X to X, $\theta = [X{\mapsto}u]$, and $\sigma' = \theta{\circ}\sigma''$. By part 2 Lemma 36, $(\Delta', \sigma'') \in \mathcal{U}(Pr'\theta)$ and by inductive hypothesis $(\Delta, \sigma) \leq (\Delta', \sigma'')$. By part 4 it follows that $(\Delta, \theta \circ \sigma) \leq (\Delta', \theta \circ \sigma'')$.

*

Lemma 36 *(1) Suppose σ' is idempotent and $X\sigma' \not\equiv X$. Then if $\Delta' \vdash Pr\sigma'$ and $\Delta' \vdash X\sigma' \approx_\alpha u\sigma'$, then $\Delta' \vdash Pr(\theta \circ \sigma'')$ where $\theta = [X{\mapsto}u]$ and σ'' acts just like σ', only $X\sigma'' \equiv X$. Furthermore, $\sigma' = \theta \circ \sigma''$.*
(2) Continuing the assumptions and notation above, if $(\Delta', \theta \circ \sigma'') \in \mathcal{U}(Pr)$ and $Pr = Pr' \cup \{\pi{\cdot}X \;?{\approx}?\; u\}$, then $(\Delta', \sigma'') \in \mathcal{U}(Pr'\theta)$.
(3) For all Δ', σ_1, and σ_2, if $\Delta' \vdash Y\sigma_1 \approx_\alpha Y\sigma_2$ for all Y, then $\Delta' \vdash u\sigma_1 \approx_\alpha u\sigma_2$ for all u. ('Two α-equivalent substitutions on a single term give two α-equivalent terms.')
(4) For all Δ, σ, Δ', and σ'', if $(\Delta, \sigma) \leq (\Delta', \sigma'')$ then $(\Delta, \theta{\circ}\sigma) \leq (\Delta', \theta{\circ}\sigma'')$.

Proof

(1) We observe that $X\sigma' \approx_\alpha u\sigma' \approx_\alpha X\theta\sigma''$ and for any other Y, $Y\sigma' \equiv Y\sigma''$.
(2) By part 1 of Corollary 25, and using idempotence.

(3) By part 1 of Corollary 25.
(4) By definition if $(\Delta, \sigma) \leq (\Delta', \sigma'')$ then for some τ, $\Delta' \vdash \Delta\sigma\tau$ and $\Delta' \vdash Y\sigma\tau \approx_\alpha Y\sigma''$ for all Y. The result follows by the previous part of this lemma.

*

We conclude with two routine but important results:

Lemma 37 *(1) If Eq is a non-empty set of inconsistent unification constraints then it has no solution.*
(2) If Γ is an inconsistent context then it has no solution.

Proof By definition, a solution to Eq, if it exists, must be of the form (Δ, σ) for some freshness context Δ such that $\Delta \vdash Eq\sigma$, where here we are a bit lax and convert the unification problems in Eq into equality problems. Lemma 32 tells us what form $Eq\sigma$ can take and we check that each kind of problem corresponds to a reduced equality problem. The second part of Theorem 17 then tells us that $\Delta \vdash Eq\sigma$ simply cannot happen.

Now suppose Γ is an inconsistent context. By Lemma 32 $\langle\Gamma\rangle_{sol} = (\langle\Gamma\rangle_{nf}, \mathtt{Id})$. Suppose this *is* a solution to Γ, so that $\langle\Gamma\rangle_{nf} \vdash \Gamma$ and $\langle\Gamma\rangle_{nf}$ is consistent. This is not possible since no derivation rule allows us to derive an inconsistent constraint from a consistent context. *

Corollary 38 *Let Pr be a unification problem such that $\langle Pr\rangle_{sol} = (\Delta \cup Contr \cup Eq, \sigma)$. Then $\mathcal{U}(Pr)$ is nonempty if and only if $Contr \cup Eq = \emptyset$.*

Proof The right-to-left implication follows by Theorem 35. For the left-to-right implication we work by induction on the length of the reduction $Pr \stackrel{*}{\Longrightarrow} \langle Pr\rangle_{sol}$.

- Suppose Pr is in normal form. Then by Lemma 32 $Pr = \Delta \cup Contr \cup Eq$.
  If $Contr \cup Eq$ is nonempty then by Lemma 37 $\mathcal{U}(Pr)$ is empty. Conversely if $Contr \cup Eq$ is empty, we observe trivially that $\Delta \vdash \Delta\mathtt{Id}$ and $\mathcal{U}(Pr)$ contains $(\Delta, \mathtt{Id})$.
- Suppose $Pr \Longrightarrow Pr'$ by some non-instantiating simplification. Then using Lemma 34 we know that $\mathcal{U}(Pr) = \mathcal{U}(Pr')$. We use the inductive hypothesis.
- Suppose $Pr \stackrel{\theta}{\Longrightarrow} Pr\theta$ by an instantiating simplification, so $Pr = \{\pi{\cdot}X \approx_\alpha u\} \cup Pr'$ and $\theta = [X \mapsto \pi^{-1}{\cdot}u]$.
  Suppose $\Delta' \vdash Pr\sigma'$. Then $\Delta' \vdash Pr\theta\sigma''$ where σ'' acts just like σ' only $X\sigma'' \equiv X$, and $(\Delta', \sigma'') \in \mathcal{U}(Pr'\theta)$. The result follows by the inductive hypothesis for $Pr'\theta$.

*

Rewriting needs a notion of matching; we develop a suitable one in §5.2.

5 Rewriting

5.1 Rewrite rules

We will define a notion of rewriting which operates on 'terms-in-consistent-contexts': a pair (Δ, s) of a consistent context and a term, which we write $\Delta \vdash s$. Then $\Delta \vdash s$ rewrites to $\Delta \vdash t$ — a freshness context is fixed. Since in a particular rewriting path the context is fixed, a given context defines a particular rewrite relation $\Delta \vdash \text{-} \to \text{-}$ which we define below.

Definition 39 *A **nominal rewrite rule** $R \equiv \nabla \vdash l \to r$ is a tuple of*

- *a consistent context ∇; and*
- *terms l and r such that $V(r, \nabla) \subseteq V(l)$.*

We now develop a theory of nominal rewriting; we will see in Section 6 that a *uniformity* condition on R becomes useful for rewriting to be truly well-behaved. However, the 'engine' driving rewriting remains what we now construct.

Example 40 In this example we will use the signature of ML and the syntactic sugar defined in Example 2.

(1) $a\#X \vdash (\lambda[a]X)Y \to X$ is a form of trivial β-reduction.
(2) $a\#X \vdash X \to \lambda[a](Xa)$ is η-expansion.
(3) Of course a rewrite rule may define any arbitrary transformation of terms, and may have an empty context, for example $\emptyset \vdash XY \to XX$.
(4) $a\#Z \vdash X\lambda[a]Y \to X$ is not a rewrite rule, because $Z \notin V(X\lambda[a]Y)$. $\emptyset \vdash X \to Y$ is also not a rewrite rule.
(5) $\emptyset \vdash a \to b$ *is* a rewrite rule. We mention this again below.

We can now write

$$(\nabla \vdash l \to r)\{X{\mapsto}s\} \quad \stackrel{\text{def}}{=} \quad \langle \nabla\{X{\mapsto}s\}\rangle_{nf} \vdash l\{X{\mapsto}s\} \to r\{X{\mapsto}s\}.$$

We shall never write the substitution in such detail, but this is how we instantiate rules.

As usual we shall consider rules up to permutative renaming of their variable symbols X, Y. Thus $a\#X \vdash (\lambda[a]X)Y \to X$ and $a\#Y \vdash (\lambda[a]Y)X \to Y$ are 'morally' the same rule.

Similarly it is convenient to consider atoms a, b up to permutative renamings, so that if we have a rule $a\#X \vdash X \to \lambda[a](Xa)$ then also $b\#X \vdash X \to \lambda[b](Xb)$ is available.

It will be useful to have a notation for permuting atoms in the syntax of rules and terms, as we just did above: Write $R^{(a\,b)}$ for that rule obtained by swapping a and b in R throughout. For example, if $R \equiv b\#X \vdash [a]X \to (b\ a)\cdot X$ then $R^{(a\,b)} \equiv a\#X \vdash [b]X \to (a\ b)\cdot X$. Write R^{π} for that rule obtained by applying π to the atoms in R according to the swapping action. Also write s^{π} for that term obtained by applying π to the atoms in s.

A simple technical lemma will be useful in Theorem 50, we mention it now:

Lemma 41 *$\Delta \vdash \pi\cdot s \approx_{\alpha} s^{\pi}\sigma$ for any Δ, where $X\sigma = \pi\cdot X$ for each X mentioned in s.*

The proof is by an easy induction on s and can be illustrated by an example: if we take $s \equiv (a\ b)\cdot X$ and $\pi = (a\ c)$ then $\pi\cdot s \equiv (a\ c)(a\ b)\cdot X$ and $s^{\pi}\sigma \equiv (c\ b)(a\ c)\cdot X$. The suspended permutations are not identical, but their difference set is empty and the result follows.

Definition 42 *A set of rewrite rules is **equivariant** when it is closed under $(-)^{(a\,b)}$ for all atoms a and b.*

A **nominal rewrite system** $(\Sigma, \mathcal{R})$ *consists of:*

(1) A nominal signature Σ.
(2) An equivariant set $\mathcal{R}$ of nominal rewrite rules over Σ.

We may drop Σ and write $\mathcal{R}$ for the rewrite system. When we write out the system we (obviously) do not bother to give every possible permutation of variables and atoms. Indeed, given any set of rewrite rules we can always obtain an equivariant one by closing under the permutation action $(-)^{(a\,b)}$ outlined above (call this the **equivariant closure** of the set of rules). We shall generally elide this step and equate a (finite, non-equivariant) set of rewrite rules with its equivariant closure.

Note that rewrite systems are "metalevel equivariant", as opposed to the "internal equivariance" of predicates $\#$ and $\approx_{\alpha}$ shown in Lemma 20 part 1. In other words, we define a rewrite system as an equivariant set of rules, whereas we can *prove* that $\#$ and α are preserved by permutations.

