Nominal Techniques

Maribel Fernández
King's College London

Scottish Programming Languages and Verification
Summer School 2020

Nominal Techniques: Syllabus

Introduction

- First-order languages
- Languages with binding operators

Specifying binders:

- α-equivalence
- Nominal syntax
- Nominal unification (unification modulo α-equivalence)
- Nominal matching (matching modulo α-equivalence)

Nominal rewriting

- Extending first-order rewriting to specify binding operators
- Closed rewriting
- Confluence
- Typed Rewriting Systems

Equational Axioms: AC operators

Further reading

- C. Urban, A. Pitts, M.J. Gabbay. *Nominal Unification*. Theoretical Computer Science 323, pages 473-497, 2004.
- C. Calvès, M. Fernández. *Matching and Alpha-Equivalence Check for Nominal Terms*. J. of Computer and System Sciences, 2010.
- M. Fernández, M.J. Gabbay. *Nominal Rewriting*. Information and Computation 205, pages 917-965, 2007.
- E. Fairweather, M. Fernández. *Typed Nominal Rewriting.* ACM Transactions on Computational Logic, 2018.
- J. Dominguez, M. Fernández. *Nominal Syntax with Atom Substitution: Matching, Unification, Rewriting.* Proceedings of FCT 2019, LNCS, Springer.
- M. Ayala-Rincón, W. Carvalho-Segundo, M. Fernández, D. Nantes, A. Oliveira. *A Formalisation of Nominal Alpha-Equivalence with A, C and AC Function Symbols.* Theoretical Computer Science, 2019.
- M. Ayala-Rincón, M. Fernández, D. Nantes. *On nominal syntax and permutation fixed points.* Logical Methods in Computer Science, Volume 16, Issue 1, 2020.

First-order languages vs. languages with binders

Most programming languages support first-order data structures and first-order operators.

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

$$\begin{array}{lcl} append(nil, x) & \to & x \\ append(cons(x, z), y) & \to & cons(x, append(z, y)) \end{array}$$

Very few programming languages support data structures with binding constructs.

However, we often need to manipulate data with bound names.
Example: compilers, type checkers, code optimisation, etc.

Binding operators: Examples

Some concrete examples of binding constructs (informally):

- Operational semantics:

$$\text{let } a = N \text{ in } M \longrightarrow (\text{fun } a.M)N$$

Binding operators: Examples

Some concrete examples of binding constructs (informally):

- Operational semantics:

$$\text{let } a = N \text{ in } M \longrightarrow (\text{fun } a.M)N$$

- β and η-reductions in the λ-calculus:

$$\begin{array}{lcll} (\lambda x.M)N & \to & M[x/N] & \\ (\lambda x.Mx) & \to & M & (x \notin \text{fv}(M)) \end{array}$$

Binding operators: Examples

Some concrete examples of binding constructs (informally):

- Operational semantics:

$$\text{let } a = N \text{ in } M \longrightarrow (\text{fun } a.M)N$$

- β and η-reductions in the λ-calculus:

$$\begin{array}{lcll} (\lambda x.M)N & \to & M[x/N] & \\ (\lambda x.Mx) & \to & M & (x \notin \text{fv}(M)) \end{array}$$

- π-calculus:

$$P \mid \nu a.Q \to \nu a.(P \mid Q) \qquad (a \notin \text{fv}(P))$$

Binding operators: Examples

Some concrete examples of binding constructs (informally):

- Operational semantics:

$$\text{let } a = N \text{ in } M \longrightarrow (\text{fun } a.M)N$$

- β and η-reductions in the λ-calculus:

$$\begin{array}{lcll} (\lambda x.M)N & \to & M[x/N] & \\ (\lambda x.Mx) & \to & M & (x \notin \text{fv}(M)) \end{array}$$

- π-calculus:

$$P \mid \nu a.Q \to \nu a.(P \mid Q) \qquad (a \notin \text{fv}(P))$$

- Logic equivalences:

$$P \textit{ and } (\forall x.Q) \Leftrightarrow \forall x(P \textit{ and } Q) \qquad (x \notin \text{fv}(P))$$

Binding operators - α-equivalence

Terms are defined modulo renaming of bound variables, i.e., α-equivalence.

Example:
In $\forall x.P$ the variable x can be renamed (avoiding name capture)

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

How can we formally define (or program) binding operators?
There are several alternatives.

First-order frameworks

We can encode α-equivalence in a first-order specification or programming language.

$\Rightarrow$ Example: λ-calculus using De Bruijn's indices with "lift" and "shift" operators to encode non-capturing substitution

First-order frameworks

We can encode α-equivalence in a first-order specification or programming language.

- Example: λ-calculus using De Bruijn's indices with "lift" and "shift" operators to encode non-capturing substitution

$\Rightarrow$ Simple notion of substitution (first-order) $(+)$

First-order frameworks

We can encode α-equivalence in a first-order specification or programming language.

- Example: λ-calculus using De Bruijn's indices with "lift" and "shift" operators to encode non-capturing substitution
- Simple notion of substitution (first-order) (+)

$\Rightarrow$ Efficient matching and unification algorithms (+)

First-order frameworks

We can encode α-equivalence in a first-order specification or programming language.

- Example: λ-calculus using De Bruijn's indices with "lift" and "shift" operators to encode non-capturing substitution
- Simple notion of substitution (first-order) (+)
- Efficient matching and unification algorithms (+)

$\Rightarrow$ No binders (-)

First-order frameworks

We can encode α-equivalence in a first-order specification or programming language.

- Example: λ-calculus using De Bruijn's indices with "lift" and "shift" operators to encode non-capturing substitution
- Simple notion of substitution (first-order) (+)
- Efficient matching and unification algorithms (+)
- No binders (-)

$\Rightarrow$ We need to 'implement' α-equivalence and non-capturing substitution from scratch (-)

First-order frameworks

We can encode α-equivalence in a first-order specification or programming language.

- Example: λ-calculus using De Bruijn's indices with "lift" and "shift" operators to encode non-capturing substitution
- Simple notion of substitution (first-order) $(+)$
- Efficient matching and unification algorithms $(+)$
- No binders (-)
- We need to 'implement' α-equivalence and non-capturing substitution from scratch (-)

$\Rightarrow$ Not user-friendly (-)

Higher-order frameworks

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

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

  One step of rewriting:

  $$app(lam([a]f(a, g(a)), b) \to f(b, g(b))$$

  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 α-equivalence.
  Example: β-rule

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

  One step of rewriting:

  $$app(lam([a]f(a, g(a)), b) \to f(b, g(b))$$

  using (a restriction of) *higher-order matching*.
- Logical frameworks based on Higher-Order Abstract Syntax also work modulo α-equivalence.

$$\text{let } a = N \text{ in } M(a) \ \longrightarrow\ (\text{fun } a \to M(a))N$$

Higher-order frameworks

$\Rightarrow$ The syntax includes binders (+)

Higher-order frameworks

- The syntax includes binders (+)

⇒ Implicit α-equivalence (+)

Higher-order frameworks

- The syntax includes binders (+)
- Implicit α-equivalence (+)

$\Rightarrow$ We targeted α but now we have to deal with β too (-)

Higher-order frameworks

- The syntax includes binders $(+)$
- Implicit α-equivalence $(+)$
- We targeted α but now we have to deal with β too (-)

$\Rightarrow$ Substitution is a meta-operation using β (-)

Higher-order frameworks

- 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 (-)

$\Rightarrow$ Unification is undecidable in general (-)

Higher-order frameworks

- 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 (-)

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

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

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

Nominal Approach [Gabbay-Pitts]

Key ideas:
Freshness conditions $a\#t$,
name swapping $(a\ b) \cdot t$.

Example

β and η rules as nominal rewriting rules:

$$\begin{array}{rcl} & app(lam([a]Z), Z') & \to & subst([a]Z, Z') \\ a\#M \vdash & (\lambda([a]app(M, a)) & \to & M \end{array}$$

$\Rightarrow$ Terms with binders

Nominal Approach [Gabbay-Pitts]

Key ideas:
Freshness conditions $a\#t$,
name swapping $(a\ b) \cdot t$.

Example

β and η rules as nominal rewriting rules:

$$\begin{array}{rlcl} & app(lam([a]Z), Z') & \to & subst([a]Z, Z') \\ a\#M \vdash & (\lambda([a]app(M, a)) & \to & M \end{array}$$

- Terms with binders

$\Rightarrow$ Built-in α-equivalence

Nominal Approach [Gabbay-Pitts]

Key ideas:
Freshness conditions $a\#t$,
name swapping $(a\ b) \cdot t$.

Example

β and η rules as nominal rewriting rules:

$$\begin{array}{rlcl} & app(lam([a]Z), Z') & \to & subst([a]Z, Z') \\ a\#M \vdash & (\lambda([a]app(M, a)) & \to & M \end{array}$$

- Terms with binders
- Built-in α-equivalence

$\Rightarrow$ Simple notion of substitution (first order)

Nominal Approach [Gabbay-Pitts]

Key ideas:
Freshness conditions $a\#t$,
name swapping $(a\ b) \cdot t$.

Example

β and η rules as nominal rewriting rules:

$$\begin{array}{rcl} & app(lam([a]Z), Z') & \rightarrow & subst([a]Z, Z') \\ a\#M \vdash & (\lambda([a]app(M, a)) & \rightarrow & M \end{array}$$

- Terms with binders
- Built-in α-equivalence
- Simple notion of substitution (first order)

$\Rightarrow$ Efficient matching and unification algorithms

Nominal Approach [Gabbay-Pitts]

Key ideas:
Freshness conditions $a\#t$,
name swapping $(a\ b) \cdot t$.

Example

β and η rules as nominal rewriting rules:

$$\begin{array}{rcl} & app(lam([a]Z), Z') & \rightarrow & subst([a]Z, Z') \\ a\#M \vdash & (\lambda([a]app(M, a)) & \rightarrow & M \end{array}$$

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

$\Rightarrow$ Dependencies of terms on names are implicit

Nominal Approach [Gabbay-Pitts]

Key ideas:
Freshness conditions $a\#t$,
name swapping $(a\ b) \cdot t$.

Example

β and η rules as nominal rewriting rules:

$$\begin{array}{rcl} & app(lam([a]Z), Z') & \rightarrow & subst([a]Z, Z') \\ a\#M \vdash & (\lambda([a]app(M, a)) & \rightarrow & M \end{array}$$

- Terms with binders
- Built-in α-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 $a \notin \mathrm{fv}(M)$

Nominal Syntax [Urban, Pitts, Gabbay 2004]

- Variables: $M, N, X, Y, \ldots$
  Atoms: $a, b, \ldots$
  Function symbols (term formers): $f, g \ldots$

Nominal Syntax [Urban, Pitts, Gabbay 2004]

- Variables: $M, N, X, Y, \ldots$
  Atoms: $a, b, \ldots$
  Function symbols (term formers): $f, g \ldots$
- Nominal Terms:

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

  π is a permutation: finite bijection on names, represented as a list of swappings, e.g., $(a\ b)(c\ d)$, Id (empty list).
  $Id \cdot X$ written as X.
  π acts on t (notation $\pi \cdot t$): permutes names in t, suspends on variables.
  $(a\ b) \cdot a = b$, $(a\ b) \cdot b = a$, $(a\ b) \cdot c = c$

Nominal Syntax [Urban, Pitts, Gabbay 2004]

- Variables: $M, N, X, Y, \ldots$
  Atoms: $a, b, \ldots$
  Function symbols (term formers): $f, g \ldots$
- Nominal Terms:

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

  π is a permutation: finite bijection on names, represented as a list of swappings, e.g., $(a\ b)(c\ d)$, Id (empty list).
  $Id \cdot X$ written as X.
  π acts on t (notation $\pi \cdot t$): permutes names in t, suspends on variables.
  $(a\ b) \cdot a = b$, $(a\ b) \cdot b = a$, $(a\ b) \cdot c = c$
- Example (ML): $var(a)$, $app(t, t')$, $lam([a]t)$, $let(t, [a]t')$, $letrec[f]([a]t, t')$, $subst([a]t, t')$
  Syntactic sugar:
  a, (tt'), $\lambda a.t$, let $a = t$ in t', letrec $fa = t$ in t', $t[a \mapsto t']$

α-equivalence

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

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

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

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

where

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

- $a\#X, b\#X \vdash (a\ b)\cdot X \approx_\alpha X$

α-equivalence

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

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

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

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

where

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

- $a\#X, b\#X \vdash (a\ b)\cdot X \approx_\alpha X$
- $b\#X \vdash \lambda[a]X \approx_\alpha \lambda[b](a\ b)\cdot X$

Freshness

Also defined by induction:

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

$$\frac{a\#s_1 \quad \cdots \quad a\#s_n}{a\#(s_1,\ldots,s_n)} \qquad \frac{a\#s}{a\#fs} \qquad \frac{a\#s}{a\#[b]s}$$

Exercises

Are the following judgements valid? Justify your answer by giving a derivation or a counterexample.

$$
\begin{array}{rcrcl}
 & \vdash & \lambda[x]x & \approx_\alpha & \lambda[y]y \\
 & \vdash & \lambda[x]\lambda[y]x & \approx_\alpha & \lambda[y]\lambda[x]y \\
 & \vdash & \lambda[x]X & \approx_\alpha & \lambda[y]Y \\
 & \vdash & \lambda[x]X & \approx_\alpha & \lambda[y]X \\
x\#X & \vdash & \lambda[x]X & \approx_\alpha & \lambda[y]X \\
x\#X, y\#X & \vdash & \lambda[x]s(X) & \approx_\alpha & \lambda[y]s(X) \\
x\#X, y\#X & \vdash & \lambda[x]+(X,Y) & \approx_\alpha & \lambda[y]+(X,(x\ y)\cdot Y) \\
x\#X, y\#X & \vdash & \lambda[x]app(X,\lambda[y]y) & \approx_\alpha & \lambda[y]app(X,\lambda[y]y)
\end{array}
$$

Computing 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
- ...

