Nominal Rewriting

Maribel Fernández

King’s College London

ISR - June 2009
Introduction

• First-order languages
• Languages with binding operators

Specifying binders:

• $\alpha$-equivalence
• Nominal terms
• Nominal unification (unification modulo $\alpha$-equivalence)
• Nominal matching (matching modulo $\alpha$-equivalence)

Nominal rewriting

• Extending first-order rewriting
• Examples
• Properties
• Expressive Power
Further reading

First-order languages vs. languages with binders

Most programming languages permit the specification of first-order data structures and first-order operators.

Examples of first-order data structures: numbers, lists, trees, etc.
Example of first-order operator on lists:

\[
\text{append}(\text{nil}, x) \rightarrow x
\]
\[
\text{append}(\text{cons}(x, z), y) \rightarrow \text{cons}(x, \text{append}(z, y))
\]

Very few programming or specification languages give programmers tools to specify data structures that include binding operators. However, in many situations, we need to manipulate data with bound names.

Well-known examples can be found in software that deals with programs (e.g. in compilers, and in software used to analyse programs, type them, optimise them, etc).
Some concrete examples of binding operators (informally):

- **Operational semantics:**
  \[
  \text{let } a = N \text{ in } M \longrightarrow (\text{fun } a.M)N
  \]
Some concrete examples of binding operators (informally):

- Operational semantics:
  \[
  \text{let } a = N \text{ in } M \rightarrow (\text{fun } a. M)N
  \]

- $\beta$ and $\eta$-reductions in the $\lambda$-calculus:
  \[
  (\lambda x. M)N \rightarrow M[x/N]
  \]
  \[
  (\lambda x. Mx) \rightarrow M \quad (x \notin \text{fv}(M))
  \]
Some concrete examples of binding operators (informally):

- **Operational semantics:**

  \[
  \text{let } a = N \text{ in } M \rightarrow (\text{fun } a. M)N
  \]

- **\(\beta\) and \(\eta\)-reductions in the \(\lambda\)-calculus:**

  \[
  (\lambda x. M)N \rightarrow M[x/N] \\
  (\lambda x. Mx) \rightarrow M \quad (x \notin \text{fv}(M))
  \]

- **\(\pi\)-calculus:**

  \[
  P \mid \nu a. Q \rightarrow \nu a. (P \mid Q) \quad (a \notin \text{fv}(P))
  \]
Some concrete examples of binding operators (informally):

- **Operational semantics:**
  \[
  \text{let } a = N \text{ in } M \rightarrow (\text{fun } a.M)N
  \]

- **β and η-reductions in the λ-calculus:**
  \[
  (\lambda x.M)N \rightarrow M[x/N]
  (\lambda x.Mx) \rightarrow M \quad (x \notin \text{fv}(M))
  \]

- **π-calculus:**
  \[
  P \mid \nu a.Q \rightarrow \nu a.\left(P \mid Q\right) \quad (a \notin \text{fv}(P))
  \]

- **Logic equivalences:**
  \[
  P \text{ and } (\forall x.Q) \leftrightarrow \forall x(P \text{ and } Q) \quad (x \notin \text{fv}(P))
  \]
Binding operators are defined modulo renaming of bound variables, i.e., $\alpha$-equivalence.

Example:
In $\forall x. P$ the variable $x$ can be renamed. Take any fresh variable $y$, then

$$\forall x. P =_\alpha \forall y. P \{ x \mapsto y \}$$

Substitution of a bound name by a term has to avoid capture of other bound names.

How can we formally define (or program) binding operators? There are several alternatives.
We can encode $\alpha$-equivalence in a first-order specification or programming language.

⇒ We have a simple notion of substitution (first-order). (+)
We can encode $\alpha$-equivalence in a first-order specification or programming language.

- We have a simple notion of substitution (first-order). (+)

$\Rightarrow$ Efficient matching and unification algorithms. (+)
First-order frameworks

We can encode $\alpha$-equivalence in a first-order specification or programming language.

- We have a simple notion of substitution (first-order). (+)
- Efficient matching and unification algorithms. (+)

$\Rightarrow$ No binders. (-)
We can encode $\alpha$-equivalence in a first-order specification or programming language.

- We have a simple notion of substitution (first-order). (+)
- Efficient matching and unification algorithms. (+)
- No binders. (-)

$\Rightarrow$ We need to 'implement' $\alpha$-equivalence and non-capturing substitution from scratch. (-)
We can encode $\alpha$-equivalence in a first-order specification or programming language.

- We have a simple notion of substitution (first-order). (+)
- Efficient matching and unification algorithms. (+)
- No binders. (-)
- We need to 'implement' $\alpha$-equivalence and non-capturing substitution from scratch. (-)

$\Rightarrow$ For example, we can encode a system with binders such as the lambda-calculus using numbers to represent bound variables and operators such as “lift” and “shift” to encode non-capturing substitution (cf. De Bruijn’s notation).
Higher-order rewrite systems (CRS, HRS, etc.) include a general binding construct and terms are defined modulo $\alpha$-equivalence.

Example: $\beta$-rule

$$\text{app}(\text{lam}([a]Z(a)), Z') \rightarrow Z(Z')$$

One step of rewriting:

$$\text{app}(\text{lam}([a]f(a, g(a)), b) \rightarrow f(b, g(b))$$

using (a restriction of) higher-order matching.
Higher-order frameworks

- Higher-order rewrite systems (CRS, HRS, etc.) include a general binding construct and terms are defined modulo \(\alpha\)-equivalence.
  Example: \(\beta\)-rule

\[
\text{app}(\text{lam}(\lbrack a \rbrack Z(a)), Z') \rightarrow Z(Z')
\]

One step of rewriting:

\[
\text{app}(\text{lam}(\lbrack a \rbrack f(a, g(a)), b) \rightarrow f(b, g(b))
\]

using (a restriction of) higher-order matching.

