Nominal Rewriting Systems

Maribel Fernández
Dept. of Computer Science
King's College London
Strand, London WC2R 2LS
UK
maribel@dcs.kcl.ac.uk

Murdoch J. Gabbay
LIX
École Polytechnique
91128 Palaiseau Cedex
France
gabbay@lix.polytechnique.fr

Ian Mackie
Dept. of Computer Science
King's College London
Strand, London WC2R 2LS
UK
ian@dcs.kcl.ac.uk

ABSTRACT

We present a generalisation of first-order rewriting which allows us to deal with terms involving binding operations in an elegant and practical way. We use a nominal approach to binding, in which bound entities are explicitly named (rather than using a nameless syntax such as de Bruijn indices), yet we get a rewriting formalism which respects α-conversion and can be directly implemented. This is achieved by adapting to the rewriting framework the powerful techniques developed by Pitts et al. in the FreshML project.

Nominal rewriting can be seen as higher-order rewriting with a first-order syntax and built-in α-conversion. We show that standard (first-order) rewriting is a particular case of nominal rewriting, and that very expressive higher-order systems such as Klop's Combinatory Reduction Systems can be easily defined as nominal rewriting systems. Finally we study confluence properties of nominal rewriting.

Categories and Subject Descriptors

F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*lambda calculus and related systems*

General Terms

Theory

Keywords

Binders, α-conversion, first and higher-order rewriting.

1. INTRODUCTION

Term rewriting systems (TRS) provide a general framework for specifying and reasoning about computation. They are defined as transformation rules working over trees labelled by variable and function symbols. This simple idea is very powerful: TRSs can be regarded as a universal programming language which can be used to express different programming paradigms (functional, logical, parallel, etc.), as an abstract model of computation, or as transformations that can be used for program optimisation and automated theorem proving.

Standard TRSs are first-order, but a lot of effort has been devoted in the past to develop systems manipulating higher-order terms. Higher-order terms are built with higher-order functions and variables which can be free or bound. Several examples of higher-order systems can be found in the literature: Combinatory Reduction Systems (CRS) [22], Higher-order Rewrite Systems (HRS) [25], Expression Reduction Systems (ERS) [21, 20], and the rewriting calculus [10, 11], combine first-order rewriting with a notion of bound variable. The λ-calculus can be seen as a higher-order rewrite system with one binder: λ-abstraction.

Systems that manipulate binders must also deal with α-conversion: first-order substitutions (i.e. replacements) may capture variables, so substitutions in a higher-order system use renamings. In all the higher-order formalisms that we mentioned above α-conversion is an implicit operation (in other words, terms are defined modulo renamings of bound variables), and substitution is also a meta-operation. Several notions of explicit substitutions and explicit α-conversion have been defined for the λ-calculus (e.g. [1, 24, 12]) and more generally for higher-order rewrite systems (e.g. [28, 6]) 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. The explicit substitution systems that use names for variables either restrict the rewriting mechanism to avoid cases in which α-conversion would arise, or do not address the problem of α-conversion.

In this paper we present a new formalism for rewriting with bound variables. In our systems variables are named (thus the title "nominal rewriting systems", abbreviated as NRS in the sequel), substitution is first-order, and we deal with α-conversion by using an auxiliary **freshness** relation between bound variables and terms, written as $a\#t$ (read "a is fresh for the term t"). We can see nominal terms as first-order terms with built-in α-conversion. In this sense, NRSs can be seen as an explicit substitution version of higher-order rewriting where α-conversion is already formalised (it does not need to be specified explicitly).

The use of the notion of freshness in a rewriting framework is the main novelty of this work. It is based on the work reported in [29, 15]. A similar freshness relation is used to define a metalanguage for functional programming in [30] and to study unification problems in [33].

PPDP'04, August 24–26, 2004, Verona, Italy.

Following [29, 15], we call **atoms** the names that can be bound and reserve the word **variable** for the identifiers that cannot be bound (known as variables and metavariables respectively in CRSs), and 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. For example, the β-reduction rule and the η-expansion rule of the λ-calculus are written as:

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

To summarise, the main contributions of this paper are:

1. The formulation of a notion of rewriting on nominal terms which behaves as first-order rewriting modulo α-conversion for bound atoms. In particular, substitution remains a first-order notion: we deal with α-conversion without introducing meta-substitutions and β-reductions in our metalanguage (in contrast with standard notions of higher-order rewriting, which rely on meta-substitutions and/or β-reductions in the substitution calculus). There is a price to pay for keeping first-order substitutions: in some cases we need to introduce freshness assumptions in terms and rewrite rules; these will be taken into account in the matching algorithm. We use **nominal matching** [33, 34] to rewrite terms. Selecting a nominal rewrite rule that matches a given term is an NP-complete problem in general [9]. However, by restricting to **closed** rules we can avoid the exponential cost: nominal matching is polynomial in this case. CRSs, ERSs, and HRSs with patterns impose similar restrictions.

2. A Critical Pair Lemma which ensures that nominal rewriting rules which do not introduce critical pairs are locally confluent. Similar results have been proved for other notions of higher-order rewriting (see for instance [22, 25]).

3. A translation from CRSs to NRSs which is sound and complete. This shows that nominal rewriting is as expressive as standard higher-order rewriting systems. Translations between CRSs, NRSs and ERSs have already been defined (see [31]), and there is also a translation of CRSs in the rewriting calculus [5].

Related Work.

Term rewriting systems and the λ-calculus provide two useful notions of rewriting, and both formalisms have been used as a basis for specification and programming languages. Barendregt [4] showed that the term rewriting rules defining surjective pairing cannot be encoded in the λ-calculus, which motivated the study of more general formalisms combining the power of term rewriting with the expressivity of the λ-calculus to define binding operators. Combinations of term rewriting and λ-calculus (both typed and untyped) have been studied for instance in [7, 8, 19, 3]. In all these algebraic λ-calculi the β-reduction rule of the λ-calculus is combined with a set of term rewriting rules using standard first-order matching, and for this reason λ-abstractions do not appear in patterns. In contrast, higher-order rewriting systems (e.g. CRSs [22], HRSs [25], ERSs [21, 20], HORSs [31], and the ρ-calculus [10, 11]) extend first-order rewriting to include binders using higher-order substitutions and higher-order matching (i.e. matching modulo β) and therefore binders are allowed in left-hand sides. Nominal Rewriting Systems are related to these since nominal rules may have binders in left-hand sides. However, nominal rewriting does not need higher-order matching. Instead, it relies on an extension of first-order matching that takes care of α-conversion without introducing β-reductions.

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. [28, 6]), 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-conversion (a design idea borrowed from FreshML). Also, from a (human) user point of view, it is easier to use systems with variable names. 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 to share many of its properties: we have efficient matching, a critical pair lemma, and we conjecture that other confluence and termination results can be transfered. More work needs to be done in this direction.

Nominal rewriting is also related to Hamana's Binding Term Rewriting Systems (BTRS) [17]. The main difference 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 role 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. We show in this paper that nominal rewrite systems in this subclass have the same power as CRSs (and therefore also the same power as HRSs and ERSs). More work needs to be done to compare the expressive power of NRSs and BTRSs.

Overview of the paper.

Section 2 recalls some concepts from first-order and higher-order rewriting. Section 3 presents nominal signatures, terms and rules. Nominal matching, and its use in nominal rewrite steps, are described in Section 4. Section 5 gives a critical pair lemma for nominal rewriting. In Section 6 we compare nominal rewriting with first and higher-order rewriting systems. We conclude the paper in Section 7.

2. BACKGROUND

In this section we recall two formalisms that are closely related to nominal rewriting: first-order rewriting and CRSs. We refer the reader to [2, 13, 22] for details and examples.

2.1 Term Rewriting Systems

A signature $\mathcal{F}$ is a finite set of function symbols with fixed arities, $\mathcal{X}$ denotes a denumerable set of variables, and

$T(\mathcal{F}, \mathcal{X})$ denotes the set of terms built up from $\mathcal{F}$ and $\mathcal{X}$. Terms are identified with finite labelled trees as usual, and positions are strings of positive integers. The subterm of t at position p is denoted by $t|_p$ and the result of replacing $t|_p$ with u at position p in t is denoted by $t[u]_p$. $V(t)$ denotes the set of variables occurring in t. We use Greek letters for substitutions and postfix notation for their application.