Nominal Rewriting Rules:

$$\Delta \vdash l \to r \qquad V(r) \cup V(\Delta) \subseteq V(l)$$

Example: Prenex Normal Forms

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

Nominal Rewriting

$$\Delta \vdash s \xrightarrow{R} t$$

s **rewrites with** $R = \nabla \vdash l \to r$ **to** t **in the context** Δ if

1. $s \equiv C[s']$ such that θ solves $(\nabla \vdash l)\ {}_?{\approx}\ (\Delta \vdash s')$
2. $\Delta \vdash C[r\theta] \approx_\alpha t$.

Example

Beta-reduction in the Lambda-calculus:

$Beta$		$(\lambda[a]X)Y$	$\to$	$X[a{\mapsto}Y]$
σ_a		$a[a{\mapsto}Y]$	$\to$	Y
σ_{app}		$(XX')[a{\mapsto}Y]$	$\to$	$X[a{\mapsto}Y]X'[a{\mapsto}Y]$
σ_ϵ	$a\#Y \vdash$	$Y[a{\mapsto}X]$	$\to$	Y
σ_λ	$b\#Y \vdash$	$(\lambda[b]X)[a{\mapsto}Y]$	$\to$	$\lambda[b](X[a{\mapsto}Y])$

Rewriting steps: $(\lambda[c]c)Z \to c[c{\mapsto}Z] \to Z$

Computing with Nominal Terms - Unification/Matching

To implement rewriting, or to implement a functional/logic programming language, we need a matching/unification algorithm. Recall:

- 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 α-equivalence.
- Higher-order unification is undecidable.

Nominal terms have good computational properties:
Nominal unification is decidable and unitary.
Efficient algorithms to check α-equivalence, matching, unification.
$\Longrightarrow$ Nominal programming languages (Alpha-Prolog, FreshML)
$\Longrightarrow$ Nominal Rewriting.

Unification is a popular research field
Started in 1930s with Herbrand thesis
Key for logic programming languages and theorem provers:
Unification algorithms play a central role in the implementation of resolution — *Prolog*.
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, ...
- function symbols $(f, g, h, \ldots)$ with fixed arities