Example 43 To give a small-step evaluation relation for our fragment of ML (see Examples 2 and 40) we extend it with a term-former for (explicit)

substitutions `sub` which we sugar to $t\{a{\mapsto}t'\}$. The rewrite rules:

$$\begin{array}{lll}
(\mathsf{Beta}) & (\lambda[a]X)X' & \to X\{a{\mapsto}X'\} \\
(\sigma_{\mathtt{app}}) & (XX')\{a{\mapsto}Y\} & \to X\{a{\mapsto}Y\}X'\{a{\mapsto}Y\} \\
(\sigma_{\mathtt{var}}) & a\{a{\mapsto}X\} & \to X \\
(\sigma_{\epsilon}) \quad a\#Y \vdash & Y\{a{\mapsto}X\} & \to Y \\
(\sigma_{\mathtt{lam}}) \quad b\#Y \vdash & (\lambda[b]X)\{a{\mapsto}Y\} & \to \lambda[b](X\{a{\mapsto}Y\})
\end{array}$$

define a system of explicit substitutions for the λ-calculus with names. We add the following rules:

$$\begin{array}{rl}
(\mathsf{Let}) & \mathtt{let}\ a = X'\ \mathtt{in}\ X \to X\{a{\mapsto}X'\} \\
(\mathsf{Letrec}) & \mathtt{letrec}\ fa = X'\ \mathtt{in}\ X \to \\
& \qquad X\{f{\mapsto}(\lambda[a]\mathtt{letrec}\ fa = X'\ \mathtt{in}\ X')\} \\
(\sigma_{\mathtt{let}}) \quad a\#Y \vdash & (\mathtt{let}\ a = X'\ \mathtt{in}\ X)\{b{\mapsto}Y\} \to \\
& \qquad \mathtt{let}\ a = X'\{b{\mapsto}Y\}\ \mathtt{in}\ X\{b{\mapsto}Y\} \\
(\sigma_{\mathtt{letrec}}) \quad f\#Y, a\#Y \vdash & (\mathtt{letrec}\ fa = X'\ \mathtt{in}\ X)\{b{\mapsto}Y\} \to \\
& \qquad \mathtt{letrec}\ fa = X'\{b{\mapsto}Y\}\ \mathtt{in}\ X\{b{\mapsto}Y\}
\end{array}$$

Example 44 We can define a signature for first-order logic with term-formers $\forall$, $\exists$, $\neg$, $\wedge$, $\vee$, and rewrite rules to compute prenex normal forms (here we use variables P, Q) :

$$\begin{array}{l}
a\#P \vdash P \wedge \forall[a]Q \to \forall[a](P \wedge Q) \\
a\#P \vdash (\forall[a]Q) \wedge P \to \forall[a](Q \wedge P) \\
a\#P \vdash (P \vee \forall[a]Q \to \forall[a](P \vee Q) \\
a\#P \vdash ((\forall[a]Q) \vee P \to \forall[a](Q \vee P) \\
a\#P \vdash (P \wedge \exists[a]Q \to \exists[a](P \wedge Q) \\
a\#P \vdash ((\exists[a]Q) \wedge P \to \exists[a](Q \wedge P) \\
a\#P \vdash (P \vee \exists[a]Q \to \exists[a](P \vee Q) \\
a\#P \vdash ((\exists[a]Q) \vee P \to \exists[a](Q \vee P) \\
\qquad \vdash (\neg(\exists[a]Q) \to \forall[a]\neg Q \\
\qquad \vdash (\neg(\forall[a]Q) \to \exists[a]\neg Q
\end{array}$$

We could also add rules:

$$a\#X \vdash \forall[a]X \to X$$
$$\vdash \forall[a]X \wedge \forall[a]Y \to \forall[a](X \wedge Y)$$

We recapitulate aspects of nominal terms and rules which are unusual with respect to a first-order system:

(1) Moderated variables $(a\ b)\cdot X$, which let us 'suspend' renamings.
(2) The unusual term constructor abstraction $[a]t$.
(3) The freshness side-conditions, such as $a\#X$. We use them to avoid accidental variable capture.
(4) The α-equivalence relation $\approx_\alpha$, which uses all three of the above.

We now define the process of rewriting.

5.2 Matching problems, and rewriting steps

We do want a rewrite system to induce some actual *rewrites* on the set of terms in its signature. Here's how:

Definition 45 *A **matching problem (in context)** is a pair*

$$(\nabla \vdash l)\ {}_?{\approx}\ (\Delta \vdash s)$$

*where ∇, Δ are consistent contexts and l, s are nominal terms. The **solution** to this matching problem, if it exists, is a substitution θ such that:*

- $\langle \nabla,\ l\,{}_?{\approx}_?\, s\rangle_{sol} = (\Delta', \theta)$.
- $\Delta \vdash \Delta'$.
- $X\theta \equiv X$ *for* $X \in V(\Delta, s)$.

*We say that θ **solves** the matching problem.*

Note that a matching problem can be seen as a particular kind of unification problem. The conditions in the definition above ensure that: $\Delta \vdash l\theta \approx_\alpha s$ and $\Delta \vdash \nabla\theta$, and so $(\Delta, \theta) \in \mathcal{U}(\nabla,\ l\,{}_?{\approx}_?\, s)$. We can think of the solution to $(\nabla \vdash l)\ {}_?{\approx}\ (\Delta \vdash s)$ as being a most general θ such that (Δ, θ) solves $\nabla,\ l\,{}_?{\approx}_?\, s$. For notation and terminology see §4.

Remark: This is more than just matching modulo α-conversion because we can use ∇ to specify constraints which must be satisfied by the matching solution. When the conditions in ∇ are satisfied we say the matching **is triggered**.

(Soon, $\nabla \vdash l$ will be the left-hand side of a rewrite rule $\nabla \vdash l \to r$, and then we say the rule is **triggered**.) $*$

Example 46 (1) $(\vdash a) \ {}_?{\approx} \ (\vdash b)$ has no solution.
(2) $(\vdash [a]a) \ {}_?{\approx} \ (\vdash [b]b)$ has a solution $\theta = \mathtt{Id}$.
(3) $(\vdash [a][b]X') \ {}_?{\approx} \ (\vdash [b][a]X)$ has solution $\theta = [X' \mapsto (a\ b)\cdot X]$.
(4) $(a\#X \vdash [a]X) \ {}_?{\approx} \ (\vdash [a]a)$ has no solution (because the only candidate, $[X \mapsto a]$, causes the condition $a\#X$ to become inconsistent).

Say a term has a **position** when it mentions a distinguished unknown, we usually write it -, precisely once, and with trivial moderation. We let capital letters L, C, P vary over terms with a position. We write $C[s]$ for $C[\text{-} \mapsto s]$, and [-] when the term C is precisely its unique variable. Since the term C is only of interest inasmuch as - may be substituted for a term, we shall tend to **silently assume** that - is fresh.

For example, $[a](a, \text{-})$ has a position, but not $(\text{-}, \text{-})$ or $(a\ b)\cdot\text{-}$. [6]

Definition 47 *Suppose $R = \nabla \vdash l \to r$ is a rewrite rule, s and t are terms, and Δ is a consistent context. We say s **rewrites with** R **to** t **in the context** Δ, and we write $\Delta \vdash s \xrightarrow{R} t$ when:*

(1) $V(R) \cap V(\Delta, s) = \emptyset$ (we can assume this with no loss of generality).
(2) $s \equiv C[s']$ for some position $C[\text{-}]$, and term s', such that θ solves $(\nabla \vdash l) \ {}_?{\approx} \ (\Delta \vdash s')$.
(3) $\Delta \vdash C[r\theta] \approx_\alpha t$.

If $C \equiv [\text{-}]$ we say the **the rewrite occurs at the root position**. Otherwise we may (semi-formally) say that **the rewrite occurs at** C.

Given a nominal rewrite system $\mathcal{R}$ say that s **rewrites to** t in a context Δ, and write $\Delta \vdash s \xrightarrow{\mathcal{R}} t$ or just $\Delta \vdash s \to t$, when there is a rule $R \in \mathcal{R}$ such that $\Delta \vdash s \xrightarrow{R} t$.

The rewrite relation $\to^*$ is the reflexive and transitive closure of this relation. A **normal form** is a term-in-context that does not rewrite.

We now give some examples of rewrite steps:

Example 48 (1) It is easy to show that

$$\emptyset \vdash (\lambda[a]f(a,a))\ X \to^* f(X, X)$$

[6] A 'position' is, literally, the standard notion of a point in the abstract syntax tree of a term, as defined for example in [21]. It is more convenient for us to identify this with the corresponding 'initial segment' of a term.

in four steps using the rules (Beta) and ($\sigma_{\mathtt{var}}$) of Example 43 together with a rule for the propagation of substitutions under f:

$$(\sigma_f) \quad f(X, X')\{a{\mapsto}Y\} \to f(X\{a{\mapsto}Y\}, X'\{a{\mapsto}Y\})$$

In a CRS a similar reduction is done in one step, using a higher-order substitution mechanism which involves some β-reductions. NRSs use first-order substitutions and therefore we have to define explicitly the substitution mechanism, but in contrast with first-order TRSs we don't need to make explicit the α-conversions. For instance, rule ($\sigma_{\mathtt{lam}}$) (see Example 43) pushes a substitution under a λ avoiding capture, as the following rewrite step shows:

$$b\#Z \vdash (\lambda[c]Z)\{a{\mapsto}c\} \to \lambda[b](((b\ c){\cdot}Z)\{a{\mapsto}c\})$$

(2) A pathological but illuminating example of rewriting rule is $\emptyset \vdash X \to X$. Then $\emptyset \vdash \lambda[a]a \to \lambda[a]a$, but we can also verify that $\emptyset \vdash \lambda[a]a \to \lambda[b]b$. In general, in the presence of this rule, if $\Delta \vdash s \approx_\alpha t$ then $\Delta \vdash s \to t$, for example $a, b\#X \vdash X \to (a\ b){\cdot}X$.

This will not cause problems in confluence results because they are also defined up to $\approx_\alpha$.

(3) If $\emptyset \vdash a \to b$ is in an equivariant set of rewrite rules, we have a rewrite step $\emptyset \vdash a' \to b'$ for any pair of different atoms a' and b'. Our notion of matching does *not* instantiate, or even permutatively rename, atoms; however, equivariance of the rule system as a whole guarantees that if a rule exists then a rule with permutatively renamed atoms is available.

Nominal rewriting systems are more expressive than first-order systems, as Examples 43 and 44 show. They are also more expressive than standard higher-order formats, as Example 49 shows.