Given a signature $\mathcal{F}$, a **term rewriting system** (TRS) on $\mathcal{F}$ is a set of rewrite rules $R = \{l_i \to r_i\}_i$, where $l_i, r_i \in T(\mathcal{F}, \mathcal{X})$, $l_i \notin \mathcal{X}$ and $V(r_i) \subseteq V(l_i)$. A term t **rewrites** to a term u at position p with the rule $l \to r$ and the substitution σ, written $t \to_p^{l \to r} u$, or simply $t \to_R u$, if $t|_p = l\sigma$ and $u = t[r\sigma]_p$. We say that l **matches** $t|_p$ using σ.

We denote by $\to_R^+$ (resp. $\to_R^*$) the transitive (resp. transitive and reflexive) closure of the rewrite relation $\to_R$. The subindex R is omitted when it is clear from the context.

2.2 Combinatory Reduction Systems

A CRS [23, 22] is a pair consisting of an alphabet and a set of rewrite rules. The alphabet consists of: variables $a, b, c, \ldots$; metavariables with fixed arities, written as Z_i^k where k is the arity of Z_i^k (k is omitted when there is no ambiguity); function symbols $f, g, \ldots$ with fixed arities; and an abstraction operator $[\cdot]\cdot$.

In CRSs a distinction is made between **metaterms** and **terms**. Metaterms are the expressions built from the symbols in the alphabet, in the usual way. Terms are metaterms that do not contain metavariables. Variables that are in the scope of the abstraction operator are **bound**, and **free** otherwise. A (meta)term is closed if every variable occurrence is bound. CRSs adopt the usual naming conventions (also known as Barendregt's variable conventions): in particular, all bound variables are chosen to be different from the free variables.

A rewrite rule is a pair of metaterms, written $l \to r$, where l, r are closed, l has the form $f(s_1, \ldots, s_n)$, the metavariables that occur in r occur also in l, and the metavariables Z_i^k that occur in l occur only in the form $Z_i^k(a_1, \ldots, a_k)$, where $a_1, \ldots, a_k$ are pairwise distinct variables.

Example 2.1. *The β-reduction rule for the λ-calculus is written in the CRSs syntax as:*

$$\mathsf{app}(\mathsf{lam}([a]Z(a)), Z') \to Z(Z')$$

where the binary function symbol app *represents application and the unary function symbol* lam *represents λ-abstraction. Z is a unary metavariable, and Z' is 0-ary.*

The reduction relation is defined on terms. To extract from rules the actual rewrite relation, each metavariable is replaced by a special kind of λ-term, and in the obtained term all β-redexes and the residuals of these β-redexes are reduced (i.e. a development is performed). Formally, the rewrite relation is defined using **substitutes** and **valuations**. An n-ary substitute is an expression of the form $\lambda x_1 \ldots x_n.t$, where t is a term and $x_1, \ldots, x_n$ are different variables. An n-ary substitute can be applied to a n-tuple $s_1, \ldots, s_n$ of terms, and the result is the term t where $x_1, \ldots, x_n$ are simultaneously replaced by $s_1, \ldots, s_n$. A valuation σ is a map that assigns an n-ary substitute to each n-ary metavariable. It is extended to a mapping from metaterms to terms: given a valuation σ and a metaterm t, first we replace in t all metavariables by their images in σ and then we perform the developments of the β-redexes created.

A context is a term with an occurrence of a special symbol [] called hole. A rewrite step is now defined in the usual way: if $l \to r$ is a rewrite rule, σ a valuation and $C[\,]$ a context, then $C[l\sigma] \to C[r\sigma]$.

Example 2.2. *The following is a rewrite step using the β-rule given in Example 2.1:*

$$\mathsf{app}(\mathsf{lam}([a]f(a,a)), t) \to_\beta f(t,t)$$

To generate it we use the valuation σ that maps Z to $\lambda b.f(b,b)$ and Z' to the term t.

3. NOMINAL REWRITE SYSTEMS

3.1 Sorts and Signatures

A **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 **function symbols** typically written f, to each of which is associated an **arity** $\tau_1 \to \tau_2$. If τ_1 is an empty product we say that f is 0-ary (i.e. a constant) and omit the arrow as usual.

Example 3.1. *A nominal signature for a fragment of ML has one sort of atoms ν, one sort of data exp, and function symbols with arities as follows:*

$$\begin{array}{ll} \mathtt{var} : \nu \to exp & \mathtt{app} : exp \times exp \to exp \\ \mathtt{lam} : [\nu]exp \to exp & \mathtt{let} : exp \times [\nu]exp \to exp \\ \multicolumn{2}{c}{\mathtt{letrec} : [\nu](([\nu]exp) \times exp) \to exp} \end{array}$$

In the next section we define terms and give examples from this signature. The first three constructors alone are a signature for the untyped λ-calculus; in the course of this paper we shall tend to simplify the examples to that subsignature. The example above, derived from [33], is useful because it illustrates clearly how sorts indicate binding scope. The actual mechanics of binding is handled by a nontrivial decidable notion of equality on terms $\Delta \vdash s \approx_\alpha t$, defined below.

3.2 Terms, Swappings, and Contexts

We fix some signature Σ. For each term sort τ, we fix a countably infinite set $\mathcal{X}_\tau$ of **term variables** X, Y, Z of that sort. They will represent meta-level unknowns in the rewrite system.

$$\mathcal{X} \stackrel{\text{def}}{=} \bigcup_\tau \mathcal{X}_\tau$$

For each sort of atoms $\nu \in \Sigma$ fix a distinct countably infinite set $\mathcal{A}_\tau$ of **atoms** $a, b, c, f, g, h, \ldots$ of that sort.

$$\mathcal{A} \stackrel{\text{def}}{=} \bigcup_\tau \mathcal{A}_\tau$$

Variables are sorted and we may include explicit information in a subscript, for example X_τ; X_τ and $X_{\tau'}$ are different term variables, for which we have overloaded the symbol X. Similarly for a_ν.

A **swapping** is a pair of atoms of the same sort, which we write $(a\,b)$. **Permutations** π are generated by the grammar $\pi ::= \mathbf{Id} \mid (a\ b) \cdot \pi$. We call **Id** the **identity permutation**.

Swappings have an action on atoms $(a\ b)(n)$ given by

$$(a\,b)(a) \stackrel{\text{def}}{=} b \qquad (a\,b)(b) \stackrel{\text{def}}{=} a \quad \text{and} \quad (a\,b)(c) \stackrel{\text{def}}{=} c\,(c \neq a, b). \tag{1}$$

This extends to an action by permutations $\pi(n)$. It is also useful to define the **difference set**

$$ds(\pi, \pi') \stackrel{\text{def}}{=} \{n \mid \pi(n) \neq \pi'(n)\}. \tag{2}$$

For example, $ds((a\ b), \mathbf{Id}) = \{a, b\}$.

Nominal Terms (we will generally write just 'terms') are generated by the following grammar:

$$\begin{aligned} t ::= \; & a_\nu \mid (\pi \cdot X)_\tau \mid (t_{1\tau_1}, \ldots, t_{n\tau_n})_{\tau_1 \times \ldots \times \tau_n} \mid \\ & ([a_\nu]t_\tau)_{[\nu]\tau} \mid (f_{\tau_1 \to \tau_2} t_{\tau_1})_{\tau_2} \end{aligned}$$

and called resp. atoms, moderated variables, tuples, abstractions and function applications. **Ground terms** are terms without variables. In future we shall usually omit sorting annotations, outermost brackets for abstractions and applications, and brackets of empty tuples. In an abstraction $[a]t$, we will say that all the occurrences of a are **abstracted** (or bound); unabstracted occurrences are called **free**. However, we do not work modulo α-conversion of bound atoms at the syntactic level. In other words, syntactic equality, which we write $\equiv$, is not defined modulo α-equivalence: $[a]a$ and $[b]b$ are *not* equal syntax trees.

We write $T(\Sigma, \mathcal{A}, \mathcal{X})$ for the set of terms over a signature Σ.

For Σ as in Example 3.1, we shall sugar the syntax of terms to standard notation. We write a for $\mathtt{var}(a)$, tt' for $\mathtt{app}(t, t')$, $\lambda[a]t$ for $\mathtt{lam}([a]t)$, $\mathtt{let}\ a = t\ \mathtt{in}\ t'$ for $\mathtt{let}(t, [a]t')$, and $\mathtt{letrec}\ f\ a = t\ \mathtt{in}\ t'$ for $\mathtt{letrec}[f]([a]t, t')$. For example, a, $(\lambda[a]aa)(\lambda[a]aa)$, and $\mathtt{letrec}\ f\ a = a\ \mathtt{in}\ fb$ are terms. Note that the sorts tell us that f is abstracted in t and t' and a in t in the expression $\mathtt{letrec}\ f\ a = t\ \mathtt{in}\ t'$.