- Logical frameworks based on Higher-Order Abstract Syntax also work modulo \(\alpha\)-equivalence.

\[
\text{let } a = N \text{ in } M(a) \rightarrow (\text{fun } a \rightarrow M(a))N
\]
The syntax includes binders. (+)

• Implicit α-equivalence. (+)
• We targeted α but now we have to deal with β too. (-)
• Substitution is a meta-operation using β. (-)
• Unification is undecidable in general. (-)
• Leaving name dependencies implicit is convenient, e.g.
  \texttt{let a = N in M} vs. \texttt{let a = N in M (a) app (lambda [a] Z, Z')} vs. \texttt{app (lam ([a] Z (a)), Z')}.
Higher-order frameworks

- The syntax includes binders. (+)
  \[ \Rightarrow \text{Implicit } \alpha\text{-equivalence. (+)} \]
- The syntax includes binders. (+)
- Implicit $\alpha$-equivalence. (+)

$\Rightarrow$ We targeted $\alpha$ but now we have to deal with $\beta$ too. (-)
Higher-order frameworks

- The syntax includes binders. (+)
- Implicit $\alpha$-equivalence. (+)
- We targeted $\alpha$ but now we have to deal with $\beta$ too. (-)

⇒ Substitution is a meta-operation using $\beta$. (-)
Higher-order frameworks

- The syntax includes binders. (+)
- Implicit $\alpha$-equivalence. (+)
- We targeted $\alpha$ but now we have to deal with $\beta$ too. (-)
- Substitution is a meta-operation using $\beta$. (-)

$\Rightarrow$ Unification is undecidable in general. (-)
Higher-order frameworks

- The syntax includes binders. (+)
- Implicit $\alpha$-equivalence. (+)
- We targeted $\alpha$ but now we have to deal with $\beta$ too. (-)
- Substitution is a meta-operation using $\beta$. (-)
- Unification is undecidable in general. (-)

$\Rightarrow$ Leaving name dependencies implicit is convenient, e.g.

\[
\text{let } a = N \text{ in } M \quad \text{vs.} \quad \text{let } a = N \text{ in } M(a)
\]

\[
\text{app}(\text{lambda}[a]Z, Z') \quad \text{vs.} \quad \text{app}(\text{lam}([a]Z(a)), Z').
\]
Nominal Approach

Inspired by the work on Nominal Logic (Pitts et al.)
Key ideas: Freshness conditions $a \# t$, name swapping $(a \ b) \cdot t$.
Example: $\beta$ and $\eta$ rules as NRS:

\[
\begin{align*}
app(lam([a]Z), Z') & \rightarrow subst([a]Z, Z') \\
\alpha \# M \vdash (\lambda([a]app(M, a))) & \rightarrow M
\end{align*}
\]

$\Rightarrow$ Terms with binders.
Nominal Approach

Inspired by the work on Nominal Logic (Pitts et al.)
Key ideas: Freshness conditions $a\#_t$, name swapping $(a b) \cdot t$.
Example: $\beta$ and $\eta$ rules as NRS:

\[
\begin{align*}
\text{app(lam([a]Z), Z')} & \rightarrow \text{subst([a]Z, Z')} \\
\text{a\#M} \vdash (\lambda([a]\text{app}(M, a))) & \rightarrow M
\end{align*}
\]

- Terms with binders.

$\Rightarrow$ Built-in $\alpha$-equivalence.
Inspired by the work on Nominal Logic (Pitts et al.)
Key ideas: Freshness conditions $a\#t$, name swapping $(a \, b) \cdot t$.
Example: $\beta$ and $\eta$ rules as NRS:

\[
\text{app}(\text{lam}([a]Z), Z') \rightarrow \text{subst}([a]Z, Z')
\]

\[
a\#M \vdash (\lambda([a]\text{app}(M, a)) \rightarrow M
\]

- Terms with binders.
- Built-in $\alpha$-equivalence.

$\Rightarrow$ Simple notion of substitution (first order).
Nominal Approach

Inspired by the work on Nominal Logic (Pitts et al.)
Key ideas: Freshness conditions $a\#t$, name swapping $(a \ b) \cdot t$.
Example: $\beta$ and $\eta$ rules as NRS:

\[
\text{app(lam([a]Z), Z')} \rightarrow \text{subst([a]Z, Z')}
\]

\[
a\#M \vdash (\lambda([a]\text{app}(M, a)) \rightarrow M
\]

- Terms with binders.
- Built-in $\alpha$-equivalence.
- Simple notion of substitution (first order).

$\Rightarrow$ Efficient matching and unification algorithms.
Nominal Approach

Inspired by the work on Nominal Logic (Pitts et al.)
Key ideas: Freshness conditions \( a \not\# t \), name swapping \((a \ b) \cdot t\).
Example: \( \beta \) and \( \eta \) rules as NRS:

\[
\begin{align*}
\text{app}(\text{lam}([a]Z), Z') & \rightarrow \text{subst}([a]Z, Z') \\
a \not\# M \vdash (\lambda([a]\text{app}(M, a)) & \rightarrow M
\end{align*}
\]

- Terms with binders.
- Built-in \( \alpha \)-equivalence.
- Simple notion of substitution (first order).
- Efficient matching and unification algorithms.

\( \Rightarrow \) Dependencies of terms on names are implicit.
Inspired by the work on Nominal Logic (Pitts et al.)
Key ideas: Freshness conditions \( a \not\# t \), name swapping \( (a \ b) \cdot t \).
Example: \( \beta \) and \( \eta \) rules as NRS:

\[
\text{app(lam([a]Z), Z')} \rightarrow \text{subst([a]Z, Z')}
\]
\[
a \not\# M \vdash (\lambda([a]\text{app}(M, a)) \rightarrow M
\]

- Terms with binders.
- Built-in \( \alpha \)-equivalence.
- Simple notion of substitution (first order).
- Efficient matching and unification algorithms.
- Dependencies of terms on names are implicit.

\( \Rightarrow \) Easy to express conditions such as \( a \notin \text{fv}(M) \)
Inspired by the work on Nominal Logic (Pitts et al.)
Key ideas: Freshness conditions $a \# t$, name swapping $(a \ b) \cdot t$.
Example: $\beta$ and $\eta$ rules as NRS:

$$\text{app}(\text{lam}([a]Z), Z') \rightarrow \text{subst}([a]Z, Z')$$
$$a \# M \vdash (\lambda([a]\text{app}(M, a)) \rightarrow M$$

- Terms with binders.
- Built-in $\alpha$-equivalence.
- Simple notion of substitution (first order).
- Efficient matching and unification algorithms.
- Dependencies of terms on names are implicit.
- Easy to express conditions such as $a \notin \text{fv}(M)$

$\Rightarrow$ Can be easily generalised to express more general constraints.
• Function symbols: $f, g \ldots$
Variables: $M, N, X, Y, \ldots$
Atoms: $a, b, \ldots$
Swappings: $(a \ b)$
  Def. $(a \ b)a = b, (a \ b)b = a, (a \ b)c = c$
Permutations: lists of swappings, denoted $\pi$ ($\text{Id}$ empty).
Nominal Syntax

- Function symbols: $f, g \ldots$
- Variables: $M, N, X, Y, \ldots$
- Atoms: $a, b, \ldots$
- Swappings: $(a \ b)$
  - Def. $(a \ b)a = b, (a \ b)b = a, (a \ b)c = c$
- Permutations: lists of swappings, denoted $\pi$ ($ld$ empty).