Example 49 We add to the signature of the λ-calculus with names (see Example 43) a second operator for substitution, `csub`, representing context substitution, which does not avoid capture. We introduce a unary term-former $-$ to represent 'a hole' in a λ-term, and we abbreviate $-(Z)$ to $-_Z$. We write $\mathtt{csub}(C, t)$, which we sugar as $C[t]$; we are supposed to think of this as 'replace the hole $-$ in C by t'. We specify this intuition formally using nominal rewrite rules:

$$\begin{aligned}
&\vdash -_Z[X] && \to X\\
&\vdash (\lambda[a]{-_Z})[X] && \to \lambda[a]X\\
&\vdash (X\ {-_Z})[X'] && \to X\ X'\\
&\vdash ({-_Z}\ X)[X'] && \to X'\ X
\end{aligned}$$

Note that the second rule will capture any a occurring in an instance of X.

It is hard to see how these rules would work in a formalism in which terms are taken to be α-equivalence classes.

Note that we take $-$ to be a unary term-former and *not* a constant (a 0-ary term-former); for suppose we just took $-$ as a constant. Then $\lambda[a]- \approx_\alpha \lambda[b]-$. This is not the α-equivalence behaviour we expect of a binder with a hole in its scope and it would lead to incorrect rewrites such as $(\lambda[a]-)[b] \to \lambda[b]b$.

Another solution would be to pick some distinguished variable, call it $-$, and use that for our hole. We would also have to restrict the instantiation behaviour of the nominal rewriting machinery, to prevent $(\lambda[a]a)[b]$ rewriting to $\lambda[a]b$ with the rule $(\lambda[a]-)[X] \to \lambda[a]X$.[7]

Usually, the one-step rewrite relation generated by a set of rules is defined as the "compatible closure" of a set of rules, that is, the closure of the rewrite rules by context and substitution (see for instance [16]). The definition of nominal rewriting given above satisfies these properties (taking the freshness context of the rule into account), and is also closed under permutation, as the following theorem shows:

Theorem 50 *Assume* $\Delta \vdash s \xrightarrow{R} t$ *using the rule* $R \equiv \nabla \vdash l \to r$*, then:*

(1) $\Delta \vdash C[s] \xrightarrow{R} C[t]$*. More generally, if* $\Delta \vdash s \xrightarrow{R} t$ *and* $\Delta \vdash C[t] \approx_\alpha D$*, then* $\Delta \vdash C[s] \xrightarrow{R} D$*.*
(2) If Γ *is consistent and* $\Gamma \vdash \Delta\sigma$*, then* $\Gamma \vdash s\sigma \xrightarrow{R} t\sigma$*.*
(3) $\Delta \vdash \pi{\cdot}s \xrightarrow{R^\pi} \pi{\cdot}t$*.*

Proof

(1) Intuitively this is obvious, since part 2 of Definition 47 allows for any context, and part 3 allows for any α-equivalent term on the right.

Formally: since $\Delta \vdash s \xrightarrow{R} t$, $s \equiv C'[s']$ and there exists some θ solving $(\nabla \vdash l) \ {}_?{\approx}\ (\Delta \vdash s')$. That is, $\Delta \vdash \nabla\theta$ and $\Delta \vdash l\theta \approx_\alpha s'$ and $\Delta \vdash C'[r\theta] \approx_\alpha t$.

Then $\Delta \vdash \nabla\theta$ and $\Delta \vdash C[C'[l\theta]] \approx_\alpha C[C'[s']]$ and $\Delta \vdash C[C'[r\theta]] \approx_\alpha C[t] \approx_\alpha D$, using Theorem 24, and the result follows by Definition 47.

(2) So $s \equiv C'[s']$ and there exists some θ such that $\Delta \vdash \nabla\theta$ and $\Delta \vdash l\theta \approx_\alpha s'$ and $\Delta \vdash C'[r\theta] \approx_\alpha t$.

Then $s\sigma \equiv C''[s'\sigma]$ where C'' is $C'\sigma$.[8]

[7] So we have yet another kind of hole, similar to the X of nominal rewriting in that it represents an unknown term, but this is one which we expressly do *not* want to match with any *particular* term. Another day, another paper. This idea can be emulated quite effectively in nominal rewriting as we have done, with a unary term-former.

[8] ... assuming $\text{-}\sigma \equiv \text{-}$, which should be the case since we assume - 'is always fresh

Suppose that $\Delta\sigma$ is consistent. Then by Lemma 22 we know $\langle\Delta\sigma\rangle_{nf} \vdash \nabla\theta\sigma$, $\langle\Delta\sigma\rangle_{nf} \vdash l\theta\sigma \approx_\alpha s'\sigma$, and $\langle\Delta\sigma\rangle_{nf} \vdash C''[r\theta\sigma] \approx_\alpha t\sigma$.

Now suppose Γ is consistent and $\Gamma \vdash \Delta\sigma$. Then $\Delta\sigma$ is consistent by Lemma 16 and the result now follows using part 3 of Corollary 18.

(3) So $s \equiv C'[s']$ and there exists some θ such that $\Delta \vdash \nabla\theta$ and $\Delta \vdash l\theta \approx_\alpha s'$ and $\Delta \vdash C'[r\theta] \approx_\alpha t$.

Write $(\pi\cdot\theta)$ for the substitution such that if $X\theta \equiv X$ then $X(\pi\cdot\theta) \equiv X$, and if $X\theta \not\equiv X$ then $X(\pi\cdot\theta) \equiv \pi\cdot(X\theta)$. Note that:

- Because of conditions on disjointness of variables in the matching problem which θ solves, $X\theta \not\equiv X$ for every X mentioned in R.
- By Lemma 41 it is the case that $\Delta \vdash l^\pi(\pi\cdot\theta) \approx_\alpha \pi\cdot(l\theta)$.
- Similarly $\Delta \vdash r^\pi(\pi\cdot\theta) \approx_\alpha \pi\cdot(r\theta)$.
- Similarly $\Delta \vdash \nabla^\pi(\pi\cdot\theta)$.
- $\Delta \vdash \pi\cdot s \approx_\alpha \pi\cdot(C'[s']) \approx_\alpha C''[\pi\cdot s']$ where C'' is $\pi\cdot C'$ with $\pi\cdot$- replaced by -.

Then by Lemma 20, Lemma 5, and Theorem 24 we can deduce that: $\pi\cdot s \equiv C''[\pi\cdot s']$ and $(\pi\cdot\theta)$ is such that $\Delta \vdash \nabla^\pi(\pi\cdot\theta)$ and $\Delta \vdash l^\pi(\pi\cdot\theta) \approx_\alpha \pi\cdot s'$ and $\Delta \vdash C''[r^\pi(\pi\cdot\theta)] \approx_\alpha \pi\cdot t$. This suffices to prove that $\Delta \vdash \pi\cdot s \xrightarrow{R^\pi} \pi\cdot t$.

*

Another interesting property, closure under $\approx_\alpha$, requires a more restrictive notion of rewrite rule. We come back to this point in Section 6.

5.3 Critical pairs and confluence

Definition 51 *Say a nominal rewrite system is* ***confluent*** *when if $\Delta \vdash s \to^* t$ and $\Delta \vdash s \to^* t'$, then u exists such that $\Delta \vdash t \to^* u$ and $\Delta \vdash t' \to^* u$.*

Confluence is an important property because it ensures unicity of normal forms, a form of determinism. Local confluence is a weaker property, it is defined as 'joinability of peaks'. More precisely:

Definition 52 *Fix an equivariant rewrite system $\mathcal{R}$, and write $\Delta \vdash s \to t_1, t_2$ for the appropriate pair of rewrite judgements. A pair $\Delta \vdash s \to t_1, t_2$ is called a* ***peak****. A nominal rewrite system is* ***locally confluent*** *when, if $\Delta \vdash s \to t_1, t_2$, then u exists such that $\Delta \vdash t_1 \to^* u$ and $\Delta \vdash t_2 \to^* u$. We say such a peak is* ***joinable****.*

Definition 53 *Suppose*

(1) $R_i = \nabla_i \vdash l_i \to r_i$ for $i = 1, 2$ are copies of two rules in $\mathcal{R}$ such that

enough' and rename it otherwise.

$V(R_1) \cap V(R_2) = \emptyset$ *(R_1 and R_2 could be copies of the same rule).*

(2) $l_1 \equiv L[l_1']$ *such that* $\nabla_1, \nabla_2, l_1' \ {}_?{\approx}_?\ l_2$ *has a principal solution* (Γ, θ)*, so that* $\Gamma \vdash l_1'\theta \approx_\alpha l_2\theta$ *and* $\Gamma \vdash \nabla_i\theta$ *for* $i = 1, 2$*.*

Then call the pair of terms-in-context

$$\Gamma \vdash (r_1\theta, L\theta[r_2\theta])$$

a ***critical pair****. If* $L = [\text{-}]$ *and* R_1*,* R_2 *are copies of the same rule, or if* l_1' *is a variable, then we say the critical pair is* ***trivial****.*

Example 54 There are several non-trivial critical pairs in Example 43 involving substitution rules. For instance, there is a critical pair between (σ_ϵ) and $(\sigma_{\mathtt{app}})$, and also between (σ_ϵ) and $(\sigma_{\mathtt{lam}})$.

Definition 55 *We will say that a peak* $\Delta \vdash s \to t_1, t_2$ *is an* ***instance*** *of a critical pair* $\Gamma \vdash (r_1\theta, L\theta[r_2\theta])$ *when there is some* σ *such that:*

- $\Gamma\sigma$ *is consistent.*
- $\Delta \vdash \Gamma\sigma$*.*
- $\Delta \vdash (r_1\theta\sigma, L\theta\sigma[r_2\theta\sigma]) \approx_\alpha (t_1, t_2)$*.*

Another way of phrasing this is that

$$(\Gamma \vdash [X_1{\mapsto}r_1\theta,\ X_2{\mapsto}L\theta[r_2\theta]) \leq (\Delta \vdash [X_1{\mapsto}t_1,\ X_2{\mapsto}t_2])$$

in the instantiation ordering from Definition 28, for two (suitably fresh) variables X_1 and X_2.

A critical pair is a pair of terms which can appear in a peak of a rewrite of a term-in-context. In standard (first-order) rewrite systems any instance of a critical pair gives rise directly to a peak for any substitution, but here, an instance of a critical pair only gives rise to a peak for substitutions σ (continuing the notation above) such that $\Gamma\sigma$ is consistent.

