Unification modulo equational theories in languages with binding operators

Maribel Fernández

King's College London

University of Brasilia
May 2024

Thanks

- 
  Mauricio Ayala-Rincón
- 
  Ali Khan Caires Santos
- 
  Christophe Calvès
- 
  Washington Carvalho-Segundo
- 
  James Cheney
- 
  Jesús Domínguez
- 
  Elliot Fairweather
- 
  Jamie Gabbay
- 
  Temur Kutsia
- 
  José Meseguer
- 
  Daniele Nantes-Sobrinho
- 
  Andy Pitts
- 
  Ana Rocha-Oliveira
- 
  Daniella Santaguida
- 
  Gabriel Silva
- 
  Deivid Vale

Overview

- Languages with binders: α-equivalence
- Nominal logic
- Nominal terms: unification and matching modulo α
- Equational axioms: AC operators
- Nominal rewriting (modulo α and other axioms)

References

1. A. Pitts. *Nominal Logic*. Information and Computation 183, 2003.
2. C. Urban, A. Pitts, M.J. Gabbay. *Nominal Unification*. Theoretical Computer Science 323, 2004.
3. M. Fernández, M.J. Gabbay. *Nominal Rewriting*. Information and Computation 205, 2007.
4. C. Calvès, M. Fernández. *Matching and Alpha-Equivalence Check for Nominal Terms*. J. Computer and System Sciences, 2010.
5. M. Ayala-Rincón, W. de Carvalho-Segundo, M. Fernández, D. Nantes-Sobrinho, A. Rocha Oliveira. *A Formalisation of Nominal Alpha-Equivalence with A, C and AC Function Symbols*. Theoretical Computer Science, 2019.
6. M. Ayala-Rincón, M. Fernández, T. Kutsia, D. Nantes-Sobrinho, G. Silva. *Nominal AC-Matching*. CICM 2023.

Binding operators: Examples (informally)

- Operational semantics:

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

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

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

- 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 specify and reason with binding operators?
There are several alternatives.

First-order frameworks

encode α-equivalence:

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

Encoding

Decoding

Higher-order frameworks

λ-calculus meta-language, built-in α-equivalence

Examples:

- Combinatory Reduction Systems [Klop 80]
  β-rule:

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

- Higher-Order Abstract Syntax [Pfenning, Elliott 88]

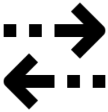
$$\text{let } a = N \text{ in } M(a) \;\longrightarrow\; (\text{fun } a \to M(a))N$$

Higher-order frameworks

- The syntax includes binders (+)
- Implicit α-equivalence (+)
- We targeted α but now we have to deal with β too (-)
- Unification is undecidable in general [Huet 75] (-)
- Interesting fragments are decidable [Miller 90] (+)

Nominal Approach [Gabbay-Pitts 99]

Key ideas:



- Names, which can be swapped
- abstraction
- freshness

Based on Nominal Set Theory [Fraenkel, Mostowski 1920-40]

Nominal Logic [Pitts03]

a sorted first-order logic theory:

$$
\begin{array}{cr}
(a\ a)x = x & (S1)\\
(a\ a')(a\ a')x = x & (S2)\\
(a\ a')a = a' & (S3)\\
(a\ a')(b\ b')x = ((a\ a')b\ (a\ a')b')(a\ a')x & (E1)\\
b \# x \Rightarrow (a\ a')b \# (a\ a')x & (E2)\\
(a\ a')f(\vec{x}) = f((a\ a')\vec{x}) & (E3)\\
p(\vec{x}) \Rightarrow p((a\ a')\vec{x}) & (E4)\\
(b\ b')[a]x = [(b\ b')a](b\ b')x & (E5)\\
a \# x \wedge a' \# x \Rightarrow (a\ a')x = x & (F1)\\
a \# a' \iff a \neq a' & (F2)\\
\forall a : ns, a' : ns'.a \# a' \qquad (ns \neq ns') & (F3)\\
\forall \vec{x}.\exists a.a \# \vec{x} & (F4)\\
[a]x = [a']x' \iff (a = a' \wedge x = x') \vee (a \# x' \wedge (a\ a')x = x) & (A1)\\
\forall x : [ns]s.\exists a : ns, y : s.x = [a]y & (A2)
\end{array}
$$