We call $\pi \cdot X$ a **moderated variable**, $(a\ b) \cdot X$ is the canonical example. We may write X for $\mathbf{Id}{\cdot}X$. The intuition of $(a\ b) \cdot X$ is "swap a and b in the syntax of X, when it is instantiated". We shall see why we need this when we define α-equivalence $\approx_\alpha$ below. Unification and matching instantiate X, so later we will write '$(a\ b) \cdot t$' for t a non-variable term. This is sugar, and we define it now:

$$\begin{gathered} (a\ b) \cdot n = (a\ b)(n) \quad (a\ b) \cdot ft = f(a\ b) \cdot t \\ (a\ b) \cdot (t_1, \ldots, t_n) = ((a\ b) \cdot t_1, \ldots, (a\ b) \cdot t_n) \\ (a\ b) \cdot [n]t = [(a\ b)(n)](a\ b)t \end{gathered} \tag{3}$$

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

The meaning of $\pi \cdot t$ is given in the natural way from the component swappings in π. We write π^{-1} for the permutation obtained by reversing the swappings in π. For example if $\pi = (a\ b)(b\ c)$ then $\pi^{-1}(a) = c$.

An **apartness condition** is a pair $(a \in \mathcal{A}, X \in \mathcal{X})$. We write it $a\#X$. **Apartness contexts** $\Delta, \nabla, \Gamma, \ldots$ are finite sets of apartness conditions.

Rewrite rules use apartness conditions to avoid accidental name capture. Intuitively, $a\#X$ means "the atom a does not occur unabstracted in X, when it is instantiated".

We define a notion of entailment $\Delta \vdash a\#s$ inductively as follows:

$$\begin{gathered} \frac{a\#s_1 \ \cdots \ a\#s_n}{a\#(s_1, \ldots, s_n)} \qquad \frac{a\#s}{a\#fs} \qquad \frac{a\#s}{a\#[b]s} \\ \frac{}{a\#b} \qquad \frac{}{a\#[a]s} \qquad \frac{\pi^{-1}(a)\#X}{a\#\pi \cdot X} \end{gathered} \tag{4}$$

We write $\Delta \vdash a\#s$ when a proof of $a\#s$ exists using elements of Δ as assumptions.

Write $\langle a\#s \rangle_{\#sol}$ for the least apartness context such that $\langle a\#s \rangle_{\#sol} \vdash a\#s$. The deduction rules in (4) are rigidly structural, and always make terms smaller, so $\langle a\#s \rangle_{\#sol}$ is well-defined, unique, and always exists. For example,

$$\begin{aligned} \langle a\#(X, [a]Y) \rangle_{\#sol} &= \{a\#X\} \\ \langle a\#((a\ b) \cdot X, (b\ c) \cdot Y) \rangle_{\#sol} &= \{b\#X, a\#Y\}. \end{aligned}$$

We now define $\approx_\alpha$, a notion of α-equivalence-in-context. Deductions are made according to the following inductive rules:

$$\begin{gathered} \frac{s_1 \approx_\alpha t_1 \ \cdots \ s_n \approx_\alpha t_n}{(s_1, \ldots, s_n) \approx_\alpha (t_1, \ldots, t_n)} \quad \frac{s \approx_\alpha t}{fs \approx_\alpha ft} \quad \frac{}{a \approx_\alpha a} \quad \frac{t \approx_\alpha t'}{t' \approx_\alpha t} \\ \frac{s \approx_\alpha t}{[a]s \approx_\alpha [a]t} \qquad \frac{a\#t \qquad s \approx_\alpha (a\ b) \cdot t}{[a]s \approx_\alpha [b]t} \qquad \frac{ds(\pi, \pi')\#X}{\pi \cdot X \approx_\alpha \pi' \cdot X} \end{gathered} \tag{5}$$

Here and elsewhere we write $a, b, c, \ldots \#X$ to indicate a set $a\#X, b\#X, \ldots$. For example, we can deduce $(a\ b) \cdot X \approx_\alpha X$ from the assumptions $a\#X$ and $b\#X$.

Write $\Delta \vdash s \approx_\alpha t$ when $s \approx_\alpha t$ is derivable using the assumptions Δ. The canonical examples are

$$\begin{aligned} a, b\#X &\vdash \quad (a\ b) \cdot X \quad \approx_\alpha \quad X \\ b\#X &\vdash \quad \lambda[a]X \quad \approx_\alpha \quad \lambda[b](b\ a) \cdot X \end{aligned}$$

An alternative characterisation of $\approx_\alpha$ is the least congruence containing the first axiom above, or a little more formally, such that if $a, b\#s$ then $s \approx_\alpha (a\ b) \cdot s$.

In future we may write $\Delta \vdash a\#t$ and $\Delta \vdash t \approx_\alpha t'$ when Δ is a collection of assumptions not necessarily of the form $a\#X$, i.e. when Δ is not necessarily a pure apartness context as defined above. The meaning is still that the conclusion can be deduced from the assumptions.

All the definitions and theorems in this paper—matching and unification, confluence, the Critical Pair Theorem 5.2—work up to this notion of α-equivalence. This differs from just quotienting terms by α-equivalence in a more traditional sense, because terms exist for us in an apartness context which allows us to equate terms such as $(a\ b) \cdot X$ and X, given assumptions $a, b\#X$ about a and b not occurring free in X. Our use of swappings rather than substitutions also contributes to making this work; we cannot deduce that $X[a/b]$ and X are α-equivalent if a and b are not free in X ($X[a/b]$ meaning, without giving a formal definition, "replace b by a throughout X, when it is instantiated"), e.g. instantiate X to $\lambda[a]\lambda[b]ab$.

Rewriting, when we define it, operates on a term-in-context: a pair (Δ, s) which we write $\Delta \vdash s$. Then $\Delta \vdash s$ rewrites to $\Delta \vdash t$—an apartness context is fixed. Since in a particular rewriting path the context is fixed, a given context defines a particular rewrite system $\Delta \vdash - \to -$ and a notion of equality $\Delta \vdash - \approx_\alpha -$.

3.3 Substitutions and Positions in Terms

Substitutions are generated by the following grammar:

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

We need to apply a substitution to a rewrite rule $\nabla \vdash l \to r$, in order to instantiate it to match a term in an apartness context. We develop the theory later, but it motivates us to give a substitution action on both terms *and* apartness contexts.

We write substitutions postfix: $l\sigma$, $\nabla\sigma$. Mostly we consider only the case $\sigma = \mathbf{Id}$ which we call the **identity substitution**, or $\sigma = [X \mapsto s]$. The general case is obtained by repeated applications. We shall assume without comment that **Id** has the trivial action whatever it is applied to.

Positions p are strings of natural numbers denoting paths in the syntax tree of a term. We write ϵ for the empty string and denote $p.q$ the concatenation of p and q (we assume the reader is familiar with this notation, see Section 2). Then ϵ is the root position, 1 indicates going under a function symbol or an abstraction $[a]s$; i refers to ith position in $(s_1, \ldots, s_n)$; permutations are ignored.

We denote the subterm of s at position p by $s|_p$. We omit a formal definition. For example

$$(X, [a]([b](b, c), X))|_{\epsilon.2.1.1.1} \equiv (b, c) \qquad (a\ b) \cdot X|_\epsilon \equiv X$$

We denote the term obtained by replacing t by t' at position p in s by $s[t']_p$. For example for s and p the term and position in the first example above, $s[a]_p \equiv (X, [a]([b]a, X))$, and in the second example $(a\ b) \cdot X[a]_\epsilon \equiv b$.

Finally we write $s[X \mapsto t]$ for the term obtained by simultaneously substituting every X in s by t, using the notion of replacement just developed specialised to the position of X in s, for every X. For example $s[X \mapsto a] = (a, [a]([b](b, c), a))$. Note there is no capture avoidance.

Substitutions apply also to contexts: we define

$$a\#X[Y \mapsto s] \stackrel{\text{def}}{=} \{a\#X\} \quad a\#X[X \mapsto s] \stackrel{\text{def}}{=} \langle a\#s \rangle_{\#sol}$$

$$\Delta[X \mapsto s] \stackrel{\text{def}}{=} \bigcup_{C \in \Delta} C[X \mapsto s].$$

3.4 Rewrite Rules and Systems

Write $V(s)$ for the variables $X \in \mathcal{X}$ occurring in the term s, $A(s)$ for the atoms mentioned in s. Similarly write $V(\nabla)$ for the variables in an apartness context.