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

A substitution σ 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

$$t\sigma = f(f(g(Y), g(a)), a)$$

Solving Queries in Prolog - Example

```
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 `append([X|L],Y,[X|Z])` with the query
`append([0],[1,2],U)`, and then we have to prove that
`append([],[1,2],Z)` holds.
Since we have a fact `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

A unification problem $\mathcal{U}$ is a set of equations between terms with variables

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

A solution to $\mathcal{U}$, also called a *unifier*, is a substitution σ such that for each equation $s_i = t_i \in \mathcal{U}$, the terms $s_i\sigma$ and $t_i\sigma$ coincide.
The most general unifier of $\mathcal{U}$ is a unifier σ such that any other unifier ρ is an instance of σ.

Unification Algorithm

Martelli and Montanari's algorithm finds the most general unifier for a unification problem (if a solution exists, otherwise it fails) by simplification:

It simplifies the unification problem until a substitution is generated.

It is specified as a set of transformation rules, which apply to sets of equations and produce new sets of equations or a failure.

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)	$f(s_1, \ldots, s_n) = g(t_1, \ldots, t_m), E$	$\rightarrow$	*failure*	
(3)	$X = X, E$	$\rightarrow$	E	
(4)	$t = X, E$	$\rightarrow$	$X = t, E$	if t is not a variable
(5)	$X = t, E$	$\rightarrow$	$X = t, E\{X \mapsto t\}$	if X not in t and X in E
(6)	$X = t, E$	$\rightarrow$	*failure*	if X in t 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 $\mapsto$ 0, L $\mapsto$ [],Y $\mapsto$ [1,2], U $\mapsto$ [0|Z]}

Back to nominal terms: checking α-equivalence

Idea:
Turn the α-equivalence derivation rules into simplification rules in the style of Martelli and Montanari.

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

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

The relation $\Longrightarrow$ is confluent and strongly normalising:
the simplification process terminates,
the result is unique: $\langle Pr \rangle_{nf}$

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

Lemma:

- $\Gamma \vdash Pr$ if and only if $\Gamma \vdash \langle Pr \rangle_{nf}$.
- Let $\langle Pr \rangle_{nf} = \Delta \cup Contr \cup Eq$. Then $\Delta \vdash Pr$ if and only if $Contr$ and Eq are empty.

Solving Equations [Urban, Pitts, Gabbay 2003]

- Nominal Unification: $l \; {}_?{\approx}_? \; t$ has solution (Δ, θ) if

$$\Delta \vdash l\theta \approx_\alpha t\theta$$

Solving Equations [Urban, Pitts, Gabbay 2003]

- Nominal Unification: $l \;{}_{?}{\approx}_{?}\; t$ has solution (Δ, θ) if

$$\Delta \vdash l\theta \approx_\alpha t\theta$$

- Nominal Matching: $s = t$ has solution (Δ, θ) if

$$\Delta \vdash s\theta \approx_\alpha t$$

(t ground or variables disjoint from s)

Solving Equations [Urban, Pitts, Gabbay 2003]

- Nominal Unification: $l \;{}_?{\approx}_?\; t$ has solution (Δ, θ) if

$$\Delta \vdash l\theta \approx_\alpha t\theta$$

- Nominal Matching: $s = t$ has solution (Δ, θ) if

$$\Delta \vdash s\theta \approx_\alpha t$$

  (t ground or variables disjoint from s)

- Examples:
  $\lambda([a]X) = \lambda([b]b)$??
  $\lambda([a]X) = \lambda([b]X)$??

Solving Equations [Urban, Pitts, Gabbay 2003]

- Nominal Unification: $l \;{}_?{\approx}_?\; t$ has solution (Δ, θ) if

$$\Delta \vdash l\theta \approx_\alpha t\theta$$

- Nominal Matching: $s = t$ has solution (Δ, θ) if

$$\Delta \vdash s\theta \approx_\alpha t$$

  (t ground or variables disjoint from s)

- Examples:
  $\lambda([a]X) = \lambda([b]b)$??
  $\lambda([a]X) = \lambda([b]X)$??
- Solutions: $(\emptyset, [X \mapsto a])$ and $(\{a\#X, b\#X\}, Id)$ resp.

Back to Nominal Rewriting

Let $R = \nabla \vdash l \to r$ where $V(l) \cap V(s) = \emptyset$

s **rewrites with** R **to** t **in the context** Δ, written $\Delta \vdash s \xrightarrow{R} t$, when:

1. $s \equiv C[s']$ such that θ solves $(\nabla \vdash l) \;{}_?{\approx}\; (\Delta \vdash s')$
2. $\Delta \vdash C[r\theta] \approx_\alpha t$.

- To define the reduction relation generated by nominal rewriting rules we use nominal matching.

Back to Nominal Rewriting

Let $R = \nabla \vdash l \to r$ where $V(l) \cap V(s) = \emptyset$

s **rewrites with** R **to** t **in the context** Δ, written $\Delta \vdash s \xrightarrow{R} t$, when:

1. $s \equiv C[s']$ such that θ solves $(\nabla \vdash l) \; {}_?{\approx} \; (\Delta \vdash s')$
2. $\Delta \vdash C[r\theta] \approx_\alpha t$.

- To define the reduction relation generated by nominal rewriting rules we use nominal matching.
- $(\nabla \vdash l) \; {}_?{\approx} \; (\Delta \vdash s')$ if
  $\nabla, l \approx_\alpha s'$ has solution (Δ', θ), that is, $\Delta' \vdash \nabla\theta, l\theta \approx_\alpha s'$
  and
  $\Delta \vdash \Delta'$

Nominal Matching

- Nominal matching is decidable [Urban, Pitts, Gabbay 2003]
  A solvable problem Pr has a unique most general solution: (Γ, θ) such that $\Gamma \vdash Pr\theta$.
- Nominal matching algorithm: add an *instantiation rule*:

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

  No occur-checks needed (left-hand side variables distinct from right-hand side variables).

Equivariance: Rules defined modulo permutative renamings of atoms.

Beta-reduction in the Lambda-calculus:

$$
\begin{array}{llccc}
\mathit{Beta} & & (\lambda[a]X)Y & \to & X[a \mapsto Y] \\
\sigma_a & & a[a \mapsto Y] & \to & Y \\
\sigma_{app} & & (XX')[a \mapsto Y] & \to & X[a \mapsto Y]X'[a \mapsto Y] \\
\sigma_{\mathsf{f}} & & (\mathsf{f}\ X)[a \mapsto Y] & \to & \mathsf{f}\ (X[a \mapsto Y]) \\
\sigma_\epsilon & a\#Y \vdash & Y[a \mapsto X] & \to & Y \\
\sigma_\lambda & b\#Y \vdash & (\lambda[b]X)[a \mapsto Y] & \to & \lambda[b](X[a \mapsto Y])
\end{array}
$$

Exercises: Are the following rewriting derivations valid? If your answer is positive, indicate the rules and substitutions used in each step.

$$\begin{array}{rlll}
 & \vdash & (\lambda[x]s(x))Y & \to^* \quad s(Y) \\
y\#Y & \vdash & (\lambda[x]\lambda[y]x)Y & \to^* \quad \lambda[y]Y \\
y\#X & \vdash & (\lambda[y]X)Y & \to^* \quad X \\
y\#Y & \vdash & ((\lambda[x]\lambda[y]x)Y)Y & \to^* \quad Y
\end{array}$$

Next questions

- Efficient nominal matching algorithm?
- Is nominal matching sufficient (complete) for nominal rewriting?

A Linear-Time Matching Algorithm

- The transformation rules create permutations.
  In polynomial implementations of nominal unification permutations are lazy: only pushed down a term when needed.

A Linear-Time Matching Algorithm

- 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).

A Linear-Time Matching Algorithm

- 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** ξ is a pair (ξ_π, ξ_A) of a permutation and a set of atoms.

Notation: $s \approx_\alpha \xi \Diamond t$ represents $s \approx_\alpha \xi_\pi \cdot t,\ \xi_A \# t$.

An **environment problem** *Pr* is either $\perp$ 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.

A Linear-Time Algorithm

The algorithms to check α-equivalence constraints and to solve matching problems are modular.

Core module (common to both algorithms) has four phases:
Phase 1 reduces environment constraints, by propagating ξ_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: $Pr = (s_i \approx_\alpha \xi_i \Diamond t_i)_i^n$

$$Pr,\quad a \approx_\alpha \xi \Diamond t \implies \begin{cases} Pr & \text{if } a = \xi_\pi \cdot t \text{ and } t \notin \xi_A \\ \bot & \text{otherwise} \end{cases}$$

$$Pr,\quad (s_1, \ldots, s_n) \approx_\alpha \xi \Diamond t \implies \begin{cases} Pr,\ (s_i \approx_\alpha \xi \Diamond u_i)_1^n & \text{if } t = (u_1, \ldots, u_n) \\ \bot & \text{otherwise} \end{cases}$$

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

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

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

The normal forms for phase 1 rules are either $\bot$ or $(\pi_i \cdot X_i \approx_\alpha \xi_i \Diamond s_i)_1^n$ where s_i are nominal terms.

Phase 2 - Input: A Phase 1 normal form.

$$\pi \cdot X \approx_\alpha \xi \Diamond t \Longrightarrow X \approx_\alpha (\pi^{-1} \cdot \xi) \Diamond t \qquad (\pi \neq Id)$$

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

Above, π^{-1} applies only to ξ_π, 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 $\perp$ 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 $(X_i \approx_\alpha \xi_i \Diamond t_i)_1^n$.

$$
\begin{array}{rcl}
\xi \Diamond a & \Longrightarrow & \begin{cases} \xi_\pi \cdot a & a \notin \xi_A \\ \bot & a \in \xi_A \end{cases} \\
\xi \Diamond f\ t & \Longrightarrow & f\ (\xi \Diamond t) \\
\xi \Diamond (t_1, \ldots, t_j) & \Longrightarrow & (\xi \Diamond t_i)_1^j \\
\xi \Diamond\ [a]s & \Longrightarrow & [\xi_\pi \cdot a]((\xi \setminus \{a\}) \Diamond s) \\
\xi \Diamond (\pi \cdot X) & \Longrightarrow & (\xi \circ \pi) \Diamond X \\
Pr[\bot] & \Longrightarrow & \bot
\end{array}
$$

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:

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

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

Correctness:
The core algorithm terminates, and preserves the set of solutions.

Checking α-equivalence constraints

To check that a set *Pr* of α-equivalence constraints is valid:

- Run the core algorithm on *Pr*

Checking α-equivalence constraints

To check that a set Pr of α-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:

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

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

Checking α-equivalence constraints

To check that a set Pr of α-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:

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

  where $supp(\pi) = \{a \mid \pi \cdot a \neq a\}$
- Normal forms: $\perp$ or $(A_i \# X_i)_1^n$.

Checking α-equivalence constraints

To check that a set Pr of α-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:

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