- Nominal Terms:

  $s, t ::= a \mid \pi \cdot X \mid [a]t \mid f \ t \mid (t_1, \ldots, t_n)$

  $ld \cdot X$ written as $X$. 
Nominal Syntax

- Function symbols: $f, g \ldots$
  Variables: $M, N, X, Y, \ldots$
  Atoms: $a, b, \ldots$
  Swappings: $(a \ b)$
    Def. $(a \ b)a = b, (a \ b)b = a, (a \ b)c = c$
  Permutations: lists of swappings, denoted $\pi$ ($ld$ empty).
- Nominal Terms:

  $$s, t ::= a \mid \pi \cdot X \mid [a]t \mid f \ t \mid (t_1, \ldots, t_n)$$

  $ld \cdot X$ written as $X$.
- Example (ML): $\text{var}(a), \text{app}(t, t'), \text{lam}([a]t), \text{let}(t, [a]t'), \text{letrec}[f]([a]t, t'), \text{subst}([a]t, t')$
  Syntactic sugar:
  $a, (tt'), \lambda a.t, \text{let } a = t \text{ in } t', \text{letrec } fa = t \text{ in } t', t[a \mapsto t']$
We use freshness to avoid name capture. $a\#X$ means $a \not\in \text{fv}(X)$ when $X$ is instantiated.

$$
\begin{align*}
\frac{}{a \equiv_\alpha a} & \\
\frac{ds(\pi, \pi') \# X}{\pi \cdot X \equiv_\alpha \pi' \cdot X}
\end{align*}
$$

$$
\begin{align*}
\frac{s_1 \equiv_\alpha t_1 \cdots s_n \equiv_\alpha t_n}{(s_1, \ldots, s_n) \equiv_\alpha (t_1, \ldots, t_n)} & \\
\frac{s \equiv_\alpha t}{fs \equiv_\alpha ft}
\end{align*}
$$

$$
\begin{align*}
\frac{s \equiv_\alpha t}{[a]s \equiv_\alpha [a]t} & \\
\frac{a\#t}{[b]s \equiv_\alpha [b]t}
\end{align*}
$$

where

$$
ds(\pi, \pi') = \{ n | \pi(n) \neq \pi'(n) \}
$$

\begin{itemize}
\item $a\#X, b\#X \vdash (a \ b) \cdot X \equiv_\alpha X$
\end{itemize}
\[\alpha\text{-equivalence}\]

We use freshness to avoid name capture.
\(a \# X\) means \(a \notin \text{fv}(X)\) when \(X\) is instantiated.

\[
\begin{array}{c}
a \approx_\alpha a \\
ds(\pi, \pi') \# X \\
\pi \cdot X \approx_\alpha \pi' \cdot X
\end{array}
\]

\[
\begin{array}{c}
(s_1, \ldots, s_n) \approx_\alpha (t_1, \ldots, t_n) \\
s \approx_\alpha t
\end{array}
\]

\[
\begin{array}{c}
(s_1 \ldots, s_n) \approx_\alpha (t_1 \ldots, t_n) \\
s \approx_\alpha t
\end{array}
\]

\[
\begin{array}{c}
(a \# t) \quad s \approx_\alpha (a \ b) \cdot t
\end{array}
\]

where

\[
\begin{align*}
ds(\pi, \pi') &= \{n | \pi(n) \neq \pi'(n)\} \\
(a \# X, b \# X) &\vdash (a \ b) \cdot X \approx_\alpha X \\
b \# X &\vdash \lambda[a]X \approx_\alpha \lambda[b](a \ b) \cdot X
\end{align*}
\]
Also defined by induction:

\[
\begin{align*}
\text{Also defined by induction:} & \\
\frac{\quad \quad a\#b \quad \quad a\#[a)s}{a\#[\pi \cdot X} & \quad \quad \frac{\pi^{-1}(a)\#X}{a\#[}\]
\end{align*}
\]

\[
\begin{align*}
\frac{a\#s_1 \cdots a\#s_n}{a\#(s_1, \ldots, s_n)} & \quad \quad \frac{a\#s}{a\#fs} & \quad \quad \frac{a\#s}{a\#[b]s}
\end{align*}
\]
Nominal rewriting is rewriting with nominal terms. Rewrite rules can be used to define

- equational theories, and theorem provers;
- algebraic specifications of operators and data structures;
- operational semantics of programs;
- a theory of functions;
- a theory of processes;
- etc.

Nominal terms can naturally express binding operators, and have been used as a basis for the definition of specification and programming languages.

Nominal terms have good computational properties: Efficient algorithms to check $\alpha$-equivalence, efficient matching and unification.
Unification has been a popular research field in the last years, motivated by the need to obtain efficient implementations for use in logic programming languages and theorem provers. Unification algorithms play a central role in the implementation of resolution. *Prolog* is one of the most popular logic programming languages. Logic programming languages

- use *logic* to express knowledge, describe a problem;
- use *inference* to compute a solution to a problem.

Prolog = Clausal Logic + Resolution + Control Strategy
Domain of computation:

**Herbrand Universe:** set of *terms* over a universal alphabet of

- *variables:* \( X, Y, \ldots \)
- and function symbols \((f, g, h, \ldots)\) with fixed arities (the arity of a symbol is the number of arguments associated with it).

A *term* is either a variable, or has the form \( f(t_1, \ldots, t_n) \) where \( f \) is a function symbol of arity \( n \) and \( t_1, \ldots, t_n \) are terms.

**Example:** \( f(f(X, g(a)), Y) \) where \( a \) is a constant, \( f \) a binary function, and \( g \) a unary function.
Values:

Values are also terms, that are associated to variables by means of automatically generated substitutions, called most general unifiers.

**Definition:** A substitution is a partial mapping from variables to terms, with a finite domain. We denote a substitution $\sigma$ by:

$$\{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}. \quad dom(\sigma) = \{X_1, \ldots, X_n\}.$$  

A substitution $\sigma$ is applied to a term $t$ or a literal $l$ by simultaneously replacing each variable occurring in $dom(\sigma)$ by the corresponding term. The resulting term is denoted $t\sigma$.

**Example:**

Let $\sigma = \{X \mapsto g(Y), Y \mapsto a\}$ and $t = f(f(X, g(a)), Y)$. Then

$$t\sigma = f(f(g(Y), g(a)), a)$$
append([],L,L).
append([X|L],Y,[X|Z]) :- append(L,Y,Z).

To solve the query  :- append([0],[1,2],U) we use the second clause.