A **nominal rewrite rule** over Σ is a tuple (∇, l, r), we write it $\nabla \vdash l \to r$, such that $V(r) \cup V(\nabla) \subseteq V(l)$. We may write $l \to r$ for $\emptyset \vdash l \to r$.

EXAMPLE 3.2.

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

$$(\nabla \vdash l \to r)[X \mapsto s] \stackrel{\text{def}}{=} \nabla[X \mapsto s] \vdash l[X \mapsto s] \to r[X \mapsto s]. \tag{6}$$

Though we shall never write the substitution in such detail, this is how we shall 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, making $a\#X \vdash X \to \lambda[a](Xa)$ and $b\#X \vdash X \to \lambda[b](Xb)$ 'morally' the same. It will be useful to have a notation for permuting atoms in the syntax of a rule: for a permutation π write $R^\pi = \nabla^\pi \vdash l^\pi \to r^\pi$ for the rule obtained by replacing every atom a in its syntax with $\pi(a)$. We formalise our morality as follows: say a set of rewrite rules $\mathcal{S}$ is **equivariant** when if $R \in \mathcal{S}$ then $R^\pi \in \mathcal{S}$ for all π.

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.

EXAMPLE 3.3. *To give a small-step evaluation relation for the signature Σ of our fragment of ML (see Example 3.1) we must extend it with a constructor for explicit substitutions* $\mathtt{sub} : ([\nu]exp) \times exp \to exp$ *which we sugar to $t[a \mapsto t']$. The rewrite rules are:*

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

These define a system of explicit substitutions for the λ-calculus with names, together with the following rules:

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

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 constructor abstraction $[a]t$.
3. The apartness 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.

$$
\begin{array}{rcl}
(l_1,\dots,l_n) \;{}_?{\approx}\; (s_1,\dots,s_n), P & \Longrightarrow & l_1 \;{}_?{\approx}\; s_1,\dots, \\
 & & l_n \;{}_?{\approx}\; s_n, P \\
fl \;{}_?{\approx}\; fs, P & \Longrightarrow & l \;{}_?{\approx}\; s, P \\
[a]l \;{}_?{\approx}\; [a]s, P & \Longrightarrow & l \;{}_?{\approx}\; s, P \\
[b]l \;{}_?{\approx}\; [a]s, P & \Longrightarrow & (a\ b)\cdot l \;{}_?{\approx}\; s, a\#l, P \\
a \;{}_?{\approx}\; a, P & \Longrightarrow & P \\
\pi\cdot X \;{}_?{\approx}\; \pi'\cdot X, P & \Longrightarrow & ds(\pi,\pi')\#X, P \\
a\#s, P & \Longrightarrow & \langle a\#s\rangle_{\#sol}, P \\
 & & (s \not\equiv X) \\
(\text{Matching}) \quad \pi\cdot Y \;{}_?{\approx}\; s, P & \stackrel{Y\mapsto\pi^{-1}\cdot s}{\Longrightarrow} & P[Y\mapsto\pi^{-1}\cdot s] \\
 & & (Y \notin s) \\
(\text{Unification}) \quad l \;{}_?{\approx}_?\; \pi\cdot X, P & \stackrel{X\mapsto\pi^{-1}\cdot l}{\Longrightarrow} & P[X\mapsto\pi^{-1}\cdot l] \\
 & & (X \notin l)
\end{array}
$$

Figure 1: Matching and Unification Rules

4. MATCHING AND REWRITING

4.1 Elementary Rewriting

A **matching problem** P is a set of **equality problems** $l \;{}_?{\approx}\; s$ and **apartness problems** $a\#s$. Problems are transformed to other (simpler) problems according to a labelled transition system as shown in Figure 1.

Here we find it convenient to make commas indicate *disjoint* set union on the left, and set union on the right. We also omit set brackets. $P[Y\mapsto s]$ denotes the pointwise application in the obvious reasonable way. 'l' suggests a rule and 's' the term against which we are trying to match the rule. In the rule (Matching) $Y \notin s$ means "Y does not appear in the syntax of s". Similarly for $X \notin l$.

Recall that $\equiv$ denotes structural identity on syntax. The side-condition $s \not\equiv X$ is just to ensure termination, since $\langle a\#X\rangle_{\#sol} = \{a\#X\}$.

If we include (Matching) and not (Unification) in a problem $\{l \;{}_?{\approx}\; s\}$ we obtain a matching algorithm, in a sense we discuss below, instantiating variables in l to make it equal in an appropriate sense to s. If we also include (Unification) we obtain a unification algorithm instantiating variables in l and s to make them equal. In that case we write $\{l \;{}_?{\approx}_?\; s\}$.

A transition is either trivially labelled $P \Longrightarrow P'$, or labelled with a substitution $P \stackrel{Y\mapsto s}{\Longrightarrow} P'$. It is easy to prove that the transition system terminates in finite time. Write $\theta_1,\dots,\theta_n$ for the nontrivial labels generated in some path to a terminal state P', and θ for their composition in that order (recall we write substitutions postfix).

Call a problem **terminal** if all its elements are of the form $a\#X$. Write terminal problems Γ. If P reduces to Γ generating θ as described in the last paragraph then say (Γ,θ) **solves** (or **is a solution to**) the problem P.

Note in passing that by construction Γ mentions none of the variable symbols in the domain of θ and that $\Gamma\theta = \Gamma$.

It is easy to verify that $\Delta \vdash l \approx_\alpha s$ precisely when $(\Gamma, \mathbf{Id})$ solves $l \;{}_?{\approx}\; s$ (or $l \;{}_?{\approx}_?\; s$) for some $\Gamma \subseteq \Delta$. Thus matching and unification do as promised work 'up to' α-equivalence in the sense of $\approx_\alpha$.

The matching and unification algorithm defined in Figure 1 was obtained by adapting the rules in [33, Fig.3] to our syntax. We assume without comment results from [33] about decidability and existence of unique most general unifiers in suitable senses (essentially, up to $\approx_\alpha$).

Say (Γ,θ) **solves** $s \;{}_?{\approx}\; t$ when it solves $\{s \;{}_?{\approx}\; t\}$.

Let Δ be an apartness context and s a term. Recall that the pair (Δ, s) is a **term-in-context** and that we write it $\Delta \vdash s$. A rewrite system $\mathcal{R}$ and an apartness context Δ determines a transition system on terms which we write $\Delta \vdash s \to t$ leaving $\mathcal{R}$ implicit. We define transitions after some useful terminology.

Define a **matching problem between terms-in-context** to be a pair of terms-in-context and write them $(\nabla \vdash l) \;{}_?{\approx}\; (\Delta \vdash s)$. A **solution** is a substitution θ such that

1. $\theta X \equiv X$ for X in $V(\Delta \vdash s)$.
2. $\Delta \vdash l\theta \approx_\alpha s$.
3. $\Delta \vdash \nabla\theta$.

It is easy to show that if a solution exists then a most general one is the θ from (θ,Γ) solving $l \;{}_?{\approx}\; s$.

Given a rewrite rule $R = \nabla \vdash l \to r$ we say that in a context Δ, s **rewrites with** R **to** t, 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. There exists a position p in s and a solution θ to $(\nabla \vdash l) \;{}_?{\approx}\; (\Delta \vdash s|_p)$.
3. $\Delta \vdash s[r\theta]_p \approx_\alpha t$.

We develop a slightly more involved notion of rewriting in the next subsection, using this one, so we may also call this **elementary rewriting**.

Finally, given a nominal rewrite system $\mathcal{R}$ say that in a context Δ s **rewrites to** t 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.

Lemma 4.1. *If $\Delta \vdash t \approx_\alpha s|_p$ then $\Delta \vdash s[t]_p \approx_\alpha s$.*

This technical lemma is very natural and is important for Theorem 5.2:

Lemma 4.2. *If $\Delta \vdash t \approx_\alpha t'$ and if p is a position in s, then $\Delta \vdash s[t]_p \approx_\alpha s[t']_p$.*

For example $\emptyset \vdash [a]a \approx_\alpha [b]b$ and $\emptyset \vdash [a][a]a \approx_\alpha [a][b]b$.

Note that if $s \approx_\alpha s'$ and $t \approx_\alpha t'$ it is not necessarily the case that $s[t]_p \approx_\alpha s'[t']_p$. For example, $[a]a \approx_\alpha [b]b$ and $a \approx_\alpha a$ but $[a]a \not\approx_\alpha [b]a$.