  where $supp(\pi) = \{a \mid \pi \cdot a \neq a\}$
- Normal forms: $\perp$ or $(A_i \# X_i)_1^n$.
- Correctness: If the normal form is $\perp$ then Pr is not valid. If the normal form of Pr is $(A_i \# X_i)_1^n$ then $(A_i \# X_i)_1^n \vdash Pr$.

Solving Matching Problems

To solve a matching problem Pr:

- Run the core algorithm on 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:
  $Pr, X \approx_\alpha s,\ X \approx_\alpha t \Longrightarrow$
  $$\begin{cases} Pr,\ X \approx_\alpha s, \overline{s \approx_\alpha t}^{\approx_\alpha} & \text{if } \overline{s \approx_\alpha t}^{\approx_\alpha} \neq \perp \\ \perp & \text{otherwise} \end{cases}$$

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:
  $Pr, X \approx_\alpha s,\ X \approx_\alpha t \Longrightarrow$
  $$\begin{cases} Pr,\ X \approx_\alpha s, \overline{s \approx_\alpha t}^{\approx_\alpha} & \text{if } \overline{s \approx_\alpha t}^{\approx_\alpha} \neq \bot \\ \bot & \text{otherwise} \end{cases}$$
- 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:
  $Pr, X \approx_\alpha s,\ X \approx_\alpha t \Longrightarrow$
  $$\begin{cases} Pr,\ X \approx_\alpha s, \overline{s \approx_\alpha t}^{\approx\alpha} & \text{if } \overline{s \approx_\alpha t}^{\approx\alpha} \neq \bot \\ \bot & \text{otherwise} \end{cases}$$
- 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:
  $$Pr, X \approx_\alpha s,\ X \approx_\alpha t \Longrightarrow \begin{cases} Pr,\ X \approx_\alpha s, \overline{s \approx_\alpha t}^{\,\approx_\alpha} & \text{if } \overline{s \approx_\alpha t}^{\,\approx_\alpha} \neq \perp \\ \perp & \text{otherwise} \end{cases}$$
- Normal forms: $\perp$ 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.

Complexity

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.

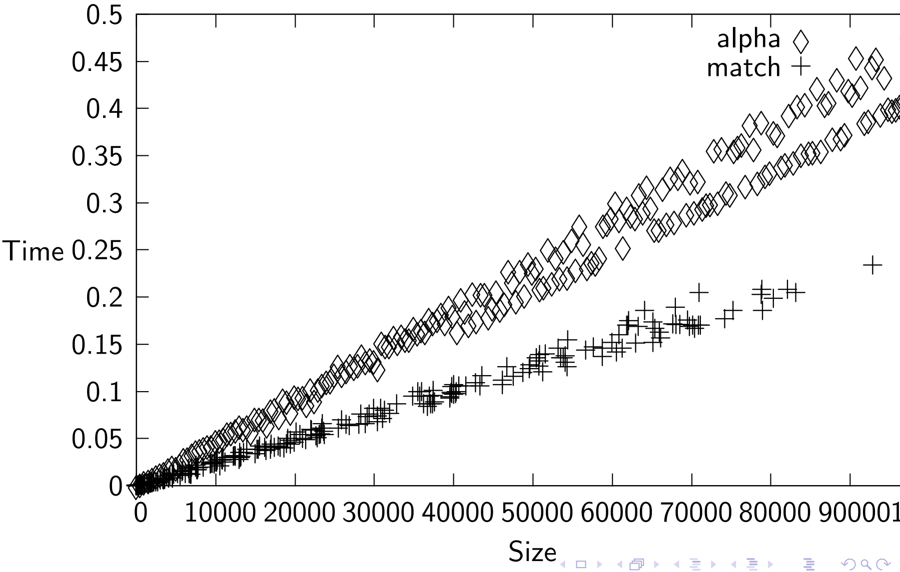
Matching: quadratic in the non-ground case (traversal of every term in the output of the core algorithm).
Worst case complexity: when phase 4 suspends permutations on all variables. If variables in the input problem are 'saturated' with permutations, then linear (permutations cannot grow).

Summary:

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

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:



Nominal Matching vs. Equivariant Matching

- Nominal matching is efficient.

Nominal Matching vs. Equivariant Matching

- 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 rules in the examples above are closed.

Closed Rules

$R \equiv \nabla \vdash l \to r$ is **closed** when

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

has a solution σ (where R' is freshened with respect to R).

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

$$\Delta \vdash s \overset{R}{\to}_c t \quad \text{when} \quad \Delta, A(R')\#V(\Delta, s) \vdash s \overset{R'}{\to} t$$

and call this **closed rewriting**.

Examples

The following rules are not closed:

$$g(a) \to a$$
$$[a]X \to X$$

Why?

Examples

The following rule is closed:

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

Why?

Exercise

Provide a nominal rewriting system defining an explicit substitution operator *subst* of arity 3 for the lambda-calculus.
$subst(x, s, t)$ should return the term obtained by substituting x by t in s.
Are your rules closed?

Examples

Closed rules that define **capture-avoiding substitution** in the lambda calculus:
(explicit) substitutions, $subst([x]M, N)$ abbreviated $M[x \mapsto N]$.

$$
\begin{array}{llcll}
(\text{Beta}) & & (\lambda[a]X)X' & \to & X[a \mapsto X'] \\
(\sigma_{app}) & & (XX')[a \mapsto Y] & \to & X[a \mapsto Y]X'[a \mapsto Y] \\
(\sigma_{a}) & & a[a \mapsto X] & \to & X \\
(\sigma_{\epsilon}) & a \# Y \ \vdash & Y[a \mapsto X] & \to & Y \\
(\sigma_{\lambda}) & b \# Y \ \vdash & (\lambda[b]X)[a \mapsto Y] & \to & \lambda[b](X[a \mapsto Y])
\end{array}
$$

Exercise

Show that the rules defining beta-reduction in the lambda-calculus in the previous slide are closed.

Properties of Closed Rewriting

Closed Nominal Rewriting:

- works uniformly in α equivalence classes of terms.

Properties of Closed Rewriting

Closed Nominal Rewriting:

- works uniformly in α equivalence classes of terms.
- is expressive: can encode Combinatory Reduction Systems.

Properties of Closed Rewriting

Closed Nominal Rewriting:

- works uniformly in α equivalence classes of terms.
- is expressive: can encode Combinatory Reduction Systems.
- is efficient: linear matching.

Closed Nominal Rewriting:

- works uniformly in α equivalence classes of terms.
- is expressive: can encode Combinatory Reduction Systems.
- is efficient: linear matching.
- inherits confluence conditions from first order rewriting.

Confluence — Critical Pairs

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

Then $\Gamma \vdash (r_1\theta, L\theta[r_2\theta])$ is a **critical pair**.
If $L = [\text{-}]$ and R_1, R_2 are copies of the same rule, or if l_1' is a variable, then we say the critical pair is **trivial**.

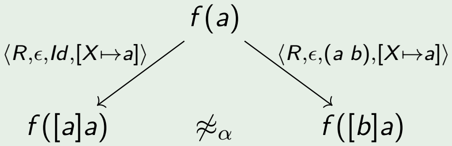
We distinguish:
If R_2 is a copy of R_1^π, the overlap is **permutative**.
Root-permutative overlap: permutative overlap at the root.
Proper overlap: not trivial and not root-permutative
Same terminology for critical pairs.

Confluence — Critical Pairs

Permutative overlap $\longrightarrow$ critical pair between rules R and R^{π}.
Only the root-permutative overlaps where π is Id are trivial.
While overlaps at the root between variable-renamed versions of first-order rules can be discarded (they generate equal terms), in nominal rewriting we must consider non-trivial root-permutative overlaps. Indeed, they do not necessarily produce the same result.

Example

$R = (\vdash f(X) \to f([a]X))$ and $R^{(a\ b)} = (\vdash f(X) \to f([b]X))$ have a non-trivial root-permutative overlap.
Critical pair: $\vdash (f([a]X), f([b]X))$. Note that $f([a]X) \not\approx_{\alpha} f([b]X)$.
This theory is not confluent; we have for instance:



For uniform rules (i.e., rules that do not generate new atoms), joinability of non-trivial critical pairs implies local confluence; also confluence if terminating (Newman's Lemma).

Joinability of proper critical pairs is insufficient for local confluence, even for a uniform theory:
the rule in Example above is uniform. However, it is not α-stable:
$R = \nabla \vdash l \to r$ is α-**stable** when, for all $\Delta, \pi, \sigma, \sigma'$,
$\Delta \vdash \nabla\sigma, \nabla^{\pi}\sigma', l\sigma \approx_{\alpha} l^{\pi}\sigma'$ implies $\Delta \vdash r\sigma \approx_{\alpha} r^{\pi}\sigma'$.

Critical Pair Lemma for uniform α-stable theories:
Let R $= (\Sigma, Rw)$ be a uniform rewrite theory where all the rewrite rules in Rw are α-stable. If every proper critical pair is joinable, then R is locally confluent.

α-stability is difficult to check, however,
closed rules are α-stable.

The reverse implication does not hold:
$\vdash f(a) \rightarrow a$ is α-stable but not closed.

Corollary:
A closed nominal rewrite system where all proper critical pairs are joinable is locally confluent.

Even better: checking *fresh overlaps* and *fresh critical pairs* is sufficient for closed rewriting.

Let $R_i = \nabla_i \vdash l_i \to r_i$ $(i = 1, 2)$ be **freshened versions** of rules.
If the nominal unification problem $\nabla_1 \cup \nabla_2 \cup \{l_2 \ {}_?{\approx}_? \ l_1|_p\}$ has a most general solution $\langle \Gamma, \theta \rangle$ for some p, then R_1 **fresh overlaps** with R_2, and $\Gamma \vdash (r_1\theta, l_1\theta[p \leftarrow r_2\theta])$ is a **fresh critical pair**.
If p is a variable position, or if R_1 and R_2 are equal modulo renaming of variables and $p = \epsilon$, then we call the overlap and critical pair **trivial**.
If R_1 and R_2 are freshened versions of the same rule and $p = \epsilon$, then we call the overlap and critical pair **fresh root-permutative**.
A fresh overlap (resp. fresh critical pair) that is not trivial and not root-permutative is **proper**.

The fresh critical pair $\Gamma \vdash (r_1\theta, l_1\theta[p \leftarrow r_2\theta])$ is **joinable** if there is a term u such that $\Gamma \vdash_{\mathsf{R}} r_1\theta \rightarrow_c u$ and $\Gamma \vdash_{\mathsf{R}} (l_1\theta[p \leftarrow r_2\theta]) \rightarrow_c u$.

Critical Pair Lemma for Closed Rewriting:

Let $\mathsf{R} = (\Sigma, Rw)$ be a rewrite theory where every proper fresh critical pair is joinable. Then the closed rewriting relation generated by R is locally confluent.

Confluence — Critical Pairs

Since it is sufficient to consider just one freshened version of each rule when computing overlaps of closed rules, the number of fresh critical pairs for a finite set of rules is finite.
Thus, we have an effective criterion for local confluence, similar to the criterion for first-order systems.

Example

Explicit substitution rules in the λ-calculus (all rules except Beta) are locally confluent: every proper fresh critical pair is joinable.
If we include Beta then the system is not locally confluent.
This does not contradict the previous theorem: there is a proper fresh critical pair between (Beta) and (σ_{app}), which is not joinable, obtained from $\varnothing \vdash ((\lambda[a]X)Y)[b \mapsto Z]$:

$$\varnothing \vdash (((\lambda[a]X)[b \mapsto Z])(Y[b \mapsto Z]), (X[a \mapsto Y])[b \mapsto Z]).$$

Exercise: Critical Pairs

Compute all the proper, fresh critical pairs of the system defining beta-reduction in the lambda-calculus.

Confluence — Orthogonality

Theorem

Orthogonal (i.e., left-linear, no non-trivial overlaps) uniform nominal rewriting systems are confluent.

Call a rewrite theory $\mathrm{R} = (\Sigma, Rw)$ **fresh quasi-orthogonal** when all rules are left-linear and there are no proper fresh critical pairs.

Theorem

If R *is a fresh-quasi-orthogonal rewrite system, then the closed rewriting relation generated by* R *is confluent.*

Example

First-order logic signature: $\neg$, $\forall$ and $\exists$ of arity 1, and $\wedge, \vee$ of arity 2 (infix).
Closed rules to simplify formulas:

$\vdash \neg(X \wedge Y) \to \neg(X) \vee \neg(Y)$ and $b \# X \vdash \neg(\forall[a]X) \to \exists[b]\neg((b\ a)\cdot X)$.