Non-trivial critical pairs are important in first order term rewriting systems because it is sufficient to check their joinability to deduce local confluence. This result extends to nominal rewriting under certain conditions, which we will discuss in the next section.

6 Uniform rewriting (or: 'well-behaved' nominal rewriting)

Nominal rewriting is elementary (and easy to explain) but sometimes we need more. For example:

Lemma 56 *It is* not *necessarily the case that trivial critical pairs are joinable.*

Proof It suffices to give counterexamples. Consider the rules (for some term-former f)

$$R \equiv \ \vdash fb \to a \qquad \text{and} \qquad R' \equiv a\#X \vdash X \to [a]X.$$

These have a trivial critical pair $\vdash (a, [a]fb)$ (the term fb rewrites to both). It is not hard to see that these terms are not joinable. $*$

The fact that the left-hand side of R' is a variable is not a problem, the problem is that R 'creates' an atom a, which invalidates the freshness context in R'. Rules that create free atoms do not work uniformly in $\approx_\alpha$ equivalence classes. For instance, take

$$R \equiv \vdash [b]b \to b$$

and the term $s \equiv [a][b]b$. Then $s \approx_\alpha [b][a]a \equiv s'$ and $s \to [a]b$ but s' does not reduce to $[a]b$.

However, rewrite rules 'in nature' (see Example 43) seem to belong to a restricted class of **uniform** rules, which display good behaviour. We now characterise this better-behaved class of uniform rules and show it has good properties (see Lemma 61).

Definition 57 *Say R is **uniform** when if $\Delta \vdash s \xrightarrow{R} t$ then $\Delta, \langle a\#s\rangle_{nf} \vdash a\#t$ for any a such that $\langle a\#s\rangle_{nf}$ is consistent.*

In the judgements of the form $\Delta, \langle a\#s\rangle_{nf} \vdash a\#t$ below we will always assume that we consider only atoms such that $\langle a\#s\rangle_{nf}$ is consistent, or alternatively, we can think that if the freshness context is inconsistent any predicate is derivable, which corresponds to adding a bottom rule to the logical system.

Remark: All the example rewrite rules 'from nature' cited so far are uniform. $*$

The definition of uniformity looks hard to check — do we really have to consider *all* s and t before we can declare R to be uniform? Fortunately, there is a better way:

Lemma 58 *(1) $R \equiv \nabla \vdash l \to r$ is uniform if and only if $\nabla, \langle a\#l\rangle_{nf} \vdash a\#r$ for all a.*
(2) $R \equiv \nabla \vdash l \to r$ is uniform if and only if $\nabla, \langle a\#l\rangle_{nf} \vdash a\#r$ for all a mentioned in ∇, l, and r, and for one fresh a.

Proof Suppose R is uniform. It is easy to verify that $\nabla \vdash l \xrightarrow{R} r$ (that is, l rewrites with R to r in context ∇). Therefore by assumption, $\nabla, \langle a\#l\rangle_{nf} \vdash a\#r$.

Conversely suppose $\nabla, \langle a\#l\rangle_{nf} \vdash a\#r$. Suppose also that $\Delta \vdash s \xrightarrow{R} t$, so that:

- There is a substitution θ such that $\Delta \vdash \nabla\theta$.
- There is a position L such that $s \equiv L[s']$ and $\Delta \vdash s' \approx_\alpha l\theta$.
- $\Delta \vdash t \approx_\alpha L[r\theta]$.

We know $\langle \nabla\theta\rangle_{nf}, \langle a\#l\theta\rangle_{nf} \vdash a\#r\theta$ by Lemma 22. Also, since $\Delta \vdash \nabla\theta$, using the second part of Corollary 18 also $\Delta, \langle a\#l\theta\rangle_{nf} \vdash a\#r\theta$. Then $\Delta, \langle a\#L[l\theta]\rangle_{nf} \vdash a\#L[r\theta]$ by the technical lemma which follows.

We then conclude that $\Delta, \langle a\#L[l\theta]\rangle_{nf} \vdash a\#t$ for any $\Delta \vdash t \approx_\alpha L[r\theta]$ by Lemma 23.

For the last part, we use equivariance of the definition of uniform rewriting itself [26,25] to see that if $\nabla, \langle a\#l\rangle_{nf} \vdash a\#r$ for some a not mentioned in ∇, l, or r, then $\nabla, \langle a'\#l\rangle_{nf} \vdash a'\#r$ for all other a' not mentioned in ∇, l, or r.[9] *

The following result is useful in the proof above.

Lemma 59 *For any l and r, if $\Gamma, \langle a\#l\rangle_{nf} \vdash a\#r$, then $\Gamma, \langle a\#L[l]\rangle_{nf} \vdash a\#L[r]$.*

Proof We work by induction on the syntax of L.

- If $L = [a]L'$ the result is immediate since $a\#[a](L'[r])$ by (**#absa**).
- If $L = [b]L'$ then we observe that $\langle a\#L[l]\rangle_{nf} = \langle a\#L'[l]\rangle_{nf}$, so we may use the inductive hypothesis and (**#absb**).
- The other cases are easy.

*

Remark: Intuitively uniformity means 'if a is not free in s and s rewrites to t, then a is not free in t', or more concisely: 'uniform rules do not generate atoms'. Note that the following definition is *wrong*: $\nabla \vdash l \to r$ is 'uniform' when $\nabla \vdash a\#l$ implies $\nabla \vdash a\#r$ for all a. The reason is that l and r may contain unknowns, so we must insert *assumptions* about them, e.g. $\langle a\#l\rangle_{nf}$.

For instance, $\vdash X \to a$ is trivially 'uniform' according to the 'wrong' definition, since $\vdash b\#X$ is derivable for no b.

*

[9] Or, if the reader does not care for this degree of rigour, we can just say "since a was fresh but otherwise arbitrary, clearly $\nabla, \langle a\#l\rangle_{nf} \vdash a\#r$ holds for any other a".

As observed in the proof of Lemma 56 the validity of a freshness judgement $a\#s$ can be influenced by changes deep inside s (e.g. as occur in rewriting). With uniform rewriting, this ceases to be a concern:

Lemma 60 *If R is uniform and $\Delta \vdash s \xrightarrow{R} t$ and $\Delta \vdash a\#s$, then $\Delta \vdash a\#t$.*

Proof Suppose $\Delta \vdash a\#s$. By Lemma 16, $a\#s$ is consistent. Also by definition of uniformity $\Delta, \langle a\#s \rangle_{nf} \vdash a\#t$. We now use Corollary 18. $*$

Uniform rewriting is well-behaved:

Theorem 61 *Assume R is uniform.*

(1) If $\Delta \vdash s \xrightarrow{R} t$ and $\Delta \vdash s \approx_\alpha s'$ then $\Delta \vdash s' \rightarrow t$ does *hold (not necessarily with the same rule).*
(2) In a uniform *rewrite system, peaks which are instances of trivial critical pairs* are *joinable.*

Proof

(1) By induction on the structure of s. If the reduction takes place at the root position then the rule applies to s' too because matching takes $\approx_\alpha$ into account. If the reduction is in a strict subterm of s (then s cannot be an atom or a moderated variable), we proceed by induction. The cases of a tuple and function application are trivial, as well as the case in which $s \equiv [a]u$ and $s' \equiv [a]u'$. The only interesting case is when $s \equiv [a]u$, $s' \equiv [b]u'$, and $\Delta \vdash u \rightarrow v$. Then we know that $\Delta \vdash (b\ a)\cdot u \approx_\alpha u'$ and $\Delta \vdash b\#u$, hence $\Delta \vdash u \approx_\alpha (a\ b)\cdot u'$ and $\Delta \vdash a\#(a\ b)\cdot u$ by Lemma 20. By induction, $\Delta \vdash (a\ b)\cdot u' \rightarrow v$, and by Theorem 50, $\Delta \vdash u' \rightarrow (a\ b)\cdot v$. Then $\Delta \vdash [b]u' \rightarrow [b](a\ b)\cdot v \approx_\alpha [a]v$ (because $\Delta \vdash b\#u$ implies $\Delta \vdash b\#v$ by Lemma 60).
(2) Suppose two rules $R_i = \nabla_i \vdash l_i \rightarrow r_i$ for $i = 1, 2$ have a critical pair

$$\Gamma \vdash (r_1\theta, L\theta[r_2\theta])$$

Then by Definition 53, $l_1 \equiv L[l_1']$, and (Γ, θ) is such that $\Gamma \vdash l_1'\theta \approx_\alpha l_2\theta$, and $\Gamma \vdash \nabla_1\theta, \nabla_2\theta$. Recall also that we call the critical pair trivial when $L = [\text{-}]$ and R_1, R_2 are copies of the same rule, or l_1' is a variable.

If R_1 and R_2 are identical, then their rewrites are identical and any peak created by these rules is trivially joinable.

If R_1 and R_2 differ and l_1' is a variable, then the only way we might not be able to apply R_1 in $L\theta[r_2\theta]$ or its instances, is if some freshness condition on l_1' in ∇_1 is unsatisfiable after R_2, which was satisfiable before R_2 (see the example above). For uniform rules, Lemma 60 guarantees that this cannot happen.

Therefore instances of a trivial critical pair are joinable.

*

Theorem 62 (Critical pair lemma) *If all non-trivial critical pairs of a uniform nominal rewrite system are joinable, then it is locally confluent.*

Proof Suppose $\Delta \vdash s \to t_1, t_2$ is a peak. Then:

(1) There exist $R_i = \nabla_i \vdash l_i \to r_i$, for $i = 1, 2$.
(2) s may be written as $C_i[s_i]$, and t as $C_i[t_i]$, for $i = 1, 2$.
(3) There exist solutions σ_i to $(\nabla_i \vdash l_i, r_i)) \ {}_?{\approx}\ (\Delta \vdash (s_i, t_i))$ for $i = 1, 2$. Hence $\Delta \vdash l_i\sigma_i \approx_\alpha s_i, \nabla_i\sigma_i$.