The substitution
\{X \mapsto 0, L \mapsto [], Y \mapsto [1,2], U \mapsto [0|Z]\}
unifies \( \text{append}([X|L],Y,[X|Z]) \) with the query \( \text{append}([0],[1,2],U) \), and then we have to prove that \( \text{append}([], [1,2], Z) \) holds.
Since we have a fact \( \text{append}([],L,L) \) in the program, it is sufficient to take \( \{Z \mapsto [1,2]\} \).
Thus, \( \{U \mapsto [0,1,2]\} \) is an answer substitution.

This method is based on the Principle of Resolution.
Unification is a key step in the Principle of Resolution.

**History:**
The unification algorithm was first sketched by Jacques Herbrand in his thesis (in 1930).

In 1965 Alan Robinson introduced the Principle of Resolution and gave a unification algorithm.

Around 1974 Robert Kowalski, Alain Colmerauer and Philippe Roussel defined and implemented a logic programming language based on these ideas (Prolog).

The version of the unification algorithm that we present is based on work by Martelli and Montanari (1982).
A unification problem $\mathcal{U}$ is a set of equations between terms containing variables.

$$\{s_1 = t_1, \ldots, s_n = t_n\}$$

A solution to $\mathcal{U}$, also called a unifier, is a substitution $\sigma$ such that when we apply $\sigma$ to all the terms in the equations in $\mathcal{U}$ we obtain syntactical identities: for each equation $s_i = t_i$, the terms $s_i\sigma$ and $t_i\sigma$ coincide.

The most general unifier of $\mathcal{U}$ is a unifier $\sigma$ such that any other unifier $\rho$ is an instance of $\sigma$. 
Martelli and Montanari’s algorithm finds the most general unifier for a unification problem if a solution exists, otherwise it fails, indicating that there are no solutions.

To find the most general unifier for a unification problem, the algorithm simplifies the set of equations until a substitution is generated.

The way equations are simplified is specified by a set of transformation rules, which apply to sets of equations and produce new sets of equations or a failure.
Unification Algorithm

**Input:** A finite set of equations: \( \{ s_1 = t_1, \ldots, s_n = t_n \} \)

**Output:** A substitution (mgu for these terms), or failure.

**Transformation Rules:**
Rules are applied non-deterministically, until no rule can be applied or a failure arises.

1. \( f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n), E \rightarrow s_1 = t_1, \ldots, s_n = t_n, E \)
2. \( f(s_1, \ldots, s_n) = g(t_1, \ldots, t_m), E \rightarrow \text{failure} \)
3. \( X = X, E \rightarrow E \)
4. \( t = X, E \rightarrow X = t, E \quad \text{if } t \text{ is not a variable} \)
5. \( X = t, E \rightarrow X = t, E\{X \mapsto t\} \quad \text{if } X \text{ not in } t \text{ and } X \text{ in } E \)
6. \( X = t, E \rightarrow \text{failure} \quad \text{if } X \text{ in } t \text{ and } X \neq t \)
Remarks

- We are working with sets of equations, therefore their order in the unification problem is not important.
- The test in case (6) is called occur-check, e.g. \(X = f(X)\) fails. This test is time consuming, and for this reason in some systems it is not implemented.
- In case of success, by changing in the final set of equations the “=” by \(\mapsto\) we obtain a substitution, which is the most general unifier (mgu) of the initial set of terms.
- Cases (1) and (2) apply also to constants: in the first case the equation is deleted and in the second there is a failure.
Examples:

In the example with `append`, we solved the unification problem:
\[
\{ [X|L] = [0], \ Y = [1,2], \ [X|Z] = U \}
\]
Recall that the notation `[ | ]` represents a binary list constructor (the arguments are the head and the tail of the list). `[0]` is a shorthand for `[0 | []]`, and `[]` is a constant.

We now apply the unification algorithm to this set of the equations:
using rule (1) in the first equation, we get:
\[
\{ X = 0, \ L = [], \ Y = [1,2], \ [X|Z] = U \}
\]
using rule (5) and the first equation we get:
\[
\{ X = 0, \ L = [], \ Y = [1,2], \ [0|Z] = U \}
\]
using rule (4) and the last equation we get:
\[
\{ X = 0, \ L = [], \ Y = [1,2], \ U = [0|Z] \}
\]
and the algorithm stops.

Therefore the most general unifier is:
\[
\{ X \leftrightarrow 0, \ L \leftrightarrow [], \ Y \leftrightarrow [1,2], \ U \leftrightarrow [0|Z] \}
\]
To implement rewriting, or to implement a functional/logic programming language, we need a matching/unification algorithm.

- For first order terms, there are very efficient algorithms (linear time complexity).
- For terms with binders, we need more powerful algorithms that take into account $\alpha$-equivalence.
- Higher-order unification is undecidable.
- Nominal unification is decidable (polynomial).
- A solvable problem $Pr$ has a unique most general solution.
- Nominal matching is linear.

In this course we will use nominal matching to define nominal rewriting.
Back to nominal terms: checking $\alpha$-equivalence

The syntax-directed derivation rules that define $\alpha$-equivalence of nominal terms suggest an algorithm to check $\alpha$-equivalence, using transformation rules in the style of Martelli and Montanari’s.

\[
\begin{align*}
  a \# b, Pr & \implies Pr \\
  a \# fs, Pr & \implies a \# s, Pr \\
  a \# (s_1, \ldots, s_n), Pr & \implies a \# s_1, \ldots, a \# s_n, Pr \\
  a \# [b]s, Pr & \implies a \# s, Pr \\
  a \# [a]s, Pr & \implies Pr \\
  a \# \pi \cdot X, Pr & \implies \pi^{-1} \cdot a \# X, Pr \quad \pi \neq Id
\end{align*}
\]

\[
\begin{align*}
  a \approx_\alpha a, Pr & \implies Pr \\
  (l_1, \ldots, l_n) \approx_\alpha (s_1, \ldots, s_n), Pr & \implies l_1 \approx_\alpha s_1, \ldots, l_n \approx_\alpha s_n, Pr \\
  fl \approx_\alpha fs, Pr & \implies l \approx_\alpha s, Pr \\
  [a]l \approx_\alpha [a]s, Pr & \implies l \approx_\alpha s, Pr \\
  [b]l \approx_\alpha [a]s, Pr & \implies (a \ b) \cdot l \approx_\alpha s, a \# l, Pr \\
  \pi \cdot X \approx_\alpha \pi' \cdot X, Pr & \implies ds(\pi, \pi') \# X, Pr
\end{align*}
\]
The relation $\rightarrow^*$ is confluent and strongly normalising; i.e. the simplification process terminates and the result is unique: $\langle \text{Pr} \rangle_{nf}$.

$\langle \text{Pr} \rangle_{nf}$ is of the form $\Delta \cup \text{Contr} \cup \text{Eq}$ where:
- $\Delta$ contains consistent freshness constraints ($a \# X$)
- $\text{Contr}$ contains inconsistent freshness constraints ($a \# a$)
- $\text{Eq}$ contains reduced $\approx_\alpha$ constraints.