Corollary 4.3. *The latter two conditions defining $\Delta \vdash s \xrightarrow{R} t$ can be expressed succinctly as $(\nabla \vdash (s[l]_p, s[r]_p)) \;{}_?{\approx}\; (\Delta \vdash (s,t))$.*

Proof. Direct from the previous two lemmas. □

We now give some examples of rewrite steps:

Example 4.4. *1. It is easy to show that*

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

in four steps using the rules (Beta) *and* (σ_{var}) *of Example 3.3 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, as shown in Example 2.2,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 TRSs we don't need to make explicit the α-conversions. For instance, rule ($\sigma_{\mathtt{lam}}$) pushes a substitution under a λ without capture, as the following rewrite step shows:

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

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 all other 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. $X_\tau \to (a\ b)\cdot X_\tau$ allows us to arbitrarily swap a and b in any term of sort τ. Given equivariance of the set of rules, we can rename all atoms.*

4.2 Closed Rewriting and Efficiency

Suppose $R \equiv \nabla \vdash l \to r$ 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 and we have given the algorithm, but checking whether s matches R^π for some π is NP-complete [9]. For efficiency we are interested in conditions to make this problem polynomial, we consider this now.

Given a rule $R = \nabla \vdash l \to r$ we shall write $R' \equiv \nabla' \vdash l' \to r'$ 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 ($a\#X \vdash X \to X$) freshened with respect to itself and to the term-in-context $a'\#X \vdash a'$ is ($a''\#X' \vdash X' \to X'$), where $a'' \not\equiv a, a'$ and $X' \not\equiv X$.

We can now define:

1. $R \equiv \nabla \vdash l \to r$ is **closed** when $(\nabla' \vdash (l', r'))\ {}_?{\approx}\ (\nabla, A(R')\#V(R) \vdash (l, r))$ has a solution σ. Here R' is freshened with respect to R.

2. Given $R \equiv \nabla \vdash l \to r$ and $\Delta \vdash s$ a term-in-context write $\Delta \vdash s \stackrel{R}{\to}_c t$ when $\Delta, A(R')\#V(\Delta, s) \vdash s \stackrel{R'}{\to} t$ and call this **closed rewriting**. Here R' is freshened with respect to R, $\Delta \vdash s$, and t.

Note that $V(\Delta, s) = V(\Delta, s, t)$ because of conditions we put on rewrites that unknowns cannot just 'appear' on the right-hand side. We shall use these to simplify expressions denoting freshness contexts without comment. Note also that R' is freshened also with respect to t; in a sense "R is not allowed to interact with the atoms of s in closed rewriting". Most of this subsection is about making this observation formal, in particular Part 2 of Lemma 4.5 and Theorem 4.7. Theorem 4.9 proves it is computationally useful to consider this restriction. Lemma 4.8 adds "and R cannot inject its own atoms into t", which is later useful for the proof of the Critical Pair Lemma.

For example, the rules in Example 3.3 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 $a\#Z \vdash Z \stackrel{R}{\to} Z$). The canonical example of a closed rewrite is $Z \stackrel{R}{\to}_c Z$. 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 $Z \stackrel{R}{\to}_c Z$ even though R demands to know that some atom a is fresh for X.

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

LEMMA 4.5. *1. For $\Delta \vdash s$ and R, if*

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

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

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

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

PROOF. 1. Nominal Rewriting is equivariant in atoms; if $\Gamma \vdash u \stackrel{S}{\to} v$ then $\Gamma^\kappa \vdash u^\kappa \stackrel{S^\kappa}{\to} v^\kappa$ 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 $\Delta \vdash s \stackrel{R}{\to}_c t$ suggests the choice does not matter since we do not annotate the arrow $\to$ with R', only with R. Part 1 proves this suggestion is correct.

Now we look at some simple examples:

1. If $R' \equiv a''\#X' \vdash X' \to X'$ and $R'' \equiv a'''\#X'' \vdash X'' \to X''$, then if $a\#X, a''\#X \vdash s \stackrel{R'}{\to} t$ then $a\#X, a'''\#X \vdash s \stackrel{R''}{\to} t$.

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.

3. $a \to a$ is not closed, neither is $[a]X \to X$. $a\#X \vdash X \to X$ is closed, so is $p\#U \vdash U \to U$.

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 4.6. *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 position p used in the rewrite is ϵ. So let θ solve $(\nabla \vdash (l, r))\ {}_?{\approx}\ (\Delta \vdash (s, t))$. Recall σ exists solving $(\nabla' \vdash (l', r'))\ {}_?{\approx}\ (\nabla, A(R')\#V(R) \vdash (l, r)$. By syntactic calculations we see that $V(R\sigma) \subseteq V(\Delta, s)$. We can use these facts 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 $\nabla \vdash l \xrightarrow{R} r$, using position ϵ. 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 position ϵ and unpacking what that means we obtain precisely the conditions for closure. $\square$

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

Theorem 4.7. *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. The left-to-right implication is 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 and by the previous theorem we obtain $s \xrightarrow{R^\pi}_c t$ and so $s \xrightarrow{R}_c t$. $\square$

We can re-state this result as follows: closed R captures the rewrites of its equivariance renaming class on ground terms.

Lemma 4.8. *Let R be a closed rule. If $\Delta \vdash s \xrightarrow{R}_c t$ then $A(t) \subseteq A(s)$. If in addition $\Delta \vdash a\#s$, then $\Delta \vdash a\#t$. Similarly for $\Delta \vdash s \xrightarrow{R} t$.*

Proof. It is not hard to see that if $\Delta, A(R')\#V(\Delta, s) \vdash s \xrightarrow{R'} t$ then $A(t) \subseteq A(R') \cup A(s)$. But the same must be true of any other freshening R'', using Part 1 of Lemma 4.5. Taking an intersection we obtain the result.

Now suppose $\Delta \vdash a\#s[l\sigma]_p$; we can also suppose that $a \notin A(R')$. By assumption $\Delta \vdash a\#s[l\sigma]_p$. We now use the structural nature of the deduction rules and the fact that $V(r) \subseteq V(l)$ to construct a proof of $\Delta \vdash a\#s[r\sigma]_p$.

The final part is a consequence of Theorem 4.6. $\square$

Theorem 4.9. *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 by Lemma 4.8 the atoms in it cannot escape into t. $\square$

The restriction to closed rules gives a powerful notion of rewriting: we will show in Section 6 that we can simulate CRSs using closed nominal rules. However, we also study the general case since there are interesting systems (e.g. the π-calculus) with non-closed rules. We come back to this point in the conclusions.

5. CRITICAL PAIRS

Fix an equivariant rewrite system $\mathcal{R}$. Write $\Delta \vdash s \to t_1, t_2$ for the appropriate pair of rewrite judgements. Call a valid pair $\Delta \vdash s \to t_1, t_2$ a **peak**.

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

2. p is a position in l_1.

3. $l_1|_p\ {}_?{\approx}_?\ l_2$ has a solution (Γ, θ), so that $\Gamma \vdash l_1|_p\theta \approx_\alpha l_2\theta$.

Then call the pair of terms-in-context

$$\nabla_1\theta, \nabla_2\theta, \Gamma \vdash (r_1\theta, l_1[r_2\theta]_p)$$

a **critical pair**. If $p = \epsilon$ and R_1, R_2 are copies of the same rule, or if p is the position of a variable in l_1 then we say the critical pair is **trivial**.

Example 5.1. *There are several non-trivial critical pairs in Example 3.3 involving substitution rules. For instance, there is a critical pair between (σ_ϵ) and $(\sigma_{\mathtt{app}})$, and also between (σ_ϵ) and $(\sigma_{\mathtt{lam}})$.*

A critical pair is a pair of terms which can appear in a peak of a rewrite of a term-in-context. Critical pairs are important in term rewriting systems because it is sufficient to check their joinability to deduce joinability of all peaks (i.e. local confluence); we define joinability for NRSs below.

Let $\mathcal{R}$ be the rewrite system containing the rule $\emptyset \vdash [a][b]X \to [b][a]X$. We can verify that $\emptyset \vdash [a][b]X \to [b][a]X$, using the rule $\emptyset \vdash [a][b]X \to [b][a]X \in \mathcal{R}$, but also that $\emptyset \vdash [a][b]X \to [a][b](a\ b)\cdot X$, matching against $\emptyset \vdash [a][b]X \to [b][a]X \in \mathcal{R}$. We can verify that $\emptyset \vdash [a][b](a\ b)\cdot X \approx_\alpha [b][a]X$. This example suggests that any notion of confluence for NRSs must be defined up to $\approx_\alpha$.