Now there are two possibilities:

(1) The distinguished context variable [-] occurs at distinct subtrees of s in C_1 and C_2. Local confluence holds by a standard diagrammatic argument taken from the first-order case (see for instance [2]). We need Theorems 24 and 61 to account for the use of $\approx_\alpha$.
(2) $C_2 \equiv C_1[D[\text{-}]]$ or $C_1 \equiv C_2[D[\text{-}]]$. We consider only the first possibility. Suppose that $C_1 \equiv [\text{-}]$, so that $C_2 \equiv D$ (the general case follows using Theorems 24 and 50).

There are now three possibilities:

(1) [-] replaces a variable X in s. This is an instance of a trivial critical pair. If the rules are uniform, joinability of instances of trivial critical pairs follows from the previous lemma.
(2) $D \equiv [\text{-}]$ and R_1 and R_2 are copies of the same rule. Then $t_1 \approx_\alpha t_2$ and the peak can be trivially joined.
(3) Otherwise, this is an instance of a non-trivial critical pair (see Definition 55). Non-trivial critical pairs are joinable by assumption, and using Theorem 50 we can join their instances.

*

Remark: As a first application of this result, we can deduce that the substitution rules in Example 43 are locally confluent: they are uniform (we will show this in next section), and the non-trivial critical pairs can be easily joined.

Note that if we consider also (Beta) then the system is not locally confluent. This does not contradict the previous theorem, because there is a critical pair between (Beta) and $(\sigma_{\mathtt{app}})$ which is not joinable. Of course, the system is locally confluent on ground terms (i.e. terms without variables): the critical pair between (Beta) and $(\sigma_{\mathtt{app}})$ is joinable if we replace the variables by ground terms. *

We will say that an NRS is **terminating** if all the rewrite sequences are finite. Using Newman's Lemma [34], we obtain the following confluence result.

Corollary 63 *(1) If an NRS is terminating, uniform, and non-trivial critical pairs are joinable, then it is confluent.*
(2) Under the same assumptions, normal forms are unique modulo $\approx_\alpha$.

7 Orthogonal systems

We now treat a standard confluence criterion in rewriting theory [16,31,33].

Definition 64 *A rule* $R \equiv \Delta \vdash l \to r$ *is* ***left-linear*** *when each variable occurring in* l *occurs only once.*

A uniform nominal rewrite system with only left-linear rules and no non-trivial critical pairs is ***orthogonal****.*

For example, $a\#X, b\#X \vdash fX \to (X, X)$ is left-linear but $\vdash (X, X) \to fX$ is not.

The subsystem defining substitution in Example 43 (i.e., the σ rules) is not orthogonal, because rule σ_ϵ generates non-trivial critical pairs. However, if we replace σ_ϵ in Example 43 by the rule

$$b[a \mapsto X] \to b$$

we obtain an orthogonal system (less efficient than the original one, since substitutions will be pushed all the way to the leaves of the terms even if the concerned variable does not occur in the term).

Theorem 65 *An orthogonal uniform nominal rewrite system is confluent.*

The proof occupies the rest of this section. Henceforth, we only consider uniform rewriting.

We define a **parallel reduction** relation $\Rightarrow$ as follows:

$$\frac{}{\Delta \vdash u \Rightarrow u}\ (\mathbf{refl}) \qquad \frac{\Delta \vdash (s_i \Rightarrow t_i)_{1\le i\le n}}{\Delta \vdash (s_1,\ldots,s_n) \Rightarrow (t_1,\ldots,t_n)}\ (\mathbf{tup})$$

$$\frac{\Delta \vdash (s_i \Rightarrow t_i)_{1\le i\le n} \quad \Delta \vdash (t_1,\ldots,t_n) \xrightarrow{R_\epsilon} t'}{\Delta \vdash (s_1,\ldots,s_n) \Rightarrow t'}\ (\mathbf{tup'})$$

$$\frac{\Delta \vdash s \Rightarrow t}{\Delta \vdash [a]s \Rightarrow [a]t}\ (\mathbf{abs}) \qquad \frac{\Delta \vdash s \Rightarrow t \quad \Delta \vdash [a]t \xrightarrow{R_\epsilon} t'}{\Delta \vdash [a]s \Rightarrow t'}\ (\mathbf{abs'})$$

$$\frac{\Delta \vdash s \Rightarrow t}{\Delta \vdash fs \Rightarrow ft}\ (\mathbf{fun}) \qquad \frac{\Delta \vdash s \Rightarrow t \quad \Delta \vdash ft \xrightarrow{R_\epsilon} t'}{\Delta \vdash fs \Rightarrow t'}\ (\mathbf{fun'})$$

$$\frac{\Delta \vdash a \xrightarrow{R_\epsilon} t'}{\Delta \vdash a \Rightarrow t'}\ (\mathbf{atom'}) \qquad \frac{\Delta \vdash \pi\cdot X \xrightarrow{R_\epsilon} t'}{\Delta \vdash \pi\cdot X \Rightarrow t'}\ (\mathbf{var'})$$

We have used some new notation: $\Delta \vdash s \xrightarrow{R_\epsilon} t$ means 's rewrites to t using R where we have matched the whole of s to the left-hand side of R'. For example if $R \equiv a \to a$ then $a \xrightarrow{R_\epsilon} a$ but not $(a,a) \xrightarrow{R_\epsilon} (a,a)$.

Lemma 66 *(1) If $\Delta \vdash s \Rightarrow t$ then $\Delta \vdash \pi\cdot s \Rightarrow \pi\cdot t$.*
(2) If $\Delta \vdash s \Rightarrow t$ then $\Delta \vdash C[s] \Rightarrow C[t]$.
(3) If $\Delta \vdash s \xrightarrow{R_\epsilon} t$ then $\Delta \vdash s \Rightarrow t$.
(4) If $\Delta \vdash s \to t$ then $\Delta \vdash s \Rightarrow t$.
(5) If $\Delta \vdash s \Rightarrow t$ then $\Delta \vdash s \to^* t$.
(6) As a corollary, $\Delta \vdash s \Rightarrow^* t$ if and only if $\Delta \vdash s \to^* t$.

Proof

(1) By induction on the derivation of $\Delta \vdash s \Rightarrow t$. We exploit the syntax-directed nature of the rules; we consider only four cases.
   (a) If $\Delta \vdash a \Rightarrow t'$ then (observing how this can have been derived), it must be that $\Delta \vdash a \xrightarrow{R_\epsilon} t'$. Then $\Delta \vdash \pi\cdot a \xrightarrow{R_\epsilon} \pi\cdot t'$ by Theorem 50, and $\Delta \vdash \pi\cdot a \Rightarrow \pi\cdot t'$ by (**atom'**).
   (b) If $\Delta \vdash \tau\cdot X \Rightarrow t'$ then it must be that $\Delta \vdash \tau\cdot X \xrightarrow{R_\epsilon} t'$. Then by Theorem 50 $\Delta \vdash \pi\cdot\tau\cdot X \xrightarrow{R_\epsilon} \pi\cdot t'$ and $\Delta \vdash \pi\cdot\tau\cdot X \Rightarrow \pi\cdot t'$ by (**var'**).
   (c) If $\Delta \vdash \pi\cdot s \Rightarrow \pi\cdot t$ then $\Delta \vdash [\pi\cdot a]\pi\cdot s \Rightarrow [\pi\cdot a]\pi\cdot t$ by (**abs**). We observe that $[\pi\cdot a]\pi\cdot s \equiv \pi\cdot[a]s$.
   (d) If $\Delta \vdash \pi\cdot s \Rightarrow \pi\cdot t$ and $\Delta \vdash [\pi\cdot a]\pi\cdot t \xrightarrow{R_\epsilon} \pi\cdot t'$, then $\Delta \vdash \pi\cdot[a]s \mapsto \pi\cdot t'$ by (**abs'**).
(2) Directly by induction on C.
(3) Directly using (**refl**) and (**tup'**), (**abs'**), (**fun'**), (**atom'**), and (**var'**).
(4) Using the previous two parts.
(5) We work by induction on the derivation of $\Delta \vdash s \Rightarrow t$. If the derivation concludes with:

(a) (**refl**) then the result is trivial, since $\twoheadrightarrow^*$ is reflexive.
(b) (**tuple**) then we rewrite sequentially in each element of the tuple.
(c) (**tuple′**) then we rewrite as in the last part, and then once at top level.
(d) (**abs**), (**abs′**), (**fun**), or (**fun′**), then we reason much as we did for (**tuple**) and (**tuple′**).
(e) (**atom′**) then we know $\Delta \vdash a \to t$ so $\Delta \vdash a \twoheadrightarrow^* t$. Similarly for (**var′**).

*

Lemma 67 *If the system is uniform and orthogonal then: if $\Delta \vdash s \Rightarrow t$ and $\Delta \vdash s \Rightarrow t'$, then there exists some t'' such that $\Delta \vdash t \Rightarrow t''$ and $\Delta \vdash t' \Rightarrow t''$. Hence $\Rightarrow$ is confluent.*

Proof By induction on the derivation of $\Delta \vdash s \Rightarrow t$.

We consider one case. Suppose the derivation ends in (**tup**). By the syntax-driven nature of deduction there are three possibilities for the last rule in the derivation of $\Delta \vdash s \Rightarrow t'$: (**tup**), (**tup′**), and (**refl**):

(1) If $\Delta \vdash s \Rightarrow t'$ has a derivation ending in (**tup**) then the inductive hypothesis for $\Delta \vdash s_i \Rightarrow t_i$ and $\Delta \vdash s_i \Rightarrow t'_i$ give us t''_i such that $\Delta \vdash t_i \Rightarrow t''_i$ and $\Delta \vdash t'_i \Rightarrow t''_i$. We use (**tup**) and are done.

(2) If $\Delta \vdash s \Rightarrow t'$ has a derivation ending in (**tup′**) using $R \equiv \nabla \vdash l \to r$, that is $\Delta \vdash s \Rightarrow (t'_1, \ldots, t'_n)$ and $\Delta \vdash (t'_1, \ldots, t'_n) \xrightarrow{R_\epsilon} t'$, then θ exists such that

$$\Delta \vdash \ \nabla\theta, \quad (t'_1, \ldots, t'_n) \approx_\alpha l\theta, \quad r\theta \approx_\alpha t'.$$

We now proceed as illustrated and explained below:

$$\begin{array}{ccc} (s_1, \ldots, s_n) & \Longrightarrow & (t'_1, \ldots, t'_n) \approx_\alpha \ l\theta \xrightarrow{R_\epsilon} t' \\ \Downarrow & & \Downarrow \\ (t_1, \ldots, t_n) & \Longrightarrow & (t''_1, \ldots, t''_n) \approx_\alpha \ l\theta' \xrightarrow{R_\epsilon} r\theta' \end{array}$$

We apply the inductive hypothesis to close $\Delta \vdash t_i, t'_i \Rightarrow t''_i$ using Lemma 66 ($\twoheadrightarrow^* = \Longrightarrow$) and Lemma 72 to deduce of t''_i all freshness assumptions deducible of t'_i.