**Lemma:**
$\Gamma \vdash \text{Pr}$ if and only if $\Gamma \vdash \langle \text{Pr} \rangle_{nf}$.
Let $\langle \text{Pr} \rangle_{nf} = \Delta \cup \text{Contr} \cup \text{Eq}$. Then $\Delta \vdash \text{Pr}$ if and only if $\text{Contr}$ and $\text{Eq}$ are empty.
• Nominal Unification: \( l \equiv t \) has solution \((\Delta, \theta)\) if

\[
\Delta \vdash l\theta \equiv_\alpha t\theta
\]
• Nominal Unification: \( l \approx t \) has solution \( (\Delta, \theta) \) if

\[
\Delta \vdash l\theta \approx_{\alpha} t\theta
\]

• Nominal Matching: \( s = t \) has solution \( (\Delta, \theta) \) if

\[
\Delta \vdash s\theta \approx_{\alpha} t
\]

Examples:

\[
\lambda([a]X) = \lambda([b]X) \approx \lambda([a]X) = \lambda([b]X)
\]

Solutions: \( (\emptyset, \{X \mapsto a\}) \) and \( (\{a\#X, b\#X\}, \text{Id}) \) resp.
• Nominal Unification: \( l \approx t \) has solution \((\Delta, \theta)\) if
\[
\Delta \vdash l\theta \approx_{\alpha} t\theta
\]

• Nominal Matching: \( s = t \) has solution \((\Delta, \theta)\) if
\[
\Delta \vdash s\theta \approx_{\alpha} t
\]

• Examples:
\[
\lambda([a]X) = \lambda([b]b) \quad \text{??}
\]
\[
\lambda([a]X) = \lambda([b]X) \quad \text{??}
\]
• Nominal Unification: \( l \approx t \) has solution \((\Delta, \theta)\) if
\[
\Delta \vdash l\theta \approx_{\alpha} t\theta
\]

• Nominal Matching: \( s = t \) has solution \((\Delta, \theta)\) if
\[
\Delta \vdash s\theta \approx_{\alpha} t
\]

• Examples:
\[
\lambda([a]X) = \lambda([b]b) \quad \lambda([a]X) = \lambda([b]X)
\]

• Solutions: \((\emptyset, [X \mapsto a])\) and \((\{ a\#X, b\#X \}, ld)\) resp.
To define the reduction relation generated by nominal rewriting rules we will use nominal matching.

Recall:
\[ l \approx \alpha t \]
where
\[ V(l) \cap V(t) = \emptyset \]
has solution \((\Delta, \theta)\) if
\[ \Delta \vdash l \theta \approx \alpha t \]
Nominal matching is decidable [Urban, Pitts, Gabbay 2003]
A solvable problem \(Pr\) has a unique most general solution:
\[ (\Gamma, \theta) \text{ such that } \Gamma \vdash Pr \theta. \]
A nominal matching algorithm can be obtained by adding an instantiation rule:
\[ \pi \cdot X \approx \alpha u, Pr = \Rightarrow X \mapsto \pi^{-1} \cdot u \quad Pr[ X \mapsto \pi^{-1} \cdot u ] \]
No occur-checks needed (left-hand side variables distinct from right-hand side variables).
• To define the reduction relation generated by nominal rewriting rules we will use nominal matching.

• Recall:
  \[ l \approx_{\alpha} t \text{ where } V(l) \cap V(t) = \emptyset \text{ has solution } (\Delta, \theta) \text{ if } \]

  \[ \Delta \vdash l\theta \approx_{\alpha} t \]
Nominal Matching

- To define the reduction relation generated by nominal rewriting rules we will use nominal matching.
- Recall:
  \[ l \approx_\alpha t \text{ where } V(l) \cap V(t) = \emptyset \text{ has solution } (\Delta, \theta) \text{ if} \]
  \[ \Delta \vdash l\theta \approx_\alpha t \]
- Nominal matching is decidable [Urban, Pitts, Gabbay 2003]
Nominal Matching

- To define the reduction relation generated by nominal rewriting rules we will use nominal matching.
- Recall: \( l \approx_\alpha t \) where \( V(l) \cap V(t) = \emptyset \) has solution \((\Delta, \theta)\) if

\[
\Delta \vdash l \theta \approx_\alpha t
\]

- Nominal matching is decidable [Urban, Pitts, Gabbay 2003]
- A solvable problem \( Pr \) has a unique most general solution: \((\Gamma, \theta)\) such that \( \Gamma \vdash Pr \theta \).
• To define the reduction relation generated by nominal rewriting rules we will use nominal matching.

• Recall:

\[ l \approx_\alpha t \text{ where } V(l) \cap V(t) = \emptyset \text{ has solution } (\Delta, \theta) \text{ if} \]

\[ \Delta \vdash l \theta \approx_\alpha t \]

• Nominal matching is decidable [Urban, Pitts, Gabbay 2003]

• A solvable problem \( Pr \) has a unique most general solution: \((\Gamma, \theta)\) such that \( \Gamma \vdash Pr\theta \).

• A nominal matching algorithm can be obtained by adding an instantiation rule to the previous set:

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

No occur-checks needed (left-hand side variables distinct from right-hand side variables).
Nominal Rewriting Rules:

\[ \Delta \vdash l \rightarrow r \quad V(r) \cup V(\Delta) \subseteq V(l) \]

Examples:

\[
\begin{align*}
(\lambda[a]X)Y &\rightarrow X[a \mapsto Y] \\
a[a \mapsto Y] &\rightarrow Y \\
(XX')[a \mapsto Y] &\rightarrow X[a \mapsto Y]X'[a \mapsto Y] \\
a \# Y &\vdash Y[a \mapsto X] \rightarrow Y \\
b \# Y &\vdash (\lambda[b]X)[a \mapsto Y] \rightarrow \lambda[b](X[a \mapsto Y])
\end{align*}
\]

Equivariance: Rules are defined modulo permutative renamings of atoms.

Next questions:
What is the complexity of Nominal Matching?
Is nominal matching sufficient (complete) for nominal rewriting?
The transformation rules create permutations. In polynomial implementations of nominal unification permutations are lazy: only pushed down a term when needed.
The transformation rules create permutations. In polynomial implementations of nominal unification permutations are lazy: only pushed down a term when needed.

Problem: lazy permutations may grow (they accumulate).
• The transformation rules create permutations. In polynomial implementations of nominal unification permutations are lazy: only pushed down a term when needed.
• Problem: lazy permutations may grow (they accumulate).
• To obtain an efficient algorithm, work with a single *current* permutation, represented by an *environment*. 
An environment $\xi$ is a pair $(\xi_\pi, \xi_A)$ of a permutation and a set of atoms.