Say a nominal rewrite system is **locally confluent** when if $\Delta \vdash s \to t$ and $\Delta \vdash s \to t'$, then u and u' exist such that $\Delta \vdash t \to^* u$ and $\Delta \vdash t' \to^* u'$, and $\Delta \vdash u \approx_\alpha u'$. We say such a peak is **joinable**.

Say a nominal rewrite system is **confluent** when if $\Delta \vdash s \rightarrow^* t$ and $\Delta \vdash s \rightarrow^* t'$, then u and u' exist such that $\Delta \vdash t \rightarrow^* u$ and $\Delta \vdash t' \rightarrow^* u'$, and $\Delta \vdash u \approx_\alpha u'$.

Unlike standard TRS, trivial critical pairs are not necessarily joinable. For instance, consider the rules:

$$\begin{array}{rrcl} b\#X \vdash & (X,b) & \rightarrow & c \\ & f(a) & \rightarrow & b \end{array}$$

There is a trivial critical pair $(c,(b,b))$, obtained by unifying $f(a)$ with $b\#X \vdash X$, which is not joinable.

Also, we need to take equivariance into account when computing critical pairs. We do not insist of a rule $\nabla \vdash l \rightarrow r$ that the atoms in r occur in l. Thus $\emptyset \vdash a \rightarrow bb$ is a valid rule and it is easy to verify that $\emptyset \vdash a \rightarrow bb$ is a valid rewrite. We implicitly close rules up to permutative renamings of variables and atoms, so $\emptyset \vdash a \rightarrow cc$ for any other c not equal to a. Therefore, in the Critical Pair Theorem below we prove local confluence of NRSs under the assumption that *all* critical pairs are joinable. However, if the rules are closed then it is enough to check that the non-trivial critical pairs are joinable, as usual.

Theorem 5.2 (Critical Pair Lemma). *If all critical pairs of a nominal rewrite system are joinable, then it is locally confluent. If the rules are closed then it is sufficient that non-trivial critical pairs be joinable.*

Proof. Suppose $\Delta \vdash s \rightarrow t_1$ and $\Delta \vdash s \rightarrow t_2$ is a peak. Then:

1. There exist $R_i = \nabla_i \vdash l_i \rightarrow r_i$ and positions p_i in s, for $i = 1, 2$.

2. There exist solutions σ_i to $(\nabla \vdash (s[l_i]_{p_i}, s[r_i]_{p_i}))\ {}_?\!\approx (\Delta \vdash (s,t))$ for $i = 1, 2$.

Now there are two possibilities:

1. p_1 and p_2 are in separate subtrees. Local confluence holds by a standard diagrammatic argument taken from the first-order case [2]. We need Lemma 4.2 to account for the weaker notion of equality.

2. p_1 is a prefix of p_2 or vice versa, we consider only the first possibility. Suppose that $p_1 = \epsilon$, the general case follows using Lemma 4.2. Write $p \stackrel{\mathrm{def}}{=} p_2$.

There are now two possibilities:

1. $p = p'q$ for $l_1|_{p'}$ a moderated variable $\pi \cdot X$. This is an instance of a trivial critical pair. If we assume all critical pairs are joinable, we may join the peak. Otherwise, if the rules are closed, local confluence again follows using a standard argument and Lemma 4.2: we can reduce using R_1 then R_2 on any X that are left. Alternatively we can reduce using R_2 on X; we then use the second part of Lemma 4.8 to see that we can still reduce using R_1.

2. $l_1|_p$ is defined and not a moderated variable $\pi \cdot X$. If $p \neq \epsilon$, or $p = \epsilon$ but R_1 and R_2 are *not* copies of the same rule, then this is an instance of a non-trivial critical pair. Therefore we may join the peak. If $p = \epsilon$ and R_1, R_2 are copies of the same rule then $t_1 \approx_\alpha t_2$ and the peak can be trivially joined.

□

Remark 5.3. *As an application of this result, we can deduce that the substitution rules in Example 3.3 are locally confluent: they are closed 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* (σ_{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* (σ_{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 [27], we obtain the following confluence result.

Corollary 5.4.
1. *If an NRS is terminating, and all the critical pairs are joinable, then it is confluent. If it is terminating, closed, and non-trivial critical pairs are joinable, then it is confluent.*
2. *Under the same assumptions, each term has a unique normal form modulo $\approx_\alpha$.*

6. EXPRESSIVE POWER

6.1 Simulating First-Order Rewrite Systems

First-order terms are nominal terms with no atoms, and therefore no permutations and abstractions. In this case $\emptyset \vdash s \approx_\alpha t$ is the same as $s \equiv t$. A first-order rewriting rule $l \rightarrow r$ is a nominal rewrite rule $\emptyset \vdash l \rightarrow r$; clearly all the variable restrictions are satisfied.

We must check that a first-order rewrite step is a nominal rewriting step. It is sufficient to check that standard first-order matching is a particular case of nominal matching. This is straightforward: the nominal matching algorithm with rules to manipulate terms with atoms omitted is the standard first-order matching algorithm [18]. The matching algorithm generates no apartness conditions if a first-order problem $l\ {}_?\!\approx s$ is input. Now if first-order terms l and s can be matched using the substitution θ by these observations the solution is of the form $(\emptyset, \theta)$.

6.2 Simulating CRSs

Simulating CRS reductions is harder.

Firstly the very notion of CRS rewriting is defined on terms without unknowns, not on metaterms (terms with unknowns). At the level of terms there are no significant syntactical differences between CRSs and NRSs: variables in CRSs correspond to atoms, and since there are no metavariables in CRSs terms a CRS term is a ground nominal term. However, CRSs' metaterms cannot be represented directly in the nominal framework: we should obviously map metavariables to variables, but metavariables may have any finite arity whereas variables in NRSs are 0-ary.

Our encoding of CRSs rules takes this into account. We will define a translation function from CRS rules to nominal rules, and we will show that it is possible to simulate a rewrite step in a CRS by a sequence of nominal rewrite steps.

Consider a CRS R over an alphabet A (see section 2.2), with the usual CRS conventions in rules: *different bound variables are used in each abstraction.*

First, we define a nominal signature Σ_A with one sort of atoms (ν), one sort of data (δ), the term sorts generated from these, and a set of function symbols which contains the function symbols of the CRS R and a new function symbol `sub` representing substitution, which we sugar as before to $t[a\mapsto s]$, or more generally $t[a_1\mapsto s_1,\ldots,a_n\mapsto s_n]$.

Given a rule $l \to r$ in R, we define a partial mapping Φ from metavariables to lists of variables, such that

$$\Phi(Z^k) = [a_1,\ldots,a_k]$$

if the leftmost occurrence of the metavariable Z^k in l has the form $Z^k(a_1,\ldots,a_k)$. Recall that by definition, in left-hand sides of CRS rules metavariables occur only in this form, where $a_1,\ldots,a_k$ are pairwise different bound variables. We denote by $\Phi(Z^k)_i$ the ith element in the list $\Phi(Z^k)$.

To translate the rule $l \to r$ into a nominal rule over Σ_A, we will apply two different functions to l and r, both parameterised by Φ.

The first translation, which we denote $(\cdot)^\circ_\Phi$, works on pairs (Δ, t) of an apartness context and a metaterm. The idea is to replace, for each metavariable Z^k in l, the leftmost subterm of the form $Z^k(a_1,\ldots,a_k)$ by Z^k, and the other subterms $Z^k(b_1,\ldots,b_k)$ by $(a_1\ b_1)\cdot\ldots\cdot(a_k\ b_k)\cdot Z^k$, adding to the apartness context Δ the conditions $b_1\#Z^k,\ldots,b_k\#Z^k$.