Since rules are non-overlapping, the rewrite $(t'_1, \ldots, t'_n) \Rightarrow (t''_1, \ldots, t''_n)$ takes place in the substitution θ, that is, $\theta \Rightarrow \theta'$.

Since rules are left-linear R still applies: $\Delta \vdash (t''_1, \ldots, t''_n) \xrightarrow{R_\epsilon} r\theta'$ and therefore $\Delta \vdash (t_1, \ldots, t_n) \Rightarrow r\theta'$ by (**tup′**) for R (for some substitution θ'). Finally, we use Lemma 61 and orthogonality to close with a rewrite $t' \Rightarrow r\theta'$.

(3) If $\Delta \vdash s \Rightarrow t'$ then $t' \equiv s$ and the diamond can be trivially closed.

The other cases are similar. $*$

We now come back to our theorem:

Proof If the uniform rewrite system has only left-linear rules and only trivial critical pairs, then $\Rightarrow$ is confluent by Lemma 67. Since $\rightarrow^* \subseteq \Rightarrow^*$ and $\Rightarrow^* \subseteq \rightarrow^*$ by Lemma 66, $\rightarrow$ is confluent. $*$

8 Closed rewriting (or: 'efficiently computable' nominal rewriting)

Suppose $R \equiv \nabla \vdash l \rightarrow r$ contains atoms and is in a nominal rewrite system. By equivariance that system contains all infinitely many R^π for all renamings π of those atoms. Checking whether s matches R is polynomial [18], but checking whether s matches any R^π, for all possible π, is NP-complete [12]. For efficiency we are interested in conditions to make this problem polynomial, we consider this now.

Given a rule $R \equiv \nabla \vdash l \rightarrow r$ we shall write $R' \equiv \nabla' \vdash l' \rightarrow r'$ where the primed versions of ∇, l, and r, have atoms and variables renamed to be fresh — for R, and possibly also for other atoms occurring in a term-in-context $\Delta \vdash s$. We shall always explicitly say what R' is freshened for when this is not obvious.

For example, a freshened version of $(a\#X \vdash X \rightarrow X)$ with respect to itself and to the term-in-context $a'\#X \vdash a'$ is $(a''\#X' \vdash X' \rightarrow X')$, where $a'' \not\equiv a, a'$ and $X' \not\equiv X$.

We will write $A(R')\#V(R)$ to mean that all atoms occurring in R' are fresh for each of the variables occurring in R.

Definition 68 *(1) $R \equiv \nabla \vdash l \rightarrow r$ is **closed** when*

$$(\nabla' \vdash (l', r')) \ {}_?{\approx} \ (\nabla, A(R')\#V(R) \vdash (l, r))$$

has a solution σ.

Here $R' \equiv \nabla' \vdash (l', r')$ is freshened with respect to R.

(2) Given $R \equiv \nabla \vdash l \rightarrow r$ and $\Delta \vdash s$ a term-in-context write

$$\Delta \vdash s \stackrel{R}{\rightarrow}_c t \quad \textit{when} \quad \Delta, A(R')\#V(\Delta, s) \vdash s \stackrel{R'}{\rightarrow} t$$

and call this **closed rewriting**.

Here R' is freshened with respect to R, $\Delta \vdash s$, and t (in part 1 of Lemma 69 below we show it does not matter which particular freshened R' we choose).

In the next few paragraphs we give some examples and make some comments on this definition.

So for example if $R \equiv a\#X \vdash [a'][a]X \to [a']X$ then $A(R) = \{a, a'\}$ and $V(R) = \{X\}$ and $R' \equiv a''\#X' \vdash [a'''][a'']X' \to [a''']X'$. The condition for being closed unpacks to:

- There exists a σ such that $X\sigma \equiv X$ for all $X \in V(R)$ and:
- $\nabla, A(R')\#V(R) \vdash \nabla'\sigma$.
- $\nabla, A(R')\#V(R) \vdash l \approx_\alpha l'\sigma$.
- $\nabla, A(R')\#V(R) \vdash r \approx_\alpha r'\sigma$.

Note that $V(\Delta, s) = V(\Delta, s, t)$ because of conditions we put on rewrite rules that unknowns cannot just 'appear' on the right-hand side. We shall use these to simplify expressions denoting freshness contexts without comment.

There are two parts to this definition: closed rules, and closed rewriting. **It is possible to do (normal) rewriting with a closed rule, closed rewriting with a (normal) rule, or closed rewriting with a closed rule!** The intuition is that a closed rule generates the same rewrites with closed rewriting, as all (infinitely many) renamings of that rule generate with (normal) rewriting. The rest of this section formally develops these intuitions.

Note also that R' is freshened also with respect to t;

> "In closed rewriting, atoms explicitly mentioned in R are not allowed to interact with the atoms of the term being rewritten."

So for example if $R \equiv \emptyset \vdash a \to b$ then $a \overset{R}{\to} b$ but *not* $a \overset{R}{\to}_c b$, because R is freshened to $a' \to b'$ first.

Most of this subsection is about making this observation formal, in particular Part 2 of Lemma 69 and Theorem 71. Theorem 74 proves this restriction is computationally useful. Lemma 72 adds "and closed R are uniform", where uniformity is defined and discussed above.

For example, the rules in Example 43 are closed. A canonical example of a closed rule is $R \equiv a\#X \vdash X \to X$. Note that Z does not rewrite to Z with R (though $a\#Z \vdash Z \overset{R}{\to} Z$). The canonical example of a closed rewrite is $Z \overset{R}{\to}_c Z$. On the other hand, $a \to a$ is not a closed rule, neither are $fa \to b$, $fb \to b$ or $[a]X \to X$, but $a\#X \vdash [a]X \to X$ is closed.

If we think of closed rewriting as being such that the atoms in R are bound to that rule, the assumption $A(R')\#V(\Delta, s)$ adds "*and* for any subsequent instantiations of their unknowns". This is why the rewrite $Z \overset{R}{\to}_c Z$ occurs even though R demands to know that some atom a is fresh for X.

It is interesting to note that CRSs rules are closed by definition, in the sense that left and right-hand sides of rules cannot contain free variables (the equivalent of unabstracted atoms). But in the case of CRSs this is a structural fact, whereas here closure is defined as a logical condition. ERSs have a similar requirement, but it is expressed in terms of *admissible* substitutions. Note also that the notion of closed rewriting was generalised to Horn Clauses by Cheney and Urban [42].

The following three technical results about renaming atoms will shortly be useful:

Lemma 69 *(1) For $\Delta \vdash s$ and R, if*

$$\Delta, A(R')\#V(\Delta, s) \vdash s \xrightarrow{R'} t$$

for one freshening R' with respect to R, $\Delta \vdash s$, and t, then $\Delta, A(R'')\#V(\Delta, s) \vdash s \xrightarrow{R''} t$ for all possible freshenings R'' with respect to R, $\Delta \vdash s$, and t.

(2) For any π, $\Delta \vdash s \xrightarrow{R}_c t$ if and only if $\Delta \vdash s \xrightarrow{R^\pi}_c t$.

(3) R is closed if and only if R^π is closed.

Proof

(1) Nominal Rewriting is equivariant in atoms; if $\Gamma \vdash u \xrightarrow{S} v$ then $\Gamma^\kappa \vdash u^\kappa \xrightarrow{S^\kappa} v^\kappa$ for any κ. Nominal Rewriting is also equivariant in variable names (unknowns), so a similar result holds for them though we have not developed the notation to express it.

If the atoms and variables in R' are disjoint from Δ, s, and t (if the variables are disjoint from those in Δ and s they must be for those in t, by correctness conditions on rewriting)—then we can create a permutation κ for atoms and another for unknowns, renaming them any fresh way we like.

(2) The particular identity of the atoms in R is destroyed moving to R'. We might as well take R' fresh for R and also for R^π. The result is now easy to see using the previous result.

(3) The predicate 'R is closed' has only one argument: R. Nominal Rewriting is equivariant on atoms, so we can permute them in 'R is closed' to obtain 'R^π is closed', without changing the truth value. The reverse implication also holds since π is invertible.

*

Note that closed rewriting considers *some* freshened R' and that the associated notation $\Delta \vdash s \xrightarrow{R}_c t$ suggests the choice does not matter since we do not annotate the arrow $\rightarrow$ with R', only with R. Part 1 proves this suggestion is correct.

Now we look at some simple examples:

(1) If $R \equiv a\#X \vdash X \to X$, $R' \equiv a''\#X' \vdash X' \to X'$ and $R'' \equiv a'''\#X'' \vdash X'' \to X''$, then if $a\#X, a''\#X \vdash s \xrightarrow{R'} t$ then $a\#X, a'''\#X \vdash s \xrightarrow{R''} t$.
(2) If $R \equiv a\#X \vdash X \to X$ and $\pi = (a\ b)$ observe that $R' \equiv a'\#X' \vdash X' \to X'$ is a freshening of both R and R^π with respect to $\emptyset \vdash Z$. With a different term-in-context or π we might need a different choice of atoms and unknowns but there are infinitely many to choose from.

In what follows we may say "we assume R' is fresh for such-and-such extra terms-in-context" or "this is valid for any suitably fresh R'"; we may also use closure of R to justify closure of R^π, or closed rewriting with R to justify closed rewriting with R^π. We are using the lemma above.

Theorem 70 *R is closed if and only if for all $\Delta \vdash s$, if $\Delta \vdash s \xrightarrow{R} t$ then $\Delta \vdash s \xrightarrow{R}_c t$. ($R$ is closed if and only if rewriting implies closed rewriting.)*

Proof Assume that R is closed and that $\Delta \vdash s \xrightarrow{R} t$. For simplicity suppose that the rewrite step is at the root position (the result then follows by induction). So let θ solve $(\nabla \vdash (l, r)) \ {}_?{\approx}\ (\Delta \vdash (s, t))$. Recall σ exists solving $(\nabla' \vdash (l', r')) \ {}_?{\approx}\ (\nabla, A(R')\#V(R) \vdash (l, r)$, because R is closed. By syntactic calculations we see that $V(R\theta) \subseteq V(\Delta, s)$. We can use these facts and Lemma 22 to prove that $\sigma\theta$ solves $(\nabla' \vdash (l', r')) \ {}_?{\approx}\ (\Delta, A(R')\#V(\Delta, s) \vdash (s, t))$.