Confluence — Orthogonality

The criteria for local confluence / confluence of closed rewriting are easy to check using a **nominal unification algorithm**: just compute overlaps for the set of rules obtained by taking one freshened copy of each given rule.

For comparison, the criteria for general nominal rewriting require the computation of critical pairs for permutative variants of rules, which needs equivariant unification (exponential).

Types

So far, we have discussed untyped nominal terms.

There are also typed versions:

- many-sorted
- Simply typed — Church-style and Curry-style
- Polymorphic Curry-style systems (next slides)
- Intersection type assignment systems
- Dependently typed systems

Polymorphic Curry-Style Types for Nominal Terms

Types built from

- a set of base data sorts δ (e.g. Nat, Bool, Exp, ...), and
- type variables α,
- using type constructors C (e.g. *List*, $\rightarrow$, ...)

Types:

$$\sigma, \tau ::= \delta \mid \alpha \mid (\tau_1 \times \ldots \times \tau_n) \mid C\ \tau \mid [\sigma]\tau$$

Type declarations:

$$\rho ::= \forall(\overline{\alpha}).\langle \sigma \hookrightarrow \tau \rangle$$

Example

succ : $\langle \mathtt{Nat} \hookrightarrow \mathtt{Nat} \rangle$
length : $\forall(\alpha).\langle \mathtt{List}\,\alpha \hookrightarrow \mathtt{Nat} \rangle \ \equiv\ \forall(\beta).\langle \mathtt{List}\,\beta \hookrightarrow \mathtt{Nat} \rangle$

Instantiation: E.g. $\forall(\alpha).\langle \alpha \hookrightarrow \alpha \rangle \succcurlyeq \langle \mathtt{Nat} \hookrightarrow \mathtt{Nat} \rangle$

Quasi-typing judgements: $\Gamma \Vdash_\Sigma \Delta \vdash s\colon \tau$, defined inductively, where Γ is a typing context, Σ a signature (set of declarations for term-formers), Δ a freshness context, s a term and τ a type. Δ needed later.

$$\frac{\Gamma_a \equiv \tau}{\Gamma \Vdash_\Sigma \Delta \vdash a\colon \tau}(atm)^\tau \qquad \frac{\Gamma_X \equiv \tau}{\Gamma \Vdash_\Sigma \Delta \vdash \pi \cdot X\colon \tau}(var)^\tau$$

$$\frac{\Sigma_{\mathsf{f}} \succcurlyeq \langle \sigma \hookrightarrow \tau \rangle \quad \Gamma \Vdash_\Sigma \Delta \vdash t\colon \sigma}{\Gamma \Vdash_\Sigma \Delta \vdash \mathsf{f}\, t\colon \tau} \qquad \frac{\Gamma \bowtie (a\colon \tau) \Vdash_\Sigma \Delta \vdash t\colon \tau'}{\Gamma \Vdash_\Sigma \Delta \vdash [a]\, t\colon [\tau]\, \tau'}$$

$$\frac{\Gamma \Vdash_\Sigma \Delta \vdash t_1\colon \tau_1 \quad \ldots \quad \Gamma \Vdash_\Sigma \Delta \vdash t_n\colon \tau_n}{\Gamma \Vdash_\Sigma \Delta \vdash (t_1, \ldots, t_n)\colon (\tau_1 \times \ldots \times \tau_n)}(tpl)^\tau$$

Typing Judgements

Typing judgement:
A derivable quasi-typing judgement such that for every X,
all occurrences of X are typed in the same *essential environment*:
$\Gamma^{\pi^{-1}} - \Delta_X$ is the same for any $\pi \cdot X$ in t.

The latter is called *linearity property*.

Notation for typing judgements: $\Gamma \Vdash_\Sigma \Delta \vdash s : \tau$

Examples

$$a\colon \alpha,\ X\colon \beta \Vdash_{\varnothing} \varnothing \ \vdash\ (a,\ X)\colon (\alpha \times \beta)$$
$$\varnothing \Vdash_{\varnothing} \varnothing \ \vdash\ [a]\,a\colon [\alpha]\,\alpha$$
$$a\colon \beta \Vdash_{\varnothing} \varnothing \ \vdash\ [a]\,a\colon [\alpha]\,\alpha$$
$$a\colon \tau_1,\ b\colon \tau_2,\ X\colon \tau \Vdash_{\varnothing} \varnothing \ \vdash\ (a\ b)\cdot X\colon \tau$$
$$a\colon \tau_1,\ b\colon \tau_1,\ X\colon \tau \Vdash_{\varnothing} \varnothing \ \vdash\ ((a\ b)\cdot X,\ Id\cdot X)\colon (\tau \times \tau)$$
$$X\colon \tau \Vdash_{\varnothing} a \# X \ \vdash\ ([a]\,Id\cdot X,\ Id\cdot X)\colon (\tau \times \tau)$$
$$a\colon \alpha,\ b\colon \beta,\ X\colon \tau \Vdash_{\varnothing} \varnothing \ \vdash\ [a]\,((a\ b)\cdot X,\ Id\cdot X)\colon [\beta]\,(\tau \times \tau)$$

Exercise: Show that each of these typing judgements is valid.

Type System Features

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,
- declarations for function symbols (term formers).

Principal Types

- Every term has a principal type, and type inference is decidable.
- Principal types are obtained using a function $pt(\Gamma, \Sigma, \Delta, s)$: given a typeability problem $\Gamma \Vdash_{\Sigma} \Delta \vdash t$, pt returns a pair (S, τ) of a type substitution and a type, such that the quasi-typing judgement $\Gamma\, S \Vvdash_{\Sigma} \Delta \vdash t : \tau$ is derivable and satisfies the linearity property, or fails if there is no such S, τ.
- pt implemented in two phases:
  1) build a quasi-typing judgement derivation,
  2) check essential typings.
- pt is sound and complete.

Properties

- Meta-level equivariance of typing judgements:
  if $\Gamma \Vdash_{\Sigma} \Delta \vdash t : \tau$, then ${}^{\pi}\Gamma \Vdash_{\Sigma} {}^{\pi}\Delta \vdash {}^{\pi}t : \tau$.
- Object-level equivariance of typing judgements:
  if $\Gamma \Vdash_{\Sigma} \Delta \vdash t : \tau$ then ${}^{\pi}\Gamma \Vdash_{\Sigma} \Delta \vdash \pi \cdot t : \tau$.
- Well-typed substitutions preserve types:
  If θ is well-typed in Γ, Σ and Δ for $\Phi \Vdash_{\Sigma} \nabla \vdash t : \tau$, then $\Gamma \Vdash_{\Sigma} \Delta \vdash t\theta : \tau$.
- α-equivalence preserves types:
  $\Delta \vdash s \approx_{\alpha} t$ and $\Gamma \Vdash_{\Sigma} \Delta \vdash s : \tau$ imply $\Gamma \Vdash_{\Sigma} \Delta \vdash t : \tau$.

Typeable rewrite rule $\Phi \Vdash_\Sigma \nabla \vdash l \rightarrow r : \tau$

1. $\nabla \vdash l \rightarrow r$ is a uniform rule;
2. $\mathrm{pt}(\Phi \Vdash_\Sigma \nabla \vdash l) = (Id, \tau)$ and $\Phi \Vdash_\Sigma \nabla \vdash (l, r) : (\tau \times \tau)$.

Remark: reductions do not generate new atoms (uniform rules); l and r are both typeable with the principal type of l, so the essential environments of both sides of the rule are the same (key!).

Typed Nominal Matching: The substitution must be will be typed.

Subject Reduction:
The rewrite relation generated by typeable rewrite rules using **typed nominal matching** preserves types.

Given $\Sigma = \{\mathsf{lam}\colon \forall(\alpha,\ \beta).\langle [\alpha]\,\beta \hookrightarrow \alpha \Rightarrow \beta\rangle,\ \mathsf{app}\colon \forall(\alpha,\ \beta).\langle (\alpha \Rightarrow \beta \times \alpha) \hookrightarrow \beta\rangle,\ \mathsf{sub}\colon \forall(\alpha,\ \beta).\langle ([\alpha]\,\beta \times \alpha) \hookrightarrow \beta\rangle\}$, where $\Rightarrow$, is a type-former, written infix, to construct function types, the following rules are typeable.

$$X\colon \alpha,\ Y\colon \beta \Vdash_{\Sigma} \varnothing\ \vdash\ \mathsf{app}\,((\mathsf{lam}\,[a]\,X),\ Y) \to \mathsf{sub}\,([a]\,X,\ Y)\colon \alpha$$

$$X\colon \alpha \Rightarrow \beta \Vdash_{\Sigma} a \# X\ \vdash\ \mathsf{lam}\,[a]\,(\mathsf{app}\,(X,\ a)) \to X\colon \alpha \Rightarrow \beta$$

Exercise: Typeable Rewrite Rules for the Lambda-Calculus

Exercise: Show that the rules below satisfy the conditions in the definition of typeable rule.

$$X : \alpha,\ Z : \gamma \Vdash_\Sigma a \# X \ \vdash\ \mathsf{sub}\,([a]\,X,\ Z) \to X : \alpha$$

$$Z : \gamma \Vdash_\Sigma \varnothing \ \vdash\ \mathsf{sub}\,([a]\,a,\ Z) \to Z : \gamma$$

$$X : \beta \Rightarrow \alpha,\ Y : \beta,\ Z : \gamma \Vdash_\Sigma \varnothing \ \vdash\ \mathsf{sub}\,([a]\,(\mathsf{app}\,(X,\ Y)),\ Z)$$
$$\to \mathsf{app}\,(\mathsf{sub}\,([a]\,X,\ Z),\ \mathsf{sub}\,([a]\,Y,\ Z)) : \alpha$$

$$X : \alpha,\ Z : \gamma \Vdash_\Sigma b \# Z \ \vdash\ \mathsf{sub}\,([a]\,(\mathsf{lam}\,[b]\,X),\ Z)$$
$$\to \mathsf{lam}\,[b]\,(\mathsf{sub}\,([a]\,X,\ Z)) : \alpha' \Rightarrow \alpha$$

Why Typed Matching?

Assume $\Sigma_{\mathsf{f}} = \forall(\alpha).\langle\alpha \hookrightarrow \mathtt{Nat}\rangle$ and $\Sigma_{\mathsf{true}} = \langle() \hookrightarrow \mathtt{Bool}\rangle$ and a rule

$$X : \mathtt{Nat} \Vdash_{\Sigma} \varnothing \vdash \mathsf{f}\, X \to X : Nat$$

The untyped pattern-matching problem $\varnothing \vdash \mathsf{f}\, X \;{}^{?}{\approx_\alpha}\; \varnothing \vdash \mathsf{f}\,\mathsf{true}$ has a solution $X \mapsto \mathsf{true}$.

The typed pattern matching problem $(X : \mathtt{Nat} \Vdash_{\Sigma} \varnothing \vdash \mathsf{f}\, X) \;{}^{?}{\approx_\alpha}\; (\varnothing \Vdash_{\Sigma} \varnothing \vdash \mathsf{f}\,\mathsf{true})$ has none: the substitution $X \mapsto \mathsf{true}$ is not well-typed, because X is required to have the type `Nat`, but it is instantiated with a term of type `Bool`.

More efficient: Typed Closed Nominal Rewriting

Typeable-closed rewrite rule $\Phi \Vdash_{\Sigma} \nabla \vdash l \to r : \tau$

1. $\nabla \vdash l \to r$ is closed.
2. $\mathrm{pt}(\Phi \Vdash_{\Sigma} \nabla \vdash l) = (Id, \tau)$ and $\Phi \Vdash_{\Sigma} \nabla \vdash (l, r) : (\tau \times \tau)$.
3. Every variable in l has an occurrence within a function application $\mathsf{f}\, t$, and for every subderivation $\Gamma' \Vdash_{\Sigma} \Delta \vdash \mathsf{f}\, t : \tau'$ in l where t is not ground, if $\Sigma_{\mathsf{f}} = \forall(\overline{\alpha}).\langle \sigma \hookrightarrow \tau \rangle$, then the type of t is as general as σ.

Subject Reduction:

The closed rewriting relation generated by typeable-closed rules preserves types.

Exercises: Typed Closed Nominal Rewriting

Consider again the rewrite system defining beta-reduction in the lambda-calculus.
Are all the rules typeable-closed?

Theories with AC operators

Recall:

First Order E-Unification problem:

Instance: given two terms s and t and an equational theory E.

Question: is there a substitution σ such that $s\sigma =_E t\sigma$?

Recall:

First Order E-Unification problem:
Instance: given two terms s and t and an equational theory E.
Question: is there a substitution σ such that $s\sigma =_E t\sigma$?

Undecidable in general!

Recall:

First Order E-Unification problem:
Instance: given two terms s and t and an equational theory E.
Question: is there a substitution σ such that $s\sigma =_E t\sigma$?

Undecidable in general!

Decidable subcases: C, AC, ACU, . . .