It is formally defined as follows:

$$\begin{array}{lcl}
(\Delta, a)^\circ_\Phi & = & (\Delta, a)\\
(\Delta, [a]t)^\circ_\Phi & = & (\Delta', [a]t'),\\
 & & \text{where } (\Delta', t') = (\Delta, t)^\circ_\Phi\\
(\Delta, f(t_1,\ldots,t_n))^\circ_\Phi & = & (\Delta', f(t'_1,\ldots,t'_n)),\\
 & & \text{where } (\Delta, t_i)^\circ_\Phi = (\Delta_i, t'_i),\\
 & & \text{and } \Delta' = \bigcup_i \Delta_i\\
(\Delta, Z^k(b_1,\ldots,b_k))^\circ_\Phi & = & (\Delta\cup\Delta', Z^k) \text{ if leftmost in } l,\\
 & & \Delta' = \{b_1,\ldots,b_k\#X \mid\\
 & & \quad X\in V(l), X\neq Z^k\}
\end{array}$$

$$\begin{array}{l}
(\Delta, Z^k(b_1,\ldots,b_k))^\circ_\Phi \ =\\
(\Delta\cup\Delta',\ (\Phi(Z^k)_1\ b_1)\cdot\ldots\cdot(\Phi(Z^k)_k\ b_k)\cdot Z^k) \text{ otherwise,}\\
\qquad\qquad\qquad \Delta' = \{b_1,\ldots,b_k\#X \mid X\in V(l)\}
\end{array}$$

The second translation function, when applied to r, produces $(\Delta_r, [\![r]\!]_\Phi)$, where subterms of the form $Z^k(t_1,\ldots,t_k)$ in r are replaced by

$$Z^k[\Phi(Z^k)_1\mapsto[\![t_1]\!]_\Phi,\ldots,\Phi(Z^k)_k\mapsto[\![t_k]\!]_\Phi]$$

and $\Delta_r = \{a_i\#Z_j \mid a_i, Z_j \text{ occur in } r\}$.

$[\![r]\!]_\Phi$ is formally defined by:

$$\begin{array}{lcl}
[\![a]\!]_\Phi & = & a\\
[\![([a]t)]\!]_\Phi & = & [a][\![t]\!]_\Phi\\
[\![f(t_1,\ldots,t_n)]\!]_\Phi & = & f([\![t_1]\!]_\Phi,\ldots,[\![t_n]\!]_\Phi)\\
[\![Z(t_1,\ldots,t_k)]\!]_\Phi & = & Z[\Phi(Z)_1\mapsto[\![t_1]\!]_\Phi,\ldots,\Phi(Z)_k\mapsto[\![t_k]\!]_\Phi]
\end{array}$$

We define the **translation** of the CRS rule $l \to r$ as $\Delta \vdash l' \to r'$ where $(\emptyset, l)^\circ_\Phi = (\Delta', l')$, $\Delta = \Delta'\cup\Delta_r$ and $[\![r]\!]_\Phi = r'$.

We give some examples to illustrate the definition.

Example 6.1. *The translation of the β-rule shown in Example 2.1 according to the definitions above is:*

$$a\#Z' \ \vdash\ \mathsf{app}(\mathsf{lam}([a]Z), Z') \to Z[a\mapsto Z']$$

Consider now a CRS rule defining a differentiation operator:

$$\mathsf{diff}([a]\mathsf{sin}(Z(a))) \to [b]\mathsf{mult}(\mathsf{app}(\mathsf{diff}([c]Z(c)), b), \mathsf{cos}(Z(b)))$$

The translation of this rule is: $b,c\#Z \ \vdash\ \mathsf{diff}([a]\mathsf{sin}(Z)) \to [b]\mathsf{mult}(\mathsf{app}(\mathsf{diff}([c]Z[a\mapsto c]), b), \mathsf{cos}(Z[a\mapsto b]))$

Lemma 6.2. *Let A be a CRS alphabet and Σ_A a nominal signature as defined above. If $l \to r$ is a CRS rule over the alphabet A and $\Delta \vdash l' \to r'$ is its translation, then $\Delta \vdash l' \to r'$ is a closed nominal rewrite rule over Σ_A.*

Proof. By induction on the structure of terms, we can show that the translations of l and r are nominal terms-in-contexts. Since the metavariables that occur in r occur also in l, all the variables of r' and Δ occur in l', therefore $\Delta \vdash l' \to r'$ is a nominal rewrite rule. It is closed, since

1. l and r do not contain free variables by definition,
2. our translation respects the structure of terms,
3. all the atoms introduced (by substitution operators) are abstracted,
4. when an abstraction is created in r', it abstracts a fresh atom.

□

Let us denote by $\mathcal{R}$ the nominal rewriting system obtained by translating in this way all the rules of the CRS R and adding the rules that push the substitutions through the other constructs. The latter can be mechanically generated from the function symbols in the signature:

$$\begin{array}{llll}
(\sigma_{\mathtt{var}}) & & a[a\mapsto X] & \to\ X\\
(\sigma_\epsilon) & a\#Y\ \vdash & Y[a\mapsto X] & \to\ Y\\
(\sigma_f) & & (f\ X)[a\mapsto Y] & \to\ fX[a\mapsto Y]\\
 & & & \quad \text{for each } f \text{ in } \Sigma\\
(\sigma_{prod}) & & (X_1,\ldots,X_n)[a\mapsto Y] & \to\\
 & & & (X_1[a\mapsto Y],\ldots,X_n[a\mapsto Y])\\
(\sigma_{abs}) & b\#Y\ \vdash & ([b]X)[a\mapsto Y] & \to\ [b](X[a\mapsto Y])
\end{array}$$

We will use two properties of these rules:

Property 6.3. *1. The substitution rules terminate.*

2. The substitution rules are confluent on ground nominal terms.

The termination of the substitution rules can be easily shown by a standard interpretation argument, since the rules move the substitution operators down towards the leaves of the term.

Confluence for ground terms then follows from the fact that the rules are closed and all non-trivial critical pairs are joinable (see Remark 5.3). Let us denote $nf_\sigma(\Delta \vdash t)$ the normal form of a term-in-context $\Delta \vdash t$ (this is uniquely defined modulo $\approx_\alpha$ by Corollary 5.4).

Lemma 6.4 (Correctness of substitution). *Let t, s be terms in a CRS R (and therefore also in $\mathcal{R}$). Then $nf_\sigma(t[a\mapsto s]) \approx_\alpha t[a := s]$ where $t[a := s]$ denotes the term obtained by substituting (using the higher-order substitution of the CRS) a by s in t.*

We omit the proof by induction on t, which is standard in explicit substitution systems.

Theorem 6.5 (Soundness). *Let t be a term in a CRS R (and therefore also in $\mathcal{R}$). If $\vdash t \to_{\mathcal{R}} u$ then there exists u' such that $\vdash u \to^*_{\mathcal{R}} u'$ and $t \to_R u'$.*

Proof. Take $u' \approx_\alpha nf_\sigma(u)$, which is a CRS term by Lemma 6.4. We will show that $t \to_R u'$.

Assume $\vdash t \to_{\mathcal{R}} u$ using a rule $\Delta \vdash l' \to r'$ which is the translation of $l \to r \in R$ (i.e. $r' = [\![r]\!]_\Phi$), at position p (i.e. $t|_p$ matches l' using $(\emptyset, \theta')$ and $\Delta\theta' = \emptyset$). Then $u \approx_\alpha t[r'\theta']_p$ and $nf_\sigma(u) \approx_\alpha t[nf_\sigma(r'\theta')]_p$.

Let $Z_1, \ldots, Z_k$ be the metavariables in l (and therefore also the variables in l'), and assume that for each Z_i, $\Phi(Z_i) = [a_{i1}, \ldots, a_{ip_i}]$. Let

$$\theta' = \{Z_1 \mapsto t_1, \ldots, Z_k \mapsto t_k\}$$

then

$$\theta = \{Z_1 \mapsto \lambda a_{11} \ldots a_{1p_1}.t_1, \ldots, Z_k \mapsto \lambda a_{k1} \ldots a_{kp_k}.t_k\}$$

matches l and $t|_p$.

We show by induction on r that $r\theta \approx_\alpha nf_\sigma([\![r]\!]_\Phi\theta')$, which completes the proof. The only interesting case is when r is a metaterm $Z_i(u_1, \ldots, u_{p_i})$. Then

$$r' = [\![r]\!]_\Phi = Z_i[a_{i1} {\mapsto} [\![u_1]\!]_\Phi, \ldots, a_{ip_i} {\mapsto} [\![u_{p_i}]\!]_\Phi]$$

and

$$r\theta = t_i[a_{i1} := u_1\theta, \ldots, a_{ip_i} := u_{p_i}\theta].$$

By induction and Lemma 6.4:

$$r\theta \approx_\alpha nf_\sigma(t_i[a_{i1} {\mapsto} nf_\sigma([\![u_1]\!]_\Phi\theta'), \ldots, a_{ip_i} {\mapsto} nf_\sigma([\![u_{p_i}]\!]_\Phi\theta')])$$

By unicity of normal forms the latter coincides (modulo $\approx_\alpha$) with $nf_\sigma(t_i[a_{i1} {\mapsto} [\![u_1]\!]_\Phi\theta', \ldots, a_{ip_i} {\mapsto} [\![u_{p_i}]\!]_\Phi\theta']$, which in turn is $nf_\sigma(r'\theta')$. ☐

Theorem 6.6 (Completeness). *Let t and u be arbitrary terms in the CRS R (and therefore also in $\mathcal{R}$). If $t \to_R u$ then $\vdash t \to^*_{\mathcal{R}} u$.*

Proof. The main points of the proof are:

- Matching in CRSs can be simulated by nominal matching thanks to the condition on metavariables occurring on left-hand sides of CRSs' rules. No apartness context is generated if the terms in the CRS follow the usual Barendregt naming conventions.
- Our explicit substitution rules simulate the implicit substitution mechanism of CRSs (Lemma 6.4).