Nominal Languages

Freshness conditions $a\#t$, name swapping $(a\ b) \cdot t$, abstraction [a]t

- 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

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).
$\pi \cdot t$: π acts on t, permutes names, suspends on variables.
$(a\ b) \cdot a = b$, $(a\ b) \cdot b = a$, $(a\ b) \cdot c = c$
$Id \cdot X$ written as X.

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).
  $\pi \cdot t$: π acts on t, permutes names, suspends on variables.
  $(a\ b) \cdot a = b$, $(a\ b) \cdot b = a$, $(a\ b) \cdot c = c$
  $Id \cdot X$ written as X.
- 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 $a \notin \mathrm{fv}(X)$ 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

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

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

Also defined by induction:

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

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

Computing with Nominal Terms

Nominal rewriting: rewriting with nominal terms.

Rewrite rules specify:

- equational theories
- algebraic specifications of operators and data structures
- operational semantics of programs
- functions, processes...

Nominal Rewriting Rules:

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

Example: Prenex Normal Forms

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

Rewriting relation generated by $R = \nabla \vdash l \to r$: $\Delta \vdash s \xrightarrow{R} t$

s **rewrites with** R **to** t **in the context** Δ when:

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

Example

Beta-reduction in the Lambda-calculus:

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

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

To implement rewriting (functional/logic programming) we need a matching/unification algorithm.
Recall:

- There are efficient algorithms (linear time) for first-order terms
- Here we need more powerful algorithms: α-equivalence
- Higher-order unification is undecidable

Computing with Nominal Terms - Unification/Matching

To implement rewriting (functional/logic programming) we need a matching/unification algorithm.

Recall:

- There are efficient algorithms (linear time) for first-order terms
- Here we need more powerful algorithms: α-equivalence
- Higher-order unification is undecidable

Nominal terms have good computational properties:

- Nominal unification is decidable and unitary
- Efficient algorithms: α-equivalence, matching, unification

Checking α-equivalence

The α-equivalence derivation rules become simplification rules

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

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

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

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

(t ground or variables disjoint from l)

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

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

  (t ground or variables disjoint from l)

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

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

(t ground or variables disjoint from l)

- 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.

Nominal Matching

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

- Complexity:

  Alpha-equivalence check: linear if right-hand sides of constraints are ground. Otherwise, log-linear.

  Matching: linear in the ground case, quadratic in the non-ground case

Complexity - 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.

For more details on the implementation see [4],
see [6] for formalisations in Coq and PVS

Equivariance:
Rules defined modulo permutative renamings of atoms.

Beta-reduction in the Lambda-calculus:

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

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.

Programming and Verification

"Nominal" Programming Languages:

- Fresh-ML, CαML, Nominal Haskell, ...
- α-Prolog, α-Kanren, ...

Verification: Nominal packages for Isabelle, Agda, Coq, PVS, ...

Rely on nominal matching and unification

Programming and Verification

"Nominal" Programming Languages:

- Fresh-ML, CαML, Nominal Haskell, ...
- α-Prolog, α-Kanren, ...

Verification: Nominal packages for Isabelle, Agda, Coq, PVS, ...

Rely on nominal matching and unification

Rewriting-based programming anguages and verification frameworks?
$\Longrightarrow$ "Modulo" ... axioms

Not all term constructors are free!

Data Types: Set, Multi-set, List...

A, C, U axioms involving constructors

Not all term constructors are free!

Data Types: Set, Multi-set, List...

A, C, U axioms involving constructors

Operators obey axioms:

- OR, AND
- $||$ and $+$ in the π-calculus

$\Rightarrow$ rewriting modulo axioms, E-unification...