Conversely, assume rewriting with R implies closed rewriting with R. Note the trivial rewrite $\nabla \vdash l \xrightarrow{R} r$, at root position. Therefore by assumption this rewrite is also generated by a freshened R' in the context ∇ augmented with $A(R')\#V(R)$. From the syntactic similarity of R' to R it must be this rewrite is also generated using the root position, and by definition that means we obtain precisely the conditions for closure. $*$

$R \equiv a\#X \vdash X \to X$ is a counterexample to the assertion that closed rewriting implies rewriting for closed R. But the result holds for ground terms:

Theorem 71 *Suppose s is ground and R is closed. Then $s \xrightarrow{R}_c t$ if and only if there exists some π such that $s \xrightarrow{R^\pi} t$.*

Proof Since s has no variables (it is ground), a freshness context is irrelevant. Then, the definition of closed rewriting boils down to: $s \xrightarrow{R}_c t$ if and only if $s \xrightarrow{R'} t$. The left-to-right implication is thus trivial, we take π to be a freshening permutation κ generating R' in the definitions above, as discussed variable names do not matter.

Conversely suppose $s \xrightarrow{R^\pi} t$ for some π. R^π is closed by Lemma 69 and by the previous theorem we obtain $s \xrightarrow{R^\pi}_c t$ and so $s \xrightarrow{R}_c t$ again by Lemma 69. $*$

We can re-state this result as follows:

> If R is closed then R captures the rewrites of its equivariance renaming class on ground terms.

Lemma 72 *Let $R \equiv \nabla \vdash l \to r$ be a closed rule. Then R is uniform.*

Proof We must show that $\nabla, \langle a\#l\rangle_{nf} \vdash \langle a\#r\rangle_{nf}$, and we know by assumption that, for any freshening, $(\nabla' \vdash (l', r')) \ {}_{?}{\approx}\ (\nabla, A(R')\#V(R) \vdash (l, r))$ has a solution, write it σ. Unpacking definitions,

$$\nabla, A(R')\#V(R) \vdash \nabla'\sigma,\ l \approx_\alpha l'\sigma,\ r \approx_\alpha r'\sigma.$$

$\nabla, \langle a\#l\rangle_{nf}, A(R')\#V(R) \vdash a\#l'\sigma$ by Lemma 15 and part 1 of Lemma 23.

We can always take a freshening such that $a \notin A(l')$, and use the technical lemma which follows to deduce that $\nabla, \langle a\#l\rangle_{nf} \vdash a\#X'\sigma$ for each $X' \in V(l')$. By assumption $V(r') \subseteq V(l')$ and reversing our reasoning we obtain $\nabla, \langle a\#l\rangle_{nf} \vdash a\#r$ as required. $*$

Lemma 73 *(1) If $\Delta, a'\#X \vdash a\#s$ and $a' \notin A(s)$ then $\Delta \vdash a\#s$.*
(2) If $a \notin A(l')$ then $\Delta \vdash a\#l'\sigma$ if and only if $\Delta \vdash X'\sigma$ for every $X' \in V(l')$.

Proof Both parts are proved by appealing to the syntax-directed nature of the rules for $\#$. $*$

Theorem 74 *If a nominal rewrite system is provided as the equivariant closure of a finite set of closed rules, then*

(1) Rewriting equals closed rewriting on ground terms and rewriting is polynomial on ground terms.
(2) Closed rewriting is polynomial on all (possibly non-ground) terms.

Proof The very first part is a consequence of the previous theorem.

The algorithm to polynomially derive the closed rewrites of $\Delta \vdash s$ under R^π for *all* π is to derive just the closed rewrites of R. The choice of R' does not matter because of Lemma 69 part 1. $*$

The restriction to closed rules gives a powerful notion of rewriting: we showed in [21] that we can simulate CRSs using closed nominal rules. However, there are interesting systems (e.g. the π-calculus, see also Example 49) with non-closed (but uniform) rules. We come back to this point in the conclusions.

9 Sorts and Extended Contexts

9.1 Sorts

Sorts and types are a way of organising terms into useful classes ('represents a natural number' is the classic example). Sorts serve to organise terms into 'the right' families for term-formers to act on them. They are particularly useful for talking about the abstract syntax of programming languages. Types serve to organise the semantics of terms. As usual, it is important in computer science to distinguish between the *syntax* $1 + 2$, which is the pair 1 and 2, and its meaning, which is 3.

We now demonstrate how to impose a *sorting system* on terms. A type system is a fascinating subject (can atoms have type 'the natural numbers', and if so what does that mean?); we explore this subject in [20].

Definition 75 *A* **Sorted Nominal Signature** Σ *is:*

(1) A set of **sorts of atoms** *typically written* ν*.*
(2) A set $\mathcal{S}$ *of* **base data sorts** *typically written* s*. These are names for the domains under consideration, for example* integer*,* boolean*.*
(3) **Term sorts** *typically written* τ*, defined by the following grammar:*

$$\tau ::= \nu \mid s \mid \tau \times \ldots \times \tau \mid [\nu]\tau.$$

where $\tau_1 \times \ldots \times \tau_n$ *is called a product and* $[\nu]\tau$ *an abstraction sort.*
(4) A set of **term-formers** f *as before, to each of which is now associated an* **arity** $\tau_1 \to \tau_2$*.*

If τ_1 *is an empty product, we say that* f *is* ***0-ary*** *or* ***a constant*** *and we omit the arrow.*

Example 76 A *sorted* nominal signature for a fragment of ML has one sort of atoms: ν, one sort of data: exp, and term-formers with arities as follows:

$$\texttt{var} : \nu \to exp \qquad \texttt{app} : exp \times exp \to exp$$
$$\texttt{lam} : [\nu]exp \to exp \quad \texttt{let} : exp \times [\nu]exp \to exp$$
$$\texttt{letrec} : [\nu](([\nu]exp) \times exp) \to exp$$

This example, derived from [40], illustrates clearly how sorts indicate binding scope.

Partition unknowns into countably infinite sets of **variables of sort** τ for

each τ. Similarly partition atoms into countably infinite sets of **atoms of sort** ν. Even in the sorted context we may drop the sorting subscripts where they are obvious or we do not care, as a notational convenience; X_τ and $X_{\tau'}$ are still different term variables, for which we have overloaded the symbol X, and similarly for a_ν.

A **swapping** is a pair $(a\ b)$ of atoms *of the same sort*. Permutations π are lists of swappings as before.

Then sorts for terms may be deduced by the following syntax-directed deduction rules:

$$a_\nu : \nu \qquad \pi{\cdot}X_\tau : \tau \qquad \frac{t_1 : \tau_1 \ \cdots \ t_n : \tau_n}{(t_1, \ldots, t_n) : \tau_1 \times \ldots \times \tau_n}$$

$$\frac{t : \tau}{[a_\nu]t : [\nu]\tau} \qquad \frac{t : \tau_1}{(f_{\tau_1 \to \tau_2} t) : \tau_2.}$$

It is not hard to prove the following well-behavedness properties:

Lemma 77 *(1) If $t : \tau$ and π is a permutation then $\pi{\cdot}t : \tau$.*
(2) If $t : \tau$ and $s : \tau'$ then $t[X_{\tau'} \mapsto s] : \tau$.

Proof The first part is by induction on the structure of t. The base case is the observation that if $\pi'{\cdot}X : \tau$ then $\pi \circ \pi'{\cdot}X : \tau$, which is straight from the sorting rules.

The second part is by induction on the structure of t. The base case is $t \equiv \pi{\cdot}X_{\tau'}$. Then $t[X_{\tau'} \mapsto s] \equiv \pi{\cdot}s$. Since $s : \tau'$ by the first part, $\pi{\cdot}s : \tau'$, and we are done. $*$

9.2 Extending freshness contexts

We will now show that, thanks to the use of contexts, the framework of nominal rewriting can be easily adapted to express strategies of reduction. As an example, we will show how to define the system λ_{ca} of closed reduction for the λ-calculus [23]. λ_{ca}-terms are linear λ-terms with explicit constructs for substitutions, copying and erasing. Reduction on λ_{ca} is defined in [23] using a set of conditional rule schemes, shown in Table 1, where x, y, z denote variables, and t, u, v denote terms.

We can formally define λ_{ca} using a nominal rewriting system, where we add two new kinds of constraints: $\bullet t$ (read t **is closed**), with the intended meaning "$a\#t$ for every atom a", and $a \in t$ (read a **is unabstracted in** t), the negation of $a\#t$.

Table 1
λ_{ca}-reduction

Name	Reduction		Condition
Beta	$(\lambda x.t)v$	$\rightarrow_{\mathsf{ca}}\ t[v/x]$	$FV(v) = \emptyset$
Var	$x[v/x]$	$\rightarrow_{\mathsf{ca}}\ v$	
App1	$(tu)[v/x]$	$\rightarrow_{\mathsf{ca}}\ (t[v/x])u$	$x \in FV(t)$
App2	$(tu)[v/x]$	$\rightarrow_{\mathsf{ca}}\ t(u[v/x])$	$x \in FV(u)$
Lam	$(\lambda y.t)[v/x]$	$\rightarrow_{\mathsf{ca}}\ \lambda y.t[v/x]$	
Copy1	$(\delta_x^{y,z}.t)[v/x]$	$\rightarrow_{\mathsf{ca}}\ t[v/y][v/z]$	
Copy2	$(\delta_{x'}^{y,z}.t)[v/x]$	$\rightarrow_{\mathsf{ca}}\ \delta_{x'}^{y,z}.t[v/x]$	
Erase1	$(\epsilon_x.t)[v/x]$	$\rightarrow_{\mathsf{ca}}\ t$	
Erase2	$(\epsilon_{x'}.t)[v/x]$	$\rightarrow_{\mathsf{ca}}\ \epsilon_{x'}.t[v/x]$	

We extend the deduction and simplification rules from section 3 respectively with:

$$\frac{(\Delta \vdash a\#t)_{a\in S}}{\Delta \vdash \bullet t}\ (\bullet\mathbf{R})\quad A(t,\Delta) \subsetneq S$$

$$\bullet t, Pr \Longrightarrow \{a\#t\}_{a\in Pr,t}, a'\#t, Pr$$

Here S is any set of atoms strictly containing the atoms in t and Δ, and $a' \notin A(Pr,t)$. In effect we need $A(t,\Delta)$ *and* one fresh atom; if $\Delta \vdash a\#t$ for $a \in A(t,\Delta)$ a renaming argument gives $\Delta \vdash b\#t$ for all other $b \notin A(t,\Delta)$. This is reflected in the simplification rule, which is more algorithmic and chooses one fresh atom.