[Baader, Kapur, Narendran, Siekmann, Schmidt-Schauß, etc..]

Nominal Equational Unification problem:

Instance: given two nominal terms s and t and an equational theory E.

Question: is there a substitution σ and a freshness context ∇ such that $\nabla \vdash s\sigma \approx_{\alpha,E} t\sigma$?

Nominal Equational Unification problem:

Instance: given two nominal terms s and t and an equational theory E.

Question: is there a substitution σ and a freshness context ∇ such that $\nabla \vdash s\sigma \approx_{\alpha,E} t\sigma$?

Nominal E-Unification: α and E.
Modular extension of first-order equational unification procedures?

Nominal Equational Unification problem:

Instance: given two nominal terms s and t and an equational theory E.

Question: is there a substitution σ and a freshness context ∇ such that $\nabla \vdash s\sigma \approx_{\alpha,E} t\sigma$?

Nominal E-Unification: α and E.
Modular extension of first-order equational unification procedures?

It depends on the theory E...

Interference: Commutative Symbols OR, $+$

$$\forall[a]\texttt{OR}(p(a), p((c\ d)\cdot X)) \approx_\alpha{}^? \forall[b]\texttt{OR}(p((a\ b)\cdot X), p(b))$$

$$\Downarrow$$

Interference: Commutative Symbols OR, $+$

$$\forall[a]\mathtt{OR}(p(a), p((c\ d)\cdot X)) \approx_\alpha{}^? \forall[b]\mathtt{OR}(p((a\ b)\cdot X), p(b))$$

$$\Downarrow$$

$$\mathtt{OR}(p(a), p((c\ d)\cdot X))) \approx_\alpha{}^? (a\ b)\cdot\mathtt{OR}(p((a\ b)\cdot X), p(b)),$$

$$a\#^?\mathtt{OR}(p((a\ b)\cdot X), p(b))$$

$$\Downarrow^*$$

Interference: Commutative Symbols OR, $+$

$$\forall[a]\mathtt{OR}(p(a), p((c\ d)\cdot X)) \approx_\alpha{}^? \forall[b]\mathtt{OR}(p((a\ b)\cdot X), p(b))$$

$$\Downarrow$$

$$\mathtt{OR}(p(a), p((c\ d)\cdot X))) \approx_\alpha{}^? (a\ b)\cdot \mathtt{OR}(p((a\ b)\cdot X), p(b)),$$
$$a\#^?\mathtt{OR}(p((a\ b)\cdot X), p(b))$$

$$\Downarrow^*$$

$$\mathtt{OR}(p(a), p((c\ d)\cdot X))) \approx_\alpha{}^? \mathtt{OR}(p(X), p(a)), b\#^? X$$

$$\Downarrow$$

$$p(a) \approx_\alpha{}^? p(X),\ p((c\ d)\cdot X) \approx_\alpha{}^? p(a),\ \ b\#X$$

$$\Downarrow$$

$$a \approx_\alpha{}^? X,\ (c\ d)\cdot X \approx_\alpha{}^? a,\ \ b\#X$$

$$\Downarrow [X \mapsto a]$$

$$(c\ d)\cdot a \ \approx_\alpha{}^? a, b\#a$$

$$\Downarrow$$

$$\perp$$

$\texttt{OR}$ is a commutative symbol:

$$\texttt{OR}(p(a), p((c\ d) \cdot X))) \approx_\alpha{}^? \texttt{OR}(p(X), p(a)), b\#^? X$$

$\mathtt{OR}$ is a commutative symbol:

$$\mathtt{OR}(p(a), p((c\ d) \cdot X)) \approx^?_{\alpha,C} \mathtt{OR}(p(X), p(a)), b\#^? X$$

$$\Downarrow$$

$$p(a) \approx_\alpha{}^? p(a),\ p((c\ d) \cdot X) \approx^?_{\alpha,C} p(X), b\#^? X$$

$$\Downarrow$$

$$p((c\ d) \cdot X) \approx^?_{\alpha,C} p(X), b\#^? X$$

$$\Downarrow$$

$$(c\ d) \cdot X \approx^?_{\alpha,C} X, b\#^? X$$

$(c\ d) \cdot X \approx^?_{\alpha,C} X$ has infinite principal solutions!

- $X \mapsto c + d, X \mapsto f(c + d), X \mapsto [e]c + [e]d, \ldots$

Nominal C-Unification Procedure [Ayala-Rincón et al.]:

1. Simplification phase:
   Build a derivation tree (branching for C symbols)
2. Solve fixed point constraints $X \approx_{\alpha,C} \pi \cdot X$

$(c\ d) \cdot X \approx^?_{\alpha,C} X$ has infinite principal solutions!

- $X \mapsto c + d, X \mapsto f(c + d), X \mapsto [e]c + [e]d, \ldots$

Nominal C-Unification Procedure [Ayala-Rincón et al.]:

1. Simplification phase:
   Build a derivation tree (branching for C symbols)
2. Solve fixed point constraints $X \approx_{\alpha,C} \pi \cdot X$

First-order C-unification is finitary.

$(c\ d) \cdot X \approx^?_{\alpha,C} X$ has infinite principal solutions!

- $X \mapsto c + d, X \mapsto f(c + d), X \mapsto [e]c + [e]d, \ldots$

Nominal C-Unification Procedure [Ayala-Rincón et al.]:

1. Simplification phase:
   Build a derivation tree (branching for C symbols)
2. Solve fixed point constraints $X \approx_{\alpha,C} \pi \cdot X$

First-order C-unification is finitary.
Nominal C-unification is NOT, if we represent solutions using substitutions and freshness contexts.
Alternative representation?

Nominal Sets

$\mathtt{Perm}(\mathbb{A})$: group of finite permutations of $\mathbb{A}$
S: set equipped with an action of the group $\mathtt{Perm}(\mathbb{A})$

Definition

$A \subset \mathbb{A}$ is a *support* for an element $x \in S$ if for all $\pi \in \mathtt{Perm}(\mathbb{A})$

$$((\forall a \in A)\ \pi(a) = a) \Rightarrow \pi \cdot x = x \qquad (1)$$

A *nominal set* is a set equipped with an action of the group $\mathtt{Perm}(\mathbb{A})$, all of whose elements have finite support.

$\mathrm{supp}_S(x)$: least finite support of x
Example:
If $a \in \mathbb{A}$ then $\mathrm{supp}(a) = \{a\}$
$\mathrm{supp}(\mathrm{app}(a, g(c, d))) = \{a, c, d\}$

Freshness vs. Fixed-Point Constraints

Characterisation of Freshness [Pitts2013, Theorem 3.9]:

$$a\#X \Leftrightarrow \text{И} a'.(a\ a') \cdot X = X$$

Freshness *derived* from И and a notion of permutation fixed-point.

Freshness vs. Fixed-Point Constraints

Characterisation of Freshness [Pitts2013, Theorem 3.9]:

$$a\#X \Leftrightarrow \text{И}a'.(a\ a') \cdot X = X$$

Freshness *derived* from И and a notion of permutation fixed-point.

Let S be a nominal set.
The *fixed-point relation* $\curlywedge \subseteq \mathtt{Perm}(\mathbb{A}) \times S$ is defined as:
$\pi \curlywedge x \Leftrightarrow \pi \cdot x = x$
Read "$\pi \curlywedge x$" as "π fixes x".

α-equivalence via fixed point constraints

Notation:

- α-equivalence constraint: $s \stackrel{\curlywedge}{\approx}_\alpha t$
- Fixed-point constraint: $\pi \curlywedge t$
  Intuitively, π fixes t if $\pi \cdot t \stackrel{\curlywedge}{\approx}_\alpha t$,
  π has "no effect" on t except for possible renaming of bound names, for instance, $(a\ b) \curlywedge [a]a$ but not $(a\ b) \curlywedge \mathsf{f}\ a$.
- Primitive fixed-point constraint: $\pi \curlywedge X$
- Fixed-point context: $\Upsilon = \{\pi_1 \curlywedge X_1, \ldots, \pi_k \curlywedge X_k\}$
- Support of a permutation: $\mathtt{supp}(\pi) = \{a \mid \pi(a) \neq a\}$

Fixed-Point Rules

Notation: $\mathtt{perm}(\Upsilon|_X)$ permutations that fix X according to Υ

$$\frac{\pi(a) = a}{\Upsilon \vdash \pi \curlywedge a} (\curlywedge \mathbf{a}) \quad \frac{\mathrm{supp}(\pi^{\pi'^{-1}}) \subseteq \mathrm{supp}(\mathtt{perm}(\Upsilon|_X))}{\Upsilon \vdash \pi \curlywedge \pi' \cdot X} (\curlywedge \mathbf{var})$$

$$\frac{\Upsilon \vdash \pi \curlywedge t}{\Upsilon \vdash \pi \curlywedge \mathsf{f}\ t} (\curlywedge \mathsf{f}) \quad \frac{\Upsilon \vdash \pi \curlywedge t_1 \quad \ldots \quad \Upsilon \vdash \pi \curlywedge t_n}{\Upsilon \vdash \pi \curlywedge (t_1, \ldots, t_n)} (\curlywedge \mathbf{tuple})$$

$$\frac{\Upsilon, (c_1\ c_2) \curlywedge \mathtt{Var}(t) \vdash \pi \curlywedge (a\ c_1) \cdot t}{\Upsilon \vdash \pi \curlywedge [a]t} (\curlywedge \mathbf{abs}), \quad c_1 \text{ and } c_2 \text{ new names}$$

Alpha-Equivalence Rules

$$\frac{}{\Upsilon \vdash a \stackrel{\curlywedge}{\approx}_\alpha a}(\stackrel{\curlywedge}{\approx}_\alpha \mathbf{a}) \qquad \frac{\mathtt{supp}((\pi')^{-1} \circ \pi) \subseteq \mathtt{supp}(\mathtt{perm}(\Upsilon|_X))}{\Upsilon \vdash \pi \cdot X \stackrel{\curlywedge}{\approx}_\alpha \pi' \cdot X}(\stackrel{\curlywedge}{\approx}_\alpha \mathbf{var})$$

$$\frac{\Upsilon \vdash t \stackrel{\curlywedge}{\approx}_\alpha t'}{\Upsilon \vdash \mathsf{f}\ t \stackrel{\curlywedge}{\approx}_\alpha \mathsf{f}\ t'}(\stackrel{\curlywedge}{\approx}_\alpha \mathsf{f}) \quad \frac{\Upsilon \vdash t_1 \stackrel{\curlywedge}{\approx}_\alpha t_1' \quad \ldots \quad \Upsilon \vdash t_n \stackrel{\curlywedge}{\approx}_\alpha t_n'}{\Upsilon \vdash (t_1, \ldots, t_n) \stackrel{\curlywedge}{\approx}_\alpha (t_1', \ldots, t_n')}(\stackrel{\curlywedge}{\approx}_\alpha \mathbf{tuple})$$

$$\frac{\Upsilon \vdash t \stackrel{\curlywedge}{\approx}_\alpha t'}{\Upsilon \vdash [a]t \stackrel{\curlywedge}{\approx}_\alpha [a]t'}(\stackrel{\curlywedge}{\approx}_\alpha [\mathbf{a}])$$

$$\frac{\Upsilon \vdash s \stackrel{\curlywedge}{\approx}_\alpha (a\ b) \cdot t \quad \Upsilon, (c_1\ c_2) \curlywedge \mathtt{Var}(t) \vdash (a\ c_1) \curlywedge t}{\Upsilon \vdash [a]s \stackrel{\curlywedge}{\approx}_\alpha [b]t}(\stackrel{\curlywedge}{\approx}_\alpha \mathsf{a}\mathbf{b})$$

Correctness

Theorem

$\Upsilon \vdash \pi \curlywedge t$ *iff* $\Upsilon \vdash \pi \cdot t \overset{\curlywedge}{\approx}_\alpha t$.

$[\_]_\curlywedge$ maps freshness constraints in Δ to fixed-point constraints:

$$\begin{array}{rccl} [\_]_\curlywedge : & \Delta & \longrightarrow & \mathfrak{F}_\curlywedge \\ & a\#X & \mapsto & (a\ c_a) \curlywedge X \text{ where } c_a \text{ is a new name.} \end{array}$$

$[_]_\#$maps fixed-point constraints in Υ to freshness constraints:

$$\begin{array}{rccl} [\_]_\# : & \Upsilon & \longrightarrow & \mathfrak{F}_\# \\ & \pi \curlywedge X & \mapsto & \mathrm{supp}(\pi)\#X. \end{array}$$

Theorem

1. $\Delta \vdash s \approx_\alpha t \Rightarrow [\Delta]_\curlywedge \vdash s \overset{\curlywedge}{\approx}_\alpha t$.
2. $\Upsilon \vdash s \overset{\curlywedge}{\approx}_\alpha t \Rightarrow [\Upsilon]_\# \vdash s \approx_\alpha t$.

Simplification Rules for Nominal Unification

$$
\begin{array}{llcl}
(\curlywedge at) & \mathrm{Pr} \uplus \{\pi \curlywedge^? a\} & \Longrightarrow & \mathrm{Pr},\ \text{if } \pi(a) = a \\
(\curlywedge f) & \mathrm{Pr} \uplus \{\pi \curlywedge^? \mathsf{f}t\} & \Longrightarrow & \mathrm{Pr} \cup \{\pi \curlywedge^? t\} \\
(\curlywedge t) & \mathrm{Pr} \uplus \{\pi \curlywedge^? (\tilde{t})_n\} & \Longrightarrow & \mathrm{Pr} \cup \{\pi \curlywedge^? t_1, \ldots, \pi \curlywedge^? t_n\} \\
(\curlywedge abs) & \mathrm{Pr} \uplus \{\pi \curlywedge^? [a]t\} & \Longrightarrow & \mathrm{Pr} \cup \{\pi \curlywedge^? (a\ c_1) \cdot t, \overline{(c_1\ c_2) \curlywedge^? \mathtt{Var}(t)}\} \\
(\curlywedge var) & \mathrm{Pr} \uplus \{\pi \curlywedge^? \pi' \cdot X\} & \Longrightarrow & \mathrm{Pr} \cup \{\pi^{(\pi')^{-1}} \curlywedge^? X\},\ \text{if } \pi' \neq Id \\