Then $t \to_R u$ implies $t \to_{\mathcal{R}} u'$ such that $nf_\sigma(u') \approx_\alpha u$. Therefore, $t \to^*_{\mathcal{R}} u$ as required. ☐

Since the translation of a CRS rule is a closed nominal rule, and the rules for substitution are also closed, the class of closed nominal rewriting systems is sufficient to simulate CRS reductions.

7. CONCLUSIONS AND FUTURE WORK

The technical foundations and style of this work are most directly derived from work on nominal logic [29] and nominal unification [33]. We use a nominal matching algorithm, which is easy to derive from the unification algorithm and 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 context Δ, but Δ is fixed and does not seem to behave perniciously.

This system is also powerful, in that it encodes first-order systems and CRSs. It can be seen as an explicit substitution version of higher-order rewriting, but with built-in α-conversion. We could easily implement a higher-order rewrite system using a nominal rewrite system: we only need to make the substitution explicit. We can spare the effort of 'implementing' α-conversion using de Bruijn indices and all the associated machinery.

Future Work.

In nominal rewriting, $\Delta \vdash s \xrightarrow{R} t$ and $\Delta \vdash s \xrightarrow{R}_c t$, the context Δ is static. This is because rewrite rules have no mechanism for locally generating a fresh atom. This does occur in nature, for example in a rewrite system for the π-calculus we might wish a reaction rule which we might write as $\text{И}n.\ (\nu[n]X) \to X$, where И indicates that n must be fresh for all variables in the context in which the rewrite is introduced, which would be implemented by allowing rewrites to expand Δ with freshness conditions. The theory of this remains to be considered though Miller and Tiu's work [26] seems related in spirit.

The judgement $\Delta \vdash s \to t$ has a 'logical' flavour with its freshness context and deduction rules. It is easy to make this more concrete by allowing $a\#u$ in Δ for u a term, rather than only $a\#X$. It suffices to introduce rules like in (4) but on the left, to obtain a logical system. It is not hard to prove substitution lemmas and cut-elimination results. But we can also ask what happens if we allow implications, quantifiers, and equalities, on the left and right; we claim that this is a 'logic for matching and unification'. If we restrict to notions of Horn or Harrop clause the deduction rules of this logic reproduce algorithms from logic programming, where the status of the unknowns X is a sort of 'object-level unknown' distinct from the 'meta-level unknowns' which we write s, t, and u. The first-order flavour is preserved since we have abstraction but no β-conversion. We can take this even further if we introduce a И term-former for unknowns X.

We can also turn rewriting inside-out to **generalisation** on terms-with-abstraction, for example for machine learning in First-Order Logic. This is the problem of given s and t finding a least u and substitutions σ_1 and σ_2 such that $u\sigma_1 = s$ and $u\sigma_2 = t$. The advantage of our unknowns X over, say, higher-order variables T, is that we maintain the first-order flavour because there is no β-conversion, and that the arity of X does not change according to the number of atoms is contains, whereas the arity of T does determine how many arguments it may take.

Acknowledgements.

We would like to thank James Cheney, Andy Pitts, and Christian Urban for enlightening discussions and valuable comments on previous versions of this paper.

8. REFERENCES

[1] M. Abadi, L. Cardelli, P-L. Curien, and J-J. Lévy, *Explicit substitutions*, Journal of Functional Programming **1** (1991), no. 4, 375–416.

[2] F. Baader and T. Nipkow, *Term rewriting and all that*, Cambridge University Press, 1998.

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

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

[5] C. Bertolissi, H. Cirstea, and C. Kirchner, *Translating combinatory reduction systems into the rewriting calculus*, 4th International Workshop on Rule-Based Programming (RULE 2003), Valencia, Spain, 2003.

[6] E. Bonelli, D. Kesner, and A. Ríos, *From higher-order to first-order rewriting*, Proc. 12th Int. Conf. Rewriting Techniques and Applications, Lecture Notes in Computer Science, vol. 2051, Springer, 2001.

[7] V. Breazu-Tannen and J. Gallier, *Polymorphic rewriting conserves algebraic strong normalization*, Theoretical Computer Science **83**:1 (1991).

[8] V. Breazu-Tannen and J. Gallier, *Polymorphic rewriting conserves algebraic confluence*, Information and Computation **82** (1992), 3–28.

[9] J. Cheney, *The complexity of equivariant unification.* Proceedings of ICALP 2004, to appear.

[10] H. Cirstea and C. Kirchner, *The Rewriting Calculus - Part I*, Logic Journal of the Interest Group in Pure and Applied Logics **9** (2001), 363–399.

[11] H. Cirstea and C. Kirchner, *The Rewriting Calculus - Part II*, Logic Journal of the Interest Group in Pure and Applied Logics **9** (2001), 401–434.

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

[13] N. Dershowitz and J.-P. Jouannaud, *Rewrite Systems*, Handbook of Theoretical Computer Science: Formal Methods and Semantics (J. van Leeuwen, ed.), vol. B, North-Holland, 1989.

[14] M. J. Gabbay, *The π-calculus in FM*, Thirty-five years of Automath (Fairouz Kamareddine, ed.), Kluwer, 2003.

[15] M. J. Gabbay and A. M. Pitts, *A New Approach to Abstract Syntax with Variable Binding*. Formal Aspects of Computing **13**, pp. 341–363, 2002. A preliminary version appeared in the Proc. LICS 1999.

[16] M. J. Gabbay, *Fresh graphs*, Submitted, November 2003.

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

[18] J.-P. Jouannaud and C. Kirchner, *Solving equations in abstract algebras: a rule based survey of unification*, Computational Logic: Essays in Honor of Alan Robinson (J-L. Lassez and G. Plotkin, eds.), MIT Press, 1991.

[19] J.-P. Jouannaud and M. Okada, *Executable higher-order algebraic specification languages*, Proc. 6th IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press, 1991, pp. 350–361.

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

[21] Z. Khasidashvili, *Expression reduction systems*, Proceedings of I.Vekua Institute of Applied Mathematics (Tbisili), vol. 36, 1990, pp. 200–220.

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

[23] J-W. Klop, *Combinatory reduction systems*, Mathematical Centre Tracts, vol. 127, Mathematischen Centrum, 413 Kruislaan, Amsterdam, 1980.

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

[25] R. Mayr and T. Nipkow, *Higher-order rewrite systems and their confluence*, Theoretical Computer Science **192** (1998), 3–29.

[26] D. Miller and A. Tiu, *A proof theory for generic judgments: An extended abstract*, Proceedings of LICS 2003, IEEE Computer Society Press, 2003, 118–127.

[27] M.H.A. Newman, *On theories with a combinatorial definition of equivalence*, Annals of Mathematics **43** (1942), no. 2, 223–243.

[28] B. Pagano, *Des calculs de substitution explicite et de leur application à la compilation des langages fonctionnels*, Ph.D. thesis, Université de Paris 6, 1998.

[29] A. M. Pitts, *Nominal logic, a first order theory of names and binding*, Information and Computation **186** (2003), 165–193. 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.

[30] A. M. Pitts and M. J. Gabbay, *A metalanguage for programming with bound names modulo renaming*, Mathematics of Program Construction. 5th International Conference, MPC2000, Ponte de Lima, Portugal, July 2000. Proceedings (R. Backhouse and J. N. Oliveira, eds.), Lecture Notes in Computer Science, vol. 1837, Springer-Verlag, Heidelberg, 2000, pp. 230–255.

[31] F. van Raamsdonk, *Confluence and normalisation for higher-order rewriting*, Ph.D. thesis, Free University of Amsterdam, 1996.

[32] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay, *FreshML: Programming with binders made simple*, Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP 2003), Uppsala, Sweden, ACM Press, August 2003, pp. 263–274.

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

[34] C. Urban, *Nominal matching*, preliminary report, 2004.