E-Unification

First Order E-Unification problem:
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, . . .

E-Unification

First Order E-Unification problem:
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, . . .

Question:
Unification modulo α + E?

First Order E-Unification problem:
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, ...

Question:
Unification modulo α + E?
Nominal Narrowing - enumerates solutions [FSCD 2016]

E-Unification

First Order E-Unification problem:
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, . . .

Question:
Unification modulo α + E?
Nominal Narrowing - enumerates solutions [FSCD 2016]
Question:
Nominal C- unification, Nominal AC- Unification ??

Unification modulo $\alpha + \mathsf{C}$

Unification modulo α and unification modulo C are finitary, but . . .

Unification modulo $\alpha + \mathsf{C}$

Unification modulo α and unification modulo C are finitary, but ...

$$q(a) \ \mathtt{OR} \ p(X) \ \approx_{\alpha,C} \ p((a\ b)\cdot X) \ \mathtt{OR} \ q(a)$$

$$\Downarrow$$

$$q(a) \approx_{\alpha} q(a), \ p((a\ b)\cdot X) \approx_{\alpha,C} p(X)$$

$$\Downarrow$$

$$p((a\ b)\cdot X) \approx_{\alpha,C} p(X)$$

$$\Downarrow$$

$$(a\ b)\cdot X \approx_{\alpha,C} X$$

Solutions:
$X \mapsto p(a) \ \mathtt{OR} \ p(b), \quad X \mapsto (p(a) \ \mathtt{OR} \ p(b)) \ \mathtt{OR} \ (p(a) \ \mathtt{OR} \ p(b)), \ldots$
Not finitary
[LOPSTR 2017, 2019]

Binders as well as A, C and AC operators

- $\alpha + \{C, A, AC\}$: Decidable Equivalence, formalised in PVS [6]
- Nominal C-Matching Algorithm (Finitary)
- Nominal C-Unification Procedure:
  1. Simplification phase:
     Build a derivation tree (branching for C symbols)
  2. Enumerate solutions for fixed point constraints $X \approx_{\alpha,C} \pi \cdot X$

Binders as well as A, C and AC operators

- $\alpha + \{C, A, AC\}$: Decidable Equivalence, formalised in PVS [6]
- Nominal C-Matching Algorithm (Finitary)
- Nominal C-Unification Procedure:
  1. Simplification phase:
     Build a derivation tree (branching for C symbols)
  2. Enumerate solutions for fixed point constraints $X \approx_{\alpha,C} \pi \cdot X$

Nominal C-unification is NOT finitary, if we represent solutions using substitutions/freshness:
$X \approx_{\alpha,C} (a\ b) \cdot X$ has infinite most general solutions

Binders as well as A, C and AC operators

- $\alpha + \{C, A, AC\}$: Decidable Equivalence, formalised in PVS [6]
- Nominal C-Matching Algorithm (Finitary)
- Nominal C-Unification Procedure:
  1. Simplification phase:
     Build a derivation tree (branching for C symbols)
  2. Enumerate solutions for fixed point constraints $X \approx_{\alpha,C} \pi \cdot X$

Nominal C-unification is NOT finitary, if we represent solutions using substitutions/freshness:
$X \approx_{\alpha,C} (a\ b) \cdot X$ has infinite most general solutions

Alternative representation: fixed-point constraints instead of freshness constraints:
$\pi \curlywedge x \Leftrightarrow \pi \cdot x = x$

Using fixed-point constraints nominal C-unification is finitary.

Nominal AC-Matching - Formalised in PVS [CICM 2023]

Nominal AC-Unification - work in progress

Applications:
Nominal extensions of prog. languages and verification tools:

Maude: first-order rewrite-based language [Meseguer 90]
K: first-order verification framework to specify and implement programming languages [Rosu 2017].

Nominal techniques in Maude and K

K: successful first-order verification framework to specify and implement programming languages [Rosu 2017].