(\stackrel{\curlywedge}{\approx}_\alpha a) & \mathrm{Pr} \uplus \{a \stackrel{\curlywedge}{\approx}_\alpha^? a\} & \Longrightarrow & \mathrm{Pr} \\
(\stackrel{\curlywedge}{\approx}_\alpha f) & \mathrm{Pr} \uplus \{\mathsf{f}\ t \stackrel{\curlywedge}{\approx}_\alpha^? \mathsf{f}\ t'\} & \Longrightarrow & \mathrm{Pr} \cup \{t \approx_\alpha^? t'\} \\
(\stackrel{\curlywedge}{\approx}_\alpha t) & \mathrm{Pr} \uplus \{(\tilde{t})_n \approx_\alpha^? (\tilde{t'})_n\} & \Longrightarrow & \mathrm{Pr} \cup \{t_1 \stackrel{\curlywedge}{\approx}_\alpha^? t_1', \ldots, t_n \stackrel{\curlywedge}{\approx}_\alpha^? t_n'\} \\
(\stackrel{\curlywedge}{\approx}_\alpha ab1) & \mathrm{Pr} \uplus \{[a]t \stackrel{\curlywedge}{\approx}_\alpha^? [a]t'\} & \Longrightarrow & \mathrm{Pr} \cup \{t \stackrel{\curlywedge}{\approx}_\alpha^? t'\} \\
(\stackrel{\curlywedge}{\approx}_\alpha ab2) & \mathrm{Pr} \uplus \{[a]t \stackrel{\curlywedge}{\approx}_\alpha^? [b]s\} & \Longrightarrow & \mathrm{Pr} \cup \{t \stackrel{\curlywedge}{\approx}_\alpha^? (a\ b) \cdot s, (a\ c_1) \curlywedge^? s, \\
 & & & \quad \overline{(c_1\ c_2) \curlywedge^? \mathtt{Var}(s)}\} \\
(\stackrel{\curlywedge}{\approx}_\alpha var) & \mathrm{Pr} \uplus \{\pi \cdot X \stackrel{\curlywedge}{\approx}_\alpha^? \pi' \cdot X\} & \Longrightarrow & \mathrm{Pr} \cup \{(\pi')^{-1} \circ \pi \curlywedge^? X\} \\
(\stackrel{\curlywedge}{\approx}_\alpha inst1) & \mathrm{Pr} \uplus \{\pi \cdot X \stackrel{\curlywedge}{\approx}_\alpha^? t\} & \stackrel{[X \mapsto \pi^{-1}.t]}{\Longrightarrow} & \mathrm{Pr}\{X \mapsto \pi^{-1}.t\},\ \text{if } X \notin \mathtt{Var}(t) \\
(\stackrel{\curlywedge}{\approx}_\alpha inst2) & \mathrm{Pr} \uplus \{t \stackrel{\curlywedge}{\approx}_\alpha^? \pi \cdot X\} & \stackrel{[X \mapsto \pi^{-1}.t]}{\Longrightarrow} & \mathrm{Pr}\{X \mapsto \pi^{-1}.t\},\ \text{if } X \notin \mathtt{Var}(t)
\end{array}
$$

c_1 and c_2 are new names

Correspondence: freshness/fixed-point constraints

From # constraints:

$$[a]\mathsf{f}(X, a) \approx_\alpha^? [b]\mathsf{f}((b\ c)\cdot W, (a\ c)\cdot Y))$$
$$\Downarrow$$

From # constraints:

$$[a]\mathsf{f}(X,a) \approx_\alpha^? [b]\mathsf{f}((b\ c)\cdot W,(a\ c)\cdot Y))$$
$$\Downarrow$$
$$\mathsf{f}(X,a) \approx_\alpha^? (a\ b).\mathsf{f}((b\ c).W,(a\ c).Y))$$
$$a\#\mathsf{f}((b\ c)\cdot W,(a\ c)\cdot Y))$$
$$\Downarrow$$

From # constraints:

$$[a]\mathsf{f}(X,a) \approx_\alpha^? [b]\mathsf{f}((b\ c)\cdot W,(a\ c)\cdot Y))$$
$$\Downarrow$$
$$\mathsf{f}(X,a) \approx_\alpha^? (a\ b).\mathsf{f}((b\ c).W,(a\ c).Y))$$
$$a\#\mathsf{f}((b\ c)\cdot W,(a\ c)\cdot Y))$$
$$\Downarrow$$
$$\mathsf{f}(X,a) \approx_\alpha^? \mathsf{f}((a\ b)(b\ c).W,(a\ b)(a\ c).Y))$$
$$a\#(b\ c)\cdot W,\ a\#(a\ c)\cdot Y$$
$$\Downarrow^*$$

From # constraints:

$$[a]\mathsf{f}(X, a) \approx_\alpha{}^? [b]\mathsf{f}((b\ c)\cdot W, (a\ c)\cdot Y))$$
$$\Downarrow$$
$$\mathsf{f}(X, a) \approx_\alpha{}^? (a\ b).\mathsf{f}((b\ c).W, (a\ c).Y))$$
$$a\#\mathsf{f}((b\ c)\cdot W, (a\ c)\cdot Y))$$
$$\Downarrow$$
$$\mathsf{f}(X, a) \approx_\alpha^? \mathsf{f}((a\ b)(b\ c).W, (a\ b)(a\ c).Y))$$
$$a\#(b\ c)\cdot W,\ a\#(a\ c)\cdot Y$$
$$\Downarrow^*$$
$$X \approx_\alpha{}^? (a\ b)(b\ c)\cdot W, b \approx_\alpha{}^? Y$$
$$a\#W,\ c\#Y$$
$$\Downarrow Y \mapsto b$$
$$X \approx_\alpha{}^? (a\ b)(b\ c)\cdot W$$
$$a\#W,\ c\#b$$
$$\Downarrow X \mapsto (a\ b)(b\ c)\cdot W$$
$$a\#W$$

Correspondence: freshness/fixed-point constraints

From # constraints:

$$[a]\mathsf{f}(X,a) \approx_\alpha^? [b]\mathsf{f}((b\ c)\cdot W,(a\ c)\cdot Y))$$
$$\Downarrow$$
$$\mathsf{f}(X,a) \approx_\alpha^? (a\ b).\mathsf{f}((b\ c).W,(a\ c).Y))$$
$$a\#\mathsf{f}((b\ c)\cdot W,(a\ c)\cdot Y))$$
$$\Downarrow$$
$$\mathsf{f}(X,a) \approx_\alpha^? \mathsf{f}((a\ b)(b\ c).W,(a\ b)(a\ c).Y))$$
$$a\#(b\ c)\cdot W,\ a\#(a\ c)\cdot Y$$
$$\Downarrow^*$$
$$X \approx_\alpha^? (a\ b)(b\ c)\cdot W, b \approx_\alpha^? Y$$
$$a\#W,\ c\#Y$$
$$\Downarrow Y \mapsto b$$
$$X \approx_\alpha^? (a\ b)(b\ c)\cdot W$$
$$a\#W,\ c\#b$$
$$\Downarrow X \mapsto (a\ b)(b\ c)\cdot W$$
$$a\#W$$

$$\mathtt{Sol} = (a\#W,\ \delta)$$

Correspondence: freshness/fixed-point constraints

From $\#$ constraints:

$$[a]\mathsf{f}(X,a) \approx_\alpha{}^? [b]\mathsf{f}((b\ c)\cdot W,(a\ c)\cdot Y))$$
$$\Downarrow$$
$$\mathsf{f}(X,a) \approx_\alpha{}^? (a\ b).\mathsf{f}((b\ c).W,(a\ c).Y))$$
$$a\#\mathsf{f}((b\ c)\cdot W,(a\ c)\cdot Y))$$
$$\Downarrow$$
$$\mathsf{f}(X,a) \approx_\alpha{}^? \mathsf{f}((a\ b)(b\ c).W,(a\ b)(a\ c).Y))$$
$$a\#(b\ c)\cdot W,\ a\#(a\ c)\cdot Y$$
$$\Downarrow^*$$
$$X \approx_\alpha{}^? (a\ b)(b\ c)\cdot W, b \approx_\alpha{}^? Y$$
$$a\#W,\ c\#Y$$
$$\Downarrow Y \mapsto b$$
$$X \approx_\alpha{}^? (a\ b)(b\ c)\cdot W$$
$$a\#W,\ c\#b$$
$$\Downarrow X \mapsto (a\ b)(b\ c)\cdot W$$
$$a\#W$$

$$\mathtt{Sol} = (a\#W,\ \delta)$$

To $⅄$ constraints:

$$[a]\mathsf{f}(X,a) \overset{⅄}{\approx}{}_\alpha^? [b]\mathsf{f}((b\ c).W,(a\ c).Y))$$
$$\Downarrow$$

Correspondence: freshness/fixed-point constraints

From $\#$ constraints:

$$[a]\mathsf{f}(X,a) \approx_\alpha{}^? [b]\mathsf{f}((b\ c)\cdot W, (a\ c)\cdot Y))$$
$$\Downarrow$$
$$\mathsf{f}(X,a) \approx_\alpha{}^? (a\ b).\mathsf{f}((b\ c).W, (a\ c).Y))$$
$$a\#\mathsf{f}((b\ c)\cdot W, (a\ c)\cdot Y))$$
$$\Downarrow$$
$$\mathsf{f}(X,a) \approx_\alpha{}^? \mathsf{f}((a\ b)(b\ c).W, (a\ b)(a\ c).Y))$$
$$a\#(b\ c)\cdot W,\ a\#(a\ c)\cdot Y$$
$$\Downarrow^*$$
$$X \approx_\alpha{}^? (a\ b)(b\ c)\cdot W, b \approx_\alpha{}^?\ Y$$
$$a\#W,\ c\#Y$$
$$\Downarrow\ Y \mapsto b$$
$$X \approx_\alpha{}^? (a\ b)(b\ c)\cdot W$$
$$a\#W,\ c\#b$$
$$\Downarrow\ X \mapsto (a\ b)(b\ c)\cdot W$$
$$a\#W$$

$$\mathtt{Sol} = (a\#W,\ \delta)$$

To $\curlywedge$ constraints:

$$[a]\mathsf{f}(X,a) \overset{\curlywedge}{\approx}{}_\alpha^? [b]\mathsf{f}((b\ c).W, (a\ c).Y))$$
$$\Downarrow$$
$$\mathsf{f}(X,a) \overset{\curlywedge}{\approx}{}_\alpha^? (a\ b).\mathsf{f}((b\ c).W, (a\ c).Y))$$
$$(a\ c_1) \curlywedge^? \mathsf{f}((b\ c).W, (a\ c).Y))$$
$$(c_1\ c_2) \curlywedge^? W, (c_1\ c_2) \curlywedge^? Y$$
$$\Downarrow$$

Correspondence: freshness/fixed-point constraints

From # constraints:

$$[a]\mathsf{f}(X,a) \approx_\alpha^? [b]\mathsf{f}((b\ c)\cdot W,(a\ c)\cdot Y))$$
$$\Downarrow$$
$$\mathsf{f}(X,a) \approx_\alpha^? (a\ b).\mathsf{f}((b\ c).W,(a\ c).Y))$$
$$a\#\mathsf{f}((b\ c)\cdot W,(a\ c)\cdot Y))$$
$$\Downarrow$$
$$\mathsf{f}(X,a) \approx_\alpha^? \mathsf{f}((a\ b)(b\ c).W,(a\ b)(a\ c).Y))$$
$$a\#(b\ c)\cdot W,\ a\#(a\ c)\cdot Y$$
$$\Downarrow^*$$
$$X \approx_\alpha^? (a\ b)(b\ c)\cdot W, b \approx_\alpha^? Y$$
$$a\#W,\ c\#Y$$
$$\Downarrow Y \mapsto b$$
$$X \approx_\alpha^? (a\ b)(b\ c)\cdot W$$
$$a\#W,\ c\#b$$
$$\Downarrow X \mapsto (a\ b)(b\ c)\cdot W$$
$$a\#W$$

$$\mathtt{Sol} = (a\#W,\ \delta)$$

To $\curlywedge$ constraints:

$$[a]\mathsf{f}(X,a) \stackrel{\curlywedge}{\approx}_\alpha^? [b]\mathsf{f}((b\ c).W,(a\ c).Y))$$
$$\Downarrow$$
$$\mathsf{f}(X,a) \stackrel{\curlywedge}{\approx}_\alpha^? (a\ b).\mathsf{f}((b\ c).W,(a\ c).Y))$$
$$(a\ c_1) \curlywedge^? \mathsf{f}((b\ c).W,(a\ c).Y))$$
$$(c_1\ c_2) \curlywedge^? W, (c_1\ c_2) \curlywedge^? Y$$
$$\Downarrow$$
$$\mathsf{f}(X,a) \stackrel{\curlywedge}{\approx}_\alpha^? (a\ b).\mathsf{f}((b\ c).W,(a\ c).Y))$$
$$(a\ c_1) \curlywedge^? (b\ c).W, (a\ c_1) \curlywedge^? (a\ c).Y$$
$$(c_1\ c_2) \curlywedge^? W, (c_1\ c_2) \curlywedge^? Y$$
$$\Downarrow$$

Correspondence: freshness/fixed-point constraints

From # constraints:

$$[a]\mathsf{f}(X,a) \approx_\alpha^? [b]\mathsf{f}((b\ c)\cdot W,(a\ c)\cdot Y))$$
$$\Downarrow$$
$$\mathsf{f}(X,a) \approx_\alpha^? (a\ b).\mathsf{f}((b\ c).W,(a\ c).Y))$$
$$a\#\mathsf{f}((b\ c)\cdot W,(a\ c)\cdot Y))$$
$$\Downarrow$$
$$\mathsf{f}(X,a) \approx_\alpha^? \mathsf{f}((a\ b)(b\ c).W,(a\ b)(a\ c).Y))$$
$$a\#(b\ c)\cdot W,\ a\#(a\ c)\cdot Y$$
$$\Downarrow^*$$
$$X \approx_\alpha^? (a\ b)(b\ c)\cdot W, b \approx_\alpha^? Y$$
$$a\#W,\ c\#Y$$
$$\Downarrow Y \mapsto b$$
$$X \approx_\alpha^? (a\ b)(b\ c)\cdot W$$
$$a\#W,\ c\#b$$
$$\Downarrow X \mapsto (a\ b)(b\ c)\cdot W$$
$$a\#W$$

$$\mathtt{Sol} = (a\#W,\ \delta)$$

To $\curlywedge$ constraints:

$$[a]\mathsf{f}(X,a) \overset{\curlywedge}{\approx}{}_\alpha^? [b]\mathsf{f}((b\ c).W,(a\ c).Y))$$
$$\Downarrow$$
$$\mathsf{f}(X,a) \overset{\curlywedge}{\approx}{}_\alpha^? (a\ b).\mathsf{f}((b\ c).W,(a\ c).Y))$$
$$(a\ c_1) \curlywedge^? \mathsf{f}((b\ c).W,(a\ c).Y))$$
$$(c_1\ c_2) \curlywedge^? W, (c_1\ c_2) \curlywedge^? Y$$
$$\Downarrow$$
$$\mathsf{f}(X,a) \overset{\curlywedge}{\approx}{}_\alpha^? (a\ b).\mathsf{f}((b\ c).W,(a\ c).Y))$$
$$(a\ c_1) \curlywedge^? (b\ c).W, (a\ c_1) \curlywedge^? (a\ c).Y$$
$$(c_1\ c_2) \curlywedge^? W, (c_1\ c_2) \curlywedge^? Y$$
$$\Downarrow$$
$$X \overset{\curlywedge}{\approx}{}_\alpha^? (a\ b)(b\ c).W, a \overset{\curlywedge}{\approx}{}_\alpha^? (a\ b)(a\ c).Y$$
$$(a\ c_1) \curlywedge^? W, (c\ c_1) \curlywedge^? Y$$
$$(c_1\ c_2) \curlywedge^? W, (c_1\ c_2) \curlywedge^? Y$$
$$\Downarrow X \mapsto (a\ b)(b\ c).W,\ Y \mapsto b$$
$$(a\ c_1) \curlywedge^? W, (c_1\ c_2) \curlywedge^? W$$

Correspondence: freshness/fixed-point constraints

From # constraints:

$$[a]\mathsf{f}(X,a) \approx_\alpha^? [b]\mathsf{f}((b\ c)\cdot W,(a\ c)\cdot Y))$$
$$\Downarrow$$
$$\mathsf{f}(X,a) \approx_\alpha^? (a\ b).\mathsf{f}((b\ c).W,(a\ c).Y))$$
$$a\#\mathsf{f}((b\ c)\cdot W,(a\ c)\cdot Y))$$
$$\Downarrow$$
$$\mathsf{f}(X,a) \approx_\alpha^? \mathsf{f}((a\ b)(b\ c).W,(a\ b)(a\ c).Y))$$
$$a\#(b\ c)\cdot W,\ a\#(a\ c)\cdot Y$$
$$\Downarrow^*$$
$$X \approx_\alpha^? (a\ b)(b\ c)\cdot W, b \approx_\alpha^? Y$$
$$a\#W,\ c\#Y$$
$$\Downarrow Y \mapsto b$$
$$X \approx_\alpha^? (a\ b)(b\ c)\cdot W$$
$$a\#W,\ c\#b$$
$$\Downarrow X \mapsto (a\ b)(b\ c)\cdot W$$
$$a\#W$$

$$\mathtt{Sol} = (a\#W,\ \delta)$$

To $\curlywedge$ constraints:

$$[a]\mathsf{f}(X,a) \stackrel{\curlywedge}{\approx}_\alpha^? [b]\mathsf{f}((b\ c).W,(a\ c).Y))$$
$$\Downarrow$$
$$\mathsf{f}(X,a) \stackrel{\curlywedge}{\approx}_\alpha^? (a\ b).\mathsf{f}((b\ c).W,(a\ c).Y))$$
$$(a\ c_1) \curlywedge^? \mathsf{f}((b\ c).W,(a\ c).Y))$$
$$(c_1\ c_2) \curlywedge^? W, (c_1\ c_2) \curlywedge^? Y$$
$$\Downarrow$$