Notation: $s \approx_\alpha \xi \diamond t$ represents $s \approx_\alpha \xi_\pi \cdot t$, $\xi_A \not\approx t$.

An environment problem $Pr$ is either $\bot$ or $s_1 \approx_\alpha \xi_1 \diamond t_1, \ldots, s_n \approx_\alpha \xi_n \diamond t_n$.

It is easy to translate a standard problem into an environment problem and vice-versa.
The algorithms to check \( \alpha \)-equivalence constraints and to solve matching problems are modular. The core module is common to both algorithms, and has four phases:

Phase 1 reduces environment constraints, by propagating \( \xi_i \) over \( t_i \).
Phase 2 eliminates permutations on the left-hand side.
Phase 3 reduces freshness constraints.
Phase 4 computes the standard form of the resulting problem.

\( \overline{Pr}^c \) denotes the result of applying the core algorithm on \( Pr \).
Phase 1 - Input: $Pr = (s_i \approx_{\alpha} \xi_i \diamond t_i)_{i}^n$

$$Pr, \quad a \quad \approx_{\alpha} \xi \diamond t \quad \Rightarrow \begin{cases} Pr \quad \text{if } a = \xi_{\pi} \cdot t \text{ and } t \not\in \xi_A \\ \bot \quad \text{otherwise} \end{cases}$$

$$Pr, \quad (s_1, \ldots, s_n) \approx_{\alpha} \xi \diamond t \quad \Rightarrow \begin{cases} Pr, \quad (s_i \approx_{\alpha} \xi \diamond u_i)_{i}^n \quad \text{if } t = (u_1, \ldots, u_n) \\ \bot \quad \text{otherwise} \end{cases}$$

$$Pr, \quad f \ s \quad \approx_{\alpha} \xi \diamond t \quad \Rightarrow \begin{cases} Pr, \quad s \approx_{\alpha} \xi \diamond u \quad \text{if } t = f \ u \\ \bot \quad \text{otherwise} \end{cases}$$

$$Pr, \quad [a]s \quad \approx_{\alpha} \xi \diamond t \quad \Rightarrow \begin{cases} Pr, \quad s \approx_{\alpha} \xi' \diamond u \quad \text{if } t = [b]u \\ \bot \quad \text{otherwise} \end{cases}$$

where $\xi' = (((a \ \xi_{\pi} \cdot b) \circ \xi_{\pi}, (\xi_A \cup \{\xi_{\pi}^{-1} \cdot a\}) \setminus \{b\}) \text{ in the last rule,}$

and $a, b$ could be the same atom.

The normal forms for phase 1 rules are either $\bot$ or $(\pi_i \cdot X_i \approx_{\alpha} \xi_i \diamond s_i)_{i}^n$ where $s_i$ are nominal terms.
Phase 2 - Input: A Phase 1 normal form.

\[ \pi \cdot X \approx_\alpha \xi \diamond t \implies X \approx_\alpha (\pi^{-1} \cdot \xi) \diamond t \quad (\pi \neq Id) \]

where \( \pi^{-1} \cdot \xi = (\pi^{-1} \circ \xi_\pi, \xi_A) \).

Above, \( \pi^{-1} \) applies only to \( \xi_\pi \), because \( \pi \cdot X \approx_\alpha \xi \diamond t \) represents \( \pi \cdot X \approx_\alpha \xi_\pi \cdot t, \xi_A \# t \).

Phase 2 normal forms are either \( \bot \) or \((X_i \approx_\alpha \xi_i \diamond t_i)_1^n\), where the terms \( t_i \) are standard nominal terms.
Phase 3 - Input: A Phase 2 normal form \((X_i \approx_\alpha \xi_i \diamond t_i)_1^n\).

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

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

The normal forms are either \(\bot\) or \((X_i \approx_\alpha t_i)_1^n\) where \(t_i \in T_\xi\).

\[
T_\xi = a \mid f \ T_\xi \mid (T_\xi, \ldots, T_\xi) \mid [a] T_\xi \mid \xi \diamond X
\]
Phase 4:

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

Normal forms are either \( \bot \) or \((X_i \approx_\alpha u_i)_{i \in I}, (A_j \# X_j)_{j \in J}\) where \(u_i\) are nominal terms and \(I, J\) may be empty.

**Correctness:**
The core algorithm terminates, and preserves the set of solutions.
To check that a set $Pr$ of $\alpha$-equivalence constraints is valid:

- Run the core algorithm on $Pr$
To check that a set $Pr$ of $\alpha$-equivalence constraints is valid:

- Run the core algorithm on $Pr$
- If left-hand sides of $\approx_\alpha$-constraints in $Pr$ are ground, stop
  otherwise reduce the result $Pr^c$ using:

\[
(\alpha) \quad Pr, X \approx_\alpha t \implies \begin{cases} 
    Pr, supp(\pi) \not\# X & \text{if } t = \pi \cdot X \\
    \bot & \text{otherwise}
\end{cases}
\]

where $supp(\pi) = \{ a \mid \pi \cdot a \neq a \}$
Checking $\alpha$-equivalence constraints

To check that a set $Pr$ of $\alpha$-equivalence constraints is valid:

- Run the core algorithm on $Pr$
- If left-hand sides of $\approx_\alpha$-constraints in $Pr$ are ground, stop otherwise reduce the result $\overline{Pr}^c$ using:

$$ (\alpha) \quad Pr, \ X \approx_\alpha t \implies \begin{cases} Pr, \ supp(\pi) \not\# X & \text{if } t = \pi \cdot X \\ \perp & \text{otherwise} \end{cases} $$

where $supp(\pi) = \{ a \mid \pi \cdot a \neq a \}$

- Normal forms: $\perp$ or $(A_i \not\# X_i)_1^n$. 