Maude: popular first-order rewrite-based language [Meseguer 90]

But binders are not a primitive notion.

K: successful first-order verification framework to specify and implement programming languages [Rosu 2017].

Maude: popular first-order rewrite-based language [Meseguer 90]

But binders are not a primitive notion.
Aim:
Combine Matching Logic (K's foundation) and Rewriting Logic (Maude's foundation) with Nominal Logic to specify and reason about binding.

Matching Logic - Syntax

Signature $\mathbf{\Sigma} = (S, \mathcal{V}ar, \Sigma)$

Patterns:

$$\phi_\tau ::= x\colon \tau \mid \phi_\tau \wedge \psi_\tau \mid \neg\phi_\tau \mid \exists x\colon \tau'.\phi_\tau \mid \sigma(\phi_{\tau_1}, \ldots, \phi_{\tau_n})$$

where $x \in \mathcal{V}ar_\tau$ and $\sigma \in \Sigma_{\tau_1,\ldots,\tau_n;\tau}$.

Disjunction, implication, $\forall$, true and false defined as abbreviations: e.g. $\top_\tau \equiv \exists x\colon \tau.x\colon \tau$ and $\bot_\tau \equiv \neg\top_\tau$.

Matching Logic - Model

Valuation $\rho \colon \mathcal{V}ar \to M$ respecting sorts.
Extension to patterns:
$\overline{\rho}(x) = \{\rho(x)\}$ for all $x \in \mathcal{V}ar$, $\overline{\rho}(\phi_1 \wedge \phi_2) = \overline{\rho}(\phi_1) \cap \overline{\rho}(\phi_2)$, $\overline{\rho}(\neg\phi_\tau) = M_\tau - \overline{\rho}(\phi_\tau)$, $\overline{\rho}(\exists x \colon \tau'.\phi_\tau) = \bigcup_{a \in M_{\tau'}} \overline{\rho[a/x]}(\phi_\tau)$, $\overline{\rho}(\sigma(\phi_{\tau_1}, \ldots, \phi_{\tau_n}) = \overline{\sigma_M}(\overline{\rho}(\phi_{\tau_1}), \ldots, \overline{\rho}(\phi_{\tau_n}))$, for $\sigma \in \Sigma_{\tau_1,\ldots,\tau_n;\tau}$, where $\overline{\sigma_M}(V_1, \ldots, V_n) = \bigcup\{\sigma_M(v_1, \ldots, v_n) \mid v_1 \in V_1, \ldots, v_n \in V_n\}$.

ϕ_τ valid in M, $M \vDash \phi_\tau$, if $\overline{\rho}(\phi_\tau) = M_\tau$ for all $\rho \colon Var \to M$.

Adding NL Features to ML

1. Nominal Logic can be embbeded as a Matching Logic Theory: NLML (see [PPDP 2022])
   $\Rightarrow$ it can be directly implemented in K
   But...
   - ground names, which are useful in rewriting, logic programming and program verification, are not available in NLML
   - not clear how to incorporate the N-quantifier in a first-class way, which is needed to simplify reasoning with freshness constraints.
2. NML: Matching Logic with Built-in Names and N

Matching Logic with Built-in Names and И

NML signature $\mathbf{\Sigma} = (S, \mathcal{V}ar, Name, \Sigma)$ consists of

- a non-empty set S of sorts $\tau, \tau_1, \tau_2 \ldots$, split into a set NS of name sorts $\alpha, \alpha_1, \alpha_2, \ldots$, a set DS of data sorts $\delta, \delta_1, \delta_2, \ldots$ including a sort $Pred$, and a set AS of abstraction sorts $[\alpha]\tau$
- an S-indexed family $\mathcal{V}ar = \{\mathcal{V}ar_\tau \mid \tau \in S\}$ of countable sets of variables $x\colon \tau, y\colon \tau, \ldots$,
- an NS-indexed family $Name = \{Name_\alpha \mid \alpha \in NS\}$ of countable sets of names $\mathsf{a}\colon \alpha, \mathsf{b}\colon \alpha, \ldots$ and
- an $(S^* \times S)$-indexed family Σ of sets of many-sorted symbols σ, written $\Sigma_{\tau_1,\ldots,\tau_n;\tau}$.