$\bullet t$ is intuitively $\forall a.\ a\#t$. The rule for closure $\bullet t$ is slightly different from the usual predicate logic rule for $\forall$ because atoms behave here as constants and not variables. With that in mind the definitions are quite natural.

We can extend the deductions with rules including

$$\frac{a \in t_i}{a \in (t_1,\ldots,t_n)}\qquad \frac{}{a \in a}$$

and similarly extend the simplification rules. We can extend contexts with these new constraints and use them in ∇s of rewrite rules $\nabla \vdash l \rightarrow r$ to control triggering.

A closed reduction strategy can be specified, this time as a finite nominal

rewrite system (we only show rules $Beta$, App_1 and App_2):

$$\begin{array}{ll} Beta & \bullet V \vdash (\lambda[x]T)V \rightarrow T[x \mapsto V] \\ App1 & x \in T \vdash (TU)[x \mapsto V] \rightarrow (T[x \mapsto V])U \\ App2 & x \in U \vdash (TU)[x \mapsto V] \rightarrow T(U[x \mapsto V]) \end{array}$$

A theory of nominal rewriting with (general) constraints will be the subject of future work.

10 Conclusions

The technical foundations of this work are derived from work on nominal logic [36] and nominal unification [40]. We use a nominal matching algorithm, which is easy to derive from the unification algorithm and which inherits its good properties (such as most general unifiers), in our definition of rewriting.

Our theory stays close to the first-order case, while still allowing binding. We achieve this by working with concrete syntax, but up to a notion of equality $\approx_\alpha$ which is not just structural identity. It is not α-equivalence either: $\approx_\alpha$ is actually *logical*, in the sense that $\Delta \vdash s \approx_\alpha s'$ is something that we deduce using assumptions in Δ. We pay the price that terms, rewrites, and equalities, happen in a freshness context Δ. In practice Δ is fixed and does not seem to behave perniciously.

Nominal rewriting is more expressive than first-order rewriting and standard higher-order formats. Capture-avoiding substitution is not a primitive notion, but it is easy to define with nominal rules (we can spare the effort of 'implementing' α-conversion using de Bruijn indices and all the machinery associated to typical explicit substitution systems). It is also possible to define a nominal rewriting formalism with a primitive notion of substitution of terms for atoms (capture-avoiding).

Many directions for future work are still open. For instance:

- We have given a sufficient condition for confluence (orthogonality, for uniform systems); weaker conditions (for example, weak orthogonality) should also be considered.
- We have given a critical pair lemma, but we have not studied Knuth-Bendix style completion procedures.
- There are sort systems for nominal terms, but no type system yet. We are working on a type system and semantics which provide an interpretation

for terms with variables (usual semantics for λ-calculus and higher-order rewriting only consider ground terms). If we use term rewriting as a model of computation, termination (or strong normalisation) is an important property. It would be possible to devise sufficient conditions for termination of nominal rewriting using type systems.

Acknowledgements: We thank James Cheney, Ian Mackie, Andy Pitts, Christian Urban, and Nobuko Yoshida for useful comments.

References

[1] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, October 1991.

[2] Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, Great Britain, 1998.

[3] F. Barbanera, M. Fernández, and H. Geuvers. Modularity of strong normalization in the algebraic-λ-cube. *Journal of Functional Programming*, 6:613–660, 1997.

[4] Franco Barbanera and Maribel Fernández. Intersection type assignment systems with higher-order algebraic rewriting. *Theoretical Computer Science*, 170:173–207, 1996.

[5] H. P. Barendregt. Pairing without conventional constraints. *Zeitschrift für mathematischen Logik und Grundlagen der Mathematik*, 20:289–306, 1974.

[6] H. P. Barendregt. *The Lambda Calculus: its Syntax and Semantics (revised ed.)*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 1984.

[7] J. Bergstra and J. W. Klop. Conditional rewrite rules: Confluence and terminatio n. *Journal of Computer and System Sciences*, 32(3):323–362, 1986.

[8] R. Bloo and K. Rose. Preservation of strong normalisation in named lambda calculi with explicit substitution and garbage collection, 1995.

[9] Eduardo Bonelli, Delia Kesner, and Alejandro Ríos. From higher-order to first-order rewriting. In *Proceedings of the 12th Int. Conf. Rewriting Techniques and Applications*, volume 2051 of *Lecture Notes in Computer Science*. Springer, 2001.

[10] Val Breazu-Tannen and Jean Gallier. Polymorphic rewriting conserves algebraic strong normalization. *Theoretical Computer Science*, 83(1), 1991.

[11] Val Breazu-Tannen and Jean Gallier. Polymorphic rewriting conserves algebraic confluence. *Information and Computation*, 82:3–28, 1992.

[12] James Cheney. The complexity of equivariant unification, 2004. Submitted.

[13] H. Cirstea and C. Kirchner. The Rewriting Calculus - Part I. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9:363–399, May 2001. Also available as Technical Report A01-R-203, LORIA, Nancy (France).

[14] H. Cirstea and C. Kirchner. The Rewriting Calculus - Part II. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9:401–434, May 2001. Also available as Technical Report A01-R-204, LORIA, Nancy (France).

[15] R. David and B. Guillaume. A λ-calculus with explicit weakening and explicit substitution. *Mathematical Structure in Computer Science*, 11(1):169–206, 2001.

[16] N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Formal Methods and Semantics*, volume B. North-Holland, 1989.

[17] Daniel J. Dougherty. Adding algebraic rewriting to the untyped lambda calculus. In *Proc. 4th Rewriting Techniques and Applications, LNCS 488*, Como, Italy, 1991.

[18] M. Fernández and C. Calves. 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.

[19] M. Fernández and M.J. Gabbay. Nominal rewriting with name generation: Abstraction vs. locality. In *Proceedings of the 7th ACM-SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'05), Lisbon, Portugal*. ACM Press, 2005.

[20] M. Fernández and M.J. Gabbay. Types for nominal rewriting, 2006. Submitted.

[21] M. Fernández, M.J. Gabbay, and I. Mackie. Nominal rewriting systems. In *Proceedings of the 6th ACM-SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'04), Verona, Italy*, 2004.

[22] M. Fernández and J.-P. Jouannaud. Modular termination of term rewriting systems revisited. In *Recent Trends in Data Type Specification. Proc. 10th. Workshop on Specification of Abstract Data Types (ADT'94)*, number 906 in LNCS, Santa Margherita, Italy, 1995.

[23] M. Fernández, I. Mackie, and F-R. Sinot. Closed reduction: Explicit substitutions without alpha-conversion. *Mathematical Structures in Computer Science*, 15(2), 2005.

[24] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax involving binders. In *14th Annual Symposium on Logic in Computer Science*, pages 214–224. IEEE Computer Society Press, Washington, 1999.

[25] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2001.

[26] Murdoch J. Gabbay. *A Theory of Inductive Definitions with alpha-Equivalence*. PhD thesis, Cambridge, UK, 2000.

[27] Makoto Hamana. Term rewriting with variable binding: An initial algebra approach. In *Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP2003)*. ACM Press, 2003.

[28] J.-P. Jouannaud and M. Okada. Executable higher-order algebraic specification languages. In *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 350–361. IEEE Computer Society Press, 1991.

[29] Z. Khasidashvili and V. van Oostrom. Context sensitive conditional reduction systems. *Proc. SEGRAGRA'95, Electronic Notes in Theoretical Computer Science*, 2, 1995.

[30] Zurab Khasidashvili. Expression reduction systems. In *Proceedings of I.Vekua Institute of Applied Mathematics*, volume 36, pages 200–220, Tbisili, 1990.

[31] J.-W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems, introduction and survey. *Theoretical Computer Science*, 121:279–308, 1993.

[32] Pierre Lescanne. From $\lambda\sigma$ to $\lambda\upsilon$ a journey through calculi of explicit substitutions. In *Proceedings of the 21st ACM Symposium on Principles of Programming Languages (POPL'94)*. ACM Press, 1994.

[33] Richard Mayr and Tobias Nipkow. Higher-order rewrite systems and their confluence. *Theoretical Computer Science*, 192:3–29, 1998.

[34] M.H.A. Newman. On theories with a combinatorial definition of equivalence. *Annals of Mathematics*, 43(2):223–243, 1942.

[35] Bruno Pagano. *Des calculs de substitution explicite et de leur application à la compilation des langages fonctionnels*. PhD thesis, Université de Paris 6, 1998.

[36] A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:1g5–193, 2003. A preliminary version appeared in the *Proceedings of the 4th International Symposium on Theoretical Aspects of Computer Software* (TACS 2001), LNCS 2215, Springer-Verlag, 2001, pp 219–242.).

[37] Femke van Raamsdonk. *Confluence and Normalisation for Higher-Order Rewriting*. PhD thesis, Free University of Amsterdam, 1996.

[38] 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), Uppsala, Sweden*, pages 263–274. ACM Press, August 2003.

[39] Masako Takahashi. λ-calculi with conditional rules. In J.F. Groote M. Bezem, editor, *Typed Lambda Calculi and Applications, International Conference TLCA'93*, volume 664 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[40] C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. In M. Baaz, editor, *Computer Science Logic and 8th Kurt Gödel Colloquium (CSL'03 & KGC), Vienna, Austria. Proccedings*, volume 2803 of *Lecture Notes in Computer Science*, pages 513–527. Springer-Verlag, Berlin, 2003.

[41] C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. *Theoretical Computer Science*, 323:473 – 497, 2004.

[42] Christian Urban and James Cheney. Avoiding equivariance in alpha-prolog. In *Proceedings of Typed Lambda Calculus and Applications, TLCA 2005*, pages 401–416, 2005.