Maribel Fernández  Nominal Rewriting
Checking $\alpha$-equivalence constraints

To check that a set $Pr$ of $\alpha$-equivalence constraints is valid:

- Run the core algorithm on $Pr$
- If left-hand sides of $\approx_\alpha$-constraints in $Pr$ are ground, stop otherwise reduce the result $Pr^c$ using:

$$ (\alpha) \quad Pr, \ X \approx_\alpha t \implies \begin{cases} Pr, \ supp(\pi) \not\equiv X & \text{if } t = \pi \cdot X \\ \bot & \text{otherwise} \end{cases} $$

where $supp(\pi) = \{a | \pi \cdot a \neq a\}$

- Normal forms: $\bot$ or $(A_i \not\equiv X_i)_1^n$.
- Correctness: If the normal form is $\bot$ then $Pr$ is not valid. If the normal form of $Pr$ is $(A_i \not\equiv X_i)_1^n$ then $(A_i \not\equiv X_i)_1^n \vdash Pr$. 
To solve a matching problem $Pr$:

- Run the core algorithm on $Pr$
To solve a matching problem \( Pr \): 

- Run the core algorithm on \( Pr \)
- If the problem is non-linear, normalise the result \( \overline{Pr}^c \) by:
  \[
  \begin{cases}
  Pr, X \approx_\alpha s, \ X \approx_\alpha t \implies Pr, X \approx_\alpha s, s \approx_\alpha t \approx_\alpha & \text{if } s \approx_\alpha t \approx_\alpha \neq \bot \\
  \bot & \text{otherwise}
  \end{cases}
  \]
To solve a matching problem $Pr$:

- Run the core algorithm on $Pr$
- If the problem is non-linear, normalise the result $\overline{Pr}^c$ by:
  \[
  Pr, X \approx_\alpha s, X \approx_\alpha t \implies
  \begin{cases}
  Pr, X \approx_\alpha s, s \approx_\alpha t \approx_\alpha & \text{if } s \approx_\alpha t \approx_\alpha \neq \bot \\
  \bot & \text{otherwise}
  \end{cases}
  \]
- Normal forms: $\bot$ or a pair of a substitution and a freshness context.
Solving Matching Problems

To solve a matching problem $Pr$:

- Run the core algorithm on $Pr$
- If the problem is non-linear, normalise the result $\overline{Pr}^c$ by:
  \[
  Pr, X \approx_\alpha s, \ X \approx_\alpha t \implies \begin{cases} \Pr, X \approx_\alpha s, s \approx_\alpha t \approx_\alpha & \text{if } s \approx_\alpha t \approx_\alpha \neq \bot \\ \bot & \text{otherwise} \end{cases}
  \]
- Normal forms: $\bot$ or a pair of a substitution and a freshness context.
- Correctness:
The result is a most general solution of the matching problem $Pr$. 

Solving Matching Problems

To solve a matching problem $Pr$:

- Run the core algorithm on $Pr$
- If the problem is non-linear, normalise the result $\overline{Pr}^c$ by:
  $\overline{Pr}, X \approx_\alpha s, X \approx_\alpha t \implies$
  \[
  \begin{cases} 
  Pr, X \approx_\alpha s, s \approx_\alpha t \approx_\alpha & \text{if } s \approx_\alpha t \approx_\alpha \neq \bot \\
  \bot & \text{otherwise}
  \end{cases}
  \]
- Normal forms: $\bot$ or a pair of a substitution and a freshness context.
- Correctness:
  The result is a most general solution of the matching problem $Pr$.
- Remark:
  If variables occur linearly in patterns then the core algorithm is sufficient.
Core algorithm: linear in the size of the initial problem in the ground case, using mutable arrays. In the non-ground case, log-linear using functional maps.

Alpha-equivalence check: linear if right-hand sides of constraints are ground (core algorithm). Otherwise, log-linear using functional maps.

Matching: quadratic in the non-ground case (traversal of every term in the output of the core algorithm). Worst case complexity: when phase 4 suspends permutations on all variables. If variables in the input problem are 'saturated' with permutations, then linear (permutations cannot grow).
Summary:

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

Remark:
The representation using higher-order abstract syntax does saturate the variables (they have to be applied to the set of atoms they can capture).
Conjecture: the algorithms are linear wrt HOAS also in the non-ground case.
OCAML implementation, available from CANS webpage.
• Nominal matching is efficient.
• Nominal matching is efficient.
• Equivariant nominal matching is exponential... BUT
Nominal Matching vs. Equivariant Matching

- Nominal matching is efficient.
- Equivariant nominal matching is exponential... BUT
- if rules are CLOSED then nominal matching is sufficient. Intuitively, closed means no free atoms. The nominal rewriting system in the example above is closed.
$R \equiv \nabla \vdash l \rightarrow r$ is **closed** when

$$(\nabla' \vdash (l', r')) \approx (\nabla, A(R')\#V(R) \vdash (l, r))$$

has a solution $\sigma$ (where $R'$ is freshened with respect to $R$).

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

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

and call this **closed rewriting**.
The following rules are not closed:

\[ g(a) \rightarrow a \]
\[ [a]X \rightarrow X \]

Why?
The following rule is closed:

\[ a\#X \vdash [a]X \rightarrow X \]

Why?
The following rules are closed, they define capture-avoiding substitution operation of the lambda calculus. We introduce a term-former for (explicit) substitutions $\mathit{subst}$ which we sugar to $t[a\mapsto t'].$

(Beta) \hspace{2cm} (\lambda[a]X)X' \rightarrow X[a\mapsto X']$

$(\sigma_{\text{App}})$ \hspace{2cm} $(XX')[a\mapsto Y] \rightarrow X[a\mapsto Y]X'[a\mapsto Y]$

$(\sigma_{\text{var}})$ \hspace{2cm} $a[a\mapsto X] \rightarrow X$

$(\sigma_{\epsilon}) \hspace{2cm} a \not\# Y \vdash Y[a\mapsto X] \rightarrow Y$

$(\sigma_{f\text{n}})$ \hspace{2cm} $b \not\# Y \vdash (\lambda[b]X)[a\mapsto Y] \rightarrow \lambda[b](X[a\mapsto Y])$
Closed Nominal Rewriting:

- works uniformly in $\alpha$ equivalence classes of terms.
Closed Nominal Rewriting:

- works uniformly in $\alpha$ equivalence classes of terms.
- is expressive: can encode Combinatory Reduction Systems.
Closed Nominal Rewriting:

- works uniformly in $\alpha$ equivalence classes of terms.
- is expressive: can encode Combinatory Reduction Systems.
- is efficient: polynomial equivariant matching.
Closed Nominal Rewriting:

- works uniformly in $\alpha$ equivalence classes of terms.
- is expressive: can encode Combinatory Reduction Systems.
- is efficient: polynomial equivariant matching.
- inherits confluence conditions from first order rewriting.
Confluence

Suppose

1. $R_i = \nabla_i \vdash l_i \rightarrow r_i$ 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. $l_1 \equiv L[l'_1]$ such that $\nabla_1, \nabla_2, l'_1 \approx? l_2$ has a principal solution $(\Gamma, \theta)$, so that $\Gamma \vdash l'_1 \theta \approx_\alpha l_2 \theta$ and $\Gamma \vdash \nabla_i \theta$ for $i = 1, 2$.

Then $\Gamma \vdash (r_1 \theta, L[\theta][r_2 \theta])$ is a critical pair.