$$\mathsf{f}(X,a) \stackrel{\curlywedge}{\approx}_\alpha^? (a\ b).\mathsf{f}((b\ c).W,(a\ c).Y))$$
$$(a\ c_1) \curlywedge^? (b\ c).W, (a\ c_1) \curlywedge^? (a\ c).Y$$
$$(c_1\ c_2) \curlywedge^? W, (c_1\ c_2) \curlywedge^? Y$$
$$\Downarrow$$
$$X \stackrel{\curlywedge}{\approx}_\alpha^? (a\ b)(b\ c).W, a \stackrel{\curlywedge}{\approx}_\alpha^? (a\ b)(a\ c).Y$$
$$(a\ c_1) \curlywedge^? W, (c\ c_1) \curlywedge^? Y$$
$$(c_1\ c_2) \curlywedge^? W, (c_1\ c_2) \curlywedge^? Y$$
$$\Downarrow X \mapsto (a\ b)(b\ c).W,\ Y \mapsto b$$
$$(a\ c_1) \curlywedge^? W, (c_1\ c_2) \curlywedge^? W$$

$$\mathtt{Sol} = ((a\ c_1) \curlywedge W, (c_1\ c_2) \curlywedge W,\ \delta)$$

C-fixed point constraints

$+$: commutative symbol

C-*fixed-point constraint*: $\pi \curlywedge_C t$

C-α-*equality constraint*: $s \overset{\curlywedge}{\approx}_C t$

$$+((a\ b)\cdot X,\, a) \overset{\curlywedge ?}{\approx}_C +(Y,\, X)$$

$$\swarrow \qquad\qquad\qquad \searrow$$

C-fixed point constraints

$+$: commutative symbol

C-*fixed-point constraint*: $\pi \curlywedge_C t$

C-α-*equality constraint*: $s \stackrel{\curlywedge}{\approx}_C t$

$$+((a\ b)\cdot X, a) \stackrel{\curlywedge ?}{\approx}_C +(Y, X)$$

$$\swarrow \qquad\qquad \searrow$$

$$\{(a\ b)\cdot X \stackrel{\curlywedge ?}{\approx}_C Y, a \stackrel{\curlywedge ?}{\approx}_C X\}$$
$$\Downarrow [X \mapsto a]$$
$$\{(a\ b)\cdot a \stackrel{\curlywedge ?}{\approx}_C Y\}$$
$$\Downarrow$$
$$\{b \stackrel{\curlywedge ?}{\approx}_C Y\}$$
$$\Downarrow [Y \mapsto b]$$
$$(\emptyset, \{X \mapsto a, Y \mapsto b\})$$

C-fixed point constraints

$+$: commutative symbol

C-*fixed-point constraint*: $\pi \curlywedge_C t$

C-α-*equality constraint*: $s \overset{\curlywedge}{\approx}_C t$

$$+((a\ b)\cdot X,\ a) \overset{\curlywedge ?}{\approx}_C +(Y,\ X)$$

$$\swarrow \qquad\qquad \searrow$$

$$\begin{array}{c}
\{(a\ b)\cdot X \overset{\curlywedge ?}{\approx}_C Y, a \overset{\curlywedge ?}{\approx}_C X\} \\
\Downarrow\ [X \mapsto a] \\
\{(a\ b)\cdot a \overset{\curlywedge ?}{\approx}_C Y\} \\
\Downarrow \\
\{b \overset{\curlywedge ?}{\approx}_C Y\} \\
\Downarrow\ [Y \mapsto b] \\
(\emptyset,\ \{X \mapsto a, Y \mapsto b\})
\end{array}
\qquad
\begin{array}{c}
\{(a\ b)\cdot X \overset{\curlywedge ?}{\approx}_C X, a \overset{\curlywedge ?}{\approx}_C Y\} \\
\Downarrow\ [Y \mapsto a] \\
\{(a\ b)\cdot X \overset{\curlywedge ?}{\approx}_C X\} \\
\Downarrow \\
\{(a\ b) \curlywedge^?_C X\} \\
\Downarrow \\
((a\ b) \curlywedge_C X,\ \{Y \mapsto a\})
\end{array}$$

Fixed Point Rules

$$\frac{\pi(a) = a}{\Upsilon \vdash \pi \curlywedge_C a} (\curlywedge_\mathbf{C}\mathbf{a}) \quad \frac{\mathtt{supp}(\pi^{\pi'^{-1}}) \subseteq \mathtt{supp}(\mathtt{perm}(\Upsilon|_X))}{\Upsilon \vdash \pi \curlywedge_C \pi' \cdot X} (\curlywedge_\mathbf{C}\mathbf{var})$$

$$\frac{\Upsilon \vdash \pi \curlywedge_C t}{\Upsilon \vdash \pi \curlywedge_C \mathsf{f}\, t} \mathsf{f} \neq + \, (\curlywedge_\mathbf{C}\mathsf{f}) \quad \frac{\Upsilon \vdash \pi \curlywedge_C t_1 \quad \ldots \quad \Upsilon \vdash \pi \curlywedge_C t_n}{\Upsilon \vdash \pi \curlywedge_C (t_1, \ldots, t_n)} (\curlywedge_\mathbf{C}\mathbf{tu}$$

$$\frac{\Upsilon \vdash \pi \cdot t_0 \overset{\curlywedge}{\approx}_C t_i \quad \Upsilon \vdash \pi \cdot t_1 \overset{\curlywedge}{\approx}_C t_{(i+1) \bmod 2}}{\Upsilon \vdash \pi \curlywedge_C +(t_0, t_1)} i = 0, 1 (\curlywedge_\mathbf{C}+)$$

$$\frac{\Upsilon, (c_1\ c_2) \curlywedge_C \mathtt{Var}(t) \vdash \pi \curlywedge_C (a\ c_1) \cdot t}{\Upsilon \vdash \pi \curlywedge_C [a]t} (\curlywedge_\mathbf{C}\mathbf{abs})$$

Alpha-Equality Rules

$$\frac{}{\Upsilon \vdash a \overset{\curlywedge}{\approx}_C a} (\overset{\curlywedge}{\approx}_{\mathbf{C}} \mathbf{a}) \quad \frac{\Upsilon \vdash (\pi')^{-1} \circ \pi \curlywedge_C X}{\Upsilon \vdash \pi \cdot X \overset{\curlywedge}{\approx}_C \pi' \cdot X} (\overset{\curlywedge}{\approx}_{\mathbf{C}} \mathbf{var})$$

$$\frac{\Upsilon \vdash t \overset{\curlywedge}{\approx}_C t'}{\Upsilon \vdash \mathsf{f}\, t \overset{\curlywedge}{\approx}_C \mathsf{f}\, t'} (\overset{\curlywedge}{\approx}_{\mathbf{C}} \mathsf{f}, \mathsf{f} \neq +) \quad \frac{\Upsilon \vdash t_1 \overset{\curlywedge}{\approx}_C t_1' \quad \ldots \quad \Upsilon \vdash t_n \overset{\curlywedge}{\approx}_C t_n'}{\Upsilon \vdash (t_1, \ldots, t_n) \overset{\curlywedge}{\approx}_C (t_1', \ldots, t_n')} (\overset{\curlywedge}{\approx}_{\mathbf{C}} \mathbf{tup})$$

$$\frac{\Upsilon \vdash s_0 \overset{\curlywedge}{\approx}_C t_i \quad s_1 \overset{\curlywedge}{\approx}_C t_{(i+1) \bmod 2}}{\Upsilon \vdash +\langle s_0, s_1 \rangle \overset{\curlywedge}{\approx}_C +\langle t_0, t_1 \rangle} i = 0, 1 \; (\overset{\curlywedge}{\approx}_{\mathbf{C}} +)$$

$$\frac{\Upsilon \vdash t \overset{\curlywedge}{\approx}_C t'}{\Upsilon \vdash [a]t \overset{\curlywedge}{\approx}_C [a]t'} (\overset{\curlywedge}{\approx}_{\mathbf{C}} [\mathbf{a}])$$

$$\frac{\Upsilon \vdash s \overset{\curlywedge}{\approx}_C (a\ b)t \quad \Upsilon, (c_1\ c_2) \curlywedge_C \mathtt{Var}(t) \vdash (a\ c_1) \curlywedge_C t}{\Upsilon \vdash [a]s \overset{\curlywedge}{\approx}_C [b]t} (\overset{\curlywedge}{\approx}_{\mathbf{C}} \mathbf{ab})$$

Simplification rules for nominal C-unification

$$
\begin{array}{lcl}
\mathtt{Pr} \uplus \{\pi \curlywedge^{?}_{C} a\} & \Longrightarrow & \mathtt{Pr}, \text{ if } \ \pi(a) = a \\
\mathtt{Pr} \uplus \{\pi \curlywedge^{?}_{C} \mathsf{f}\, t\} & \Longrightarrow & \mathtt{Pr} \cup \{\pi \curlywedge^{?}_{C} t\}, \mathsf{f} \neq + \\
\mathtt{Pr} \uplus \{\pi \curlywedge^{?}_{C} +(t_0,\ t_1)\} & \Longrightarrow & \mathtt{Pr} \cup \{\pi \cdot t_0 \approx^{?} t_0, \pi \cdot t_1 \approx^{?} t_1\} \\
\mathtt{Pr} \uplus \{\pi \curlywedge^{?}_{C} +(t_0,\ t_1)\} & \Longrightarrow & \mathtt{Pr} \cup \{\pi \cdot t_0 \approx^{?} t_1, \pi \cdot t_1 \approx^{?} t_0\} \\
\mathtt{Pr} \uplus \{\pi \curlywedge^{?}_{C} (\widetilde{t})_n\} & \Longrightarrow & \mathtt{Pr} \cup \{\pi \curlywedge^{?}_{C} t_1, \ldots, \pi \curlywedge^{?}_{C} t_n\} \\
\mathtt{Pr} \uplus \{\pi \curlywedge^{?}_{C} [a]t\} & \Longrightarrow & \mathtt{Pr} \cup \{\pi \curlywedge^{?}_{C} (a\ c_1) \cdot t, \overline{(c_1\ c_2) \curlywedge^{?}_{C} \mathtt{Var}(t)}\} \\
\mathtt{Pr} \uplus \{\pi \curlywedge^{?}_{C} \pi' \cdot X\} & \Longrightarrow & \mathtt{Pr} \cup \{\pi^{(\pi')^{-1}} \curlywedge^{?}_{C} X\}, \text{ if } \pi' \neq Id \\
\mathtt{Pr} \uplus \{\mathsf{f}\, t \overset{\curlywedge?}{\approx}_{C} \mathsf{f}\, t'\} & \Longrightarrow & \mathtt{Pr} \cup \{t \overset{\curlywedge?}{\approx}_{C} t'\},\ \mathsf{f} \neq + \\
\mathtt{Pr} \uplus \{+(t_0,\ t_1) \overset{\curlywedge?}{\approx}_{C} +(s_0,\ s_1)\} & \Longrightarrow & \mathtt{Pr} \cup \{t_0 \overset{\curlywedge?}{\approx}_{C} s_0, t_1 \overset{\curlywedge?}{\approx}_{C} s_1\} \\
\mathtt{Pr} \uplus \{+(t_0,\ t_1) \overset{\curlywedge?}{\approx}_{C} +(s_0,\ s_1)\} & \Longrightarrow & \mathtt{Pr} \cup \{t_0 \overset{\curlywedge?}{\approx}_{C} s_1, t_1 \overset{\curlywedge?}{\approx}_{C} s_0\} \\
\mathtt{Pr} \uplus \{(\widetilde{t})_n \overset{\curlywedge?}{\approx}_{C} (\widetilde{t'})_n\} & \Longrightarrow & \mathtt{Pr} \cup \{t_1 \overset{\curlywedge?}{\approx}_{C} t'_1, \ldots, t_n \overset{\curlywedge?}{\approx}_{C} t'_n\} \\
\mathtt{Pr} \uplus \{[a]t \overset{\curlywedge?}{\approx}_{C} [a]t'\} & \Longrightarrow & \mathtt{Pr} \cup \{t \overset{\curlywedge?}{\approx}_{C} t'\} \\
\mathtt{Pr} \uplus \{[a]t \overset{\curlywedge?}{\approx}_{C} [b]s\} & \Longrightarrow & \mathtt{Pr} \cup \{t \overset{\curlywedge?}{\approx}_{C} (a\ b) \cdot s, \overline{(a\ c_1) \curlywedge^{?}_{C} s,} \\
 & & \qquad \overline{(c_1\ c_2) \curlywedge^{?}_{C} \mathtt{Var}(s)}\} \\
\mathtt{Pr} \uplus \{\pi \cdot X \overset{\curlywedge?}{\approx}_{C} \pi' \cdot X\} & \Longrightarrow & \mathtt{Pr}\ \cup \{(\pi')^{-1} \circ \pi \curlywedge^{?}_{C} X\} \\
\mathtt{Pr} \uplus \{\pi \cdot X \overset{\curlywedge?}{\approx}_{C} t\} & \overset{[X \mapsto pi^{-1}.t]}{\Longrightarrow} & \mathtt{Pr}\{X \mapsto \pi^{-1}.t\}, \text{ if } X \notin \mathtt{Var}(t)
\end{array}
$$

Properties

- Termination: There is no infinite chain of reductions $\Longrightarrow_C$ starting from a C-unification problem $\mathtt{Pr}$.
- Soundess and Completeness
- Nominal C Unification is finitary if solutions are represented as pairs of fixed-point context and substitution

Exercise

Show that all the simplification rules, except the instantiation rules, preserve solutions.

Generalisation

Associativity (A), AC Theories

Checking α-equality modulo A, C, AC:
Formalisation in Coq [de Carvalho et al]

C-Unification implemented in OCaml

Conclusion

- Nominal Terms: first-order syntax with binders.
- Nominal unification is quadratic (unknown lower bound) [Levy&Villaret, Calvès & F.]
- Nominal unification is used in the language α-Prolog [Cheney & Urban]
- Nominal matching is linear, equivariant matching is linear with closed rules.
- Applications in functional and logic programming languages, theorem provers, model checkers (eg. FreshML, AlphaProlog, AlphaCheck, Nominal package in Isabelle-HOL, etc.).
- Extensions: AC-Nominal Unification, E-Nominal Unification, Nominal Narrowing [Ayala-Rincón et al]
- Implementations/Formalisations: in OCaML, Haskell, Coq, Isabelle-HOL, PVS

Conclusion

- NRSs are first-order systems with built-in α-equivalence: first-order substitutions, matching modulo α.
- Closed NRSs have the expressive power of higher-order rewriting.
  Capture-avoiding atom substitutions are easy to define using freshness. They can also be included as primitive BUT unification becomes undecidable [Dominguez&F.]
- Closed NRSs have the properties of first-order rewriting (critical pair lemma, orthogonality, completion).
- Intersection types can be added to give semantics to terms and to obtain sufficient conditions for termination.
- Hindley-Milner style types [Fairweather&F.]: Typing is decidable and there are principal types, α-equivalence preserves types.
  Sufficient conditions for Subject Reduction (rewriting preserves types).