If $L = [-]$ 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.

Critical Pair Lemma:
A closed nominal rewrite system where all non-trivial critical pairs are joinable, is locally confluent.

Orthogonality:
If all the rules are closed, left-linear, and without superpositions nominal rewriting is confluent.
So far, we have discussed untyped nominal terms.

There exist many-sorted versions of nominal syntax, and also type assignment systems “à la Curry” for nominal terms.
Types for Nominal Terms

Types built from

- a set of base data sorts $\delta$ (e.g. Nat, Bool, Exp, ...)
- type variables $\alpha$, and
- type constructors $C$ (e.g. $\times$, $\to$, List, ...)

Types and type schemes:

$$\tau ::= \delta \mid \alpha \mid (\tau_1 \times \ldots \times \tau_n) \mid C\;\tau \mid [\tau]\tau' \quad \sigma ::= \forall\!\alpha.\tau$$

Type declarations (arity):

$$\rho ::= (\tau')\tau$$

E.g. $\text{succ}: (\text{Nat})\text{Nat}$

Instantiation: $\sigma \leq \tau$ E.g. $\forall\alpha.\alpha \leq \text{Nat}$, $(\alpha)\alpha \leq (\text{Nat})\text{Nat}$
Typing judgement: $\Gamma; \Delta \vdash s : \tau$ where $\Gamma$ is a typing context, $\Delta$ a freshness context, $s$ a term and $\tau$ a type.

\[
\frac{\sigma \leq \tau}{\Gamma, a : \sigma; \Delta \vdash a : \tau}
\]

\[
\frac{\sigma \leq \tau}{\Gamma; \Delta \vdash \pi \cdot X : \Diamond}
\]

\[
\frac{\Gamma, X : \sigma; \Delta \vdash \pi \cdot X : \tau}{\Gamma; \Delta \vdash a : \tau}
\]

\[
\frac{\Gamma, a : \tau; \Delta \vdash t : \tau'}{\Gamma; \Delta \vdash [a]t : [\tau]\tau'}
\]

\[
\frac{\Gamma; \Delta \vdash t_i : \tau_i \ (1 \leq i \leq n)}{\Gamma; \Delta \vdash (t_1, \ldots, t_n) : \tau_1 \times \ldots \times \tau_n}
\]

\[
\frac{\Gamma; \Delta \vdash t : \tau' \quad f : \rho \leq (\tau')\tau}{\Gamma; \Delta \vdash ft : \tau}
\]

$\Gamma; \Delta \vdash \pi \cdot X : \Diamond$ holds if for any $a$ such that $\pi \cdot a \neq a$, $\Delta \vdash a \# X$ or for some $\sigma$, $a : \sigma, \pi \cdot a : \sigma \in \Gamma$. 
Examples

\[
\begin{align*}
  a : \forall \alpha.\alpha, X : \beta & \vdash (a, X) : \beta \times \beta \\
  & \vdash [a]a : [\alpha]\alpha \\
  a : \beta & \vdash [a]a : [\alpha]\alpha \\
  a : \alpha, b : \alpha, X : \tau & \vdash (a \ b) \cdot X : \tau \\
  X : \tau; a \# X, b \# X & \vdash (a \ b) \cdot X : \tau \\
  X : \tau, a : \alpha, b : \alpha & \vdash [a]((a \ b) \cdot X, b) : [\alpha](\tau \times \alpha)
\end{align*}
\]

Generalisation of Hindley-Milner’s type system:

- atoms (can be abstracted or unabstracted),
- variables (cannot be abstracted but can be instantiated, with non-capture-avoiding substitutions),
- suspended permutations.
Principal Types

- Every term has a principal type, and type inference is decidable.
- Principal types are obtained using a function $pt(\Gamma; \Delta \vdash s)$. $pt$ is sound and complete.
  - $pt(\Gamma, a : \forall \alpha \tau; \Delta \vdash a) = (Id, \tau)$, where $\alpha \in \overline{\alpha}$ are fresh.
  - $pt(\Gamma, X : \forall \alpha \tau; \Delta \vdash \pi \cdot X) = (S, \tau S)$ ($\alpha \in \overline{\alpha}$ fresh) provided that for each $a$ in $\pi$ such that $a \neq \pi \cdot a$, $\Delta \vdash a \neq X$, or $a : \sigma, \pi \cdot a : \sigma' \in \Gamma$ for some $\sigma, \sigma'$ that are unifiable. $S$ is the mgu of those pairs, or $Id$ if all such $a$ are fresh for $X$.
  - $pt(\Gamma; \Delta \vdash (t_1, \ldots, t_n)) = (S_1 \ldots S_n, \phi_1 S_2 \ldots S_n \times \ldots \times \phi_{n-1} S_n \times \phi_n)$ where $pt(\Gamma; \Delta \vdash t_1) = (S_1, \phi_1), pt(\Gamma S_1; \Delta \vdash t_2) = (S_2, \phi_2), \ldots, pt(\Gamma S_1 \ldots S_{n-1}; \Delta \vdash t_n) = (S_n, \phi_n)$.
  - $pt(\Gamma; \Delta \vdash ft) = (SS', \phi S')$ where $pt(\Gamma; \Delta \vdash t) = (S, \tau), f : \rho = (\phi')\phi$ (type variables in $\rho$ are fresh), $S' = mgu(\tau, \phi')$.
  - $pt(\Gamma; \Delta \vdash [a]s) = (S|_{\Gamma}, [\tau']_{\tau})$ where $pt(\Gamma, a : \alpha; \Delta \vdash s) = (S, \tau)$, $\alpha$ is fresh, $\alpha S = \tau'$.
\(\alpha\)-equivalence preserves types:

\[
\Delta \vdash s \approx_{\alpha} t \text{ and } \Gamma; \Delta \vdash s : \tau \text{ imply } \Gamma; \Delta \vdash t : \tau.
\]
• Nominal Terms: first-order syntax with binders.
• Nominal unification is polynomial (unknown lower bound).
• Nominal unification is used in the language $\alpha$-Prolog [Cheney and Urban]
• Nominal matching is linear, equivariant matching is linear with closed rules.
• Variants of nominal matching are used in functional languages with nominal features (eg. FreshML).
Conclusion

- NRSs are first-order systems with built-in $\alpha$-equivalence: first-order substitutions, matching modulo $\alpha$.
- Closed NRSs have the expressive power of higher-order rewriting. Higher-order substitutions are easy to define using freshness.
- Closed NRSs have the properties of first-order rewriting (critical pair lemma, orthogonality).
- Types can be added to give semantics to terms and to obtain sufficient conditions for termination.
- Hindley-Milner style types: Typing is decidable and there are principal types, $\alpha$-equivalence preserves types.