Patterns:

$$\begin{array}{rcl}\phi_\tau & ::= & x:\tau \mid a:\alpha \mid \phi_\tau \wedge \psi_\tau \mid \neg\phi_\tau \mid \exists x:\tau'.\phi_\tau \\ & & \mid \sigma(\phi_{\tau_1},\ldots,\phi_{\tau_n}) \mid \text{И}a:\alpha.\phi_\tau\end{array}$$

where $x \in \mathcal{V}ar_\tau$, $a \in Name_\alpha$, and both $\exists$ and И are binders (i.e., we work modulo α-equivalence).

Σ includes the following families of sort-indexed symbols (subscripts omitted):

$$\begin{array}{rcll}(-\ -)\cdot - & : & \alpha \times \alpha \times \tau \to \tau & \text{swapping (function)} \\ [-]- & : & \alpha \times \tau \to [\alpha]\tau & \text{abstraction (function)} \\ -@- & : & [\alpha]\tau \times \alpha \rightharpoonup \tau & \text{concretion (partial function)} \\ fresh_{\tau,\alpha} & \in & \Sigma_{\tau;\alpha} & \text{freshness (multivalued operation)} \\ -\ \#_{\alpha,\tau}\ - & : & \alpha \times \tau \rightharpoonup Pred & \text{freshness relation} \\ -^\dagger & : & \Sigma_{Pred;\tau} & \text{coercion operator, often left implicit.}\end{array}$$

NML Model

Given $\mathbf{\Sigma} = (S, \mathcal{V}ar, Name, \Sigma)$
let $\mathbb{A}$ be $\bigcup_{\alpha \in NS} \mathbb{A}_\alpha$ where each $\mathbb{A}_\alpha$ is an infinite countable set of atoms and the $\mathbb{A}_\alpha$ are pairwise disjoint,
let G be a product of permutation groups $\prod_i Sym(\mathbb{A}_i)$

An NML model $M = (\{M_\tau\}_{\tau \in S}, \{\sigma_M\}_{\sigma \in \Sigma})$ consists of

- a non-empty nominal G-set M_τ for each $\tau \in S - NS$;
- an equivariant interpretation $\sigma_M : M_{\tau_1} \times \cdots \times M_{\tau_n} \to \mathcal{P}_{fin}(M_\tau)$ for each $\sigma \in \Sigma_{\tau_1,\ldots,\tau_n;\tau}$.

NML Model

A model is *standard* if the interpretation of:

1. each name sort α is $\mathbb{A}_\alpha$
2. the sort *Pred* is a singleton set $\{*\}$, where $*$ is equivariant: $\{*\}$ is a nominal set whose powerset is isomorphic to Bool
3. each abstraction sort $[\alpha]\tau$ is $[M_\alpha]M_\tau$
4. the swapping symbol $(- \ -) \cdot -\colon \alpha \times \alpha \times \tau \to \tau$ is the swapping function on M_τ
5. the abstraction symbol is the quotienting function mapping $\langle a, x\rangle$ to its alpha-equivalence class, i.e. $(a, x) \mapsto (a, x)/_{\equiv_\alpha}$
6. the concretion symbol is the (partial) concretion function $(X, \mathrm{a}) \mapsto \{y \mid (\mathrm{a}, y) \in X\}$
7. the freshness operation $fresh_{\tau,\alpha}$ is the function $x \mapsto \{a \mid a \notin supp(x)\}$
8. the freshness relation $\#_{\alpha,s}$ is the freshness predicate on $\mathbb{A}_\alpha \times M_\tau$, i.e., it holds for the tuples $\{(a, x) \mid a \notin supp(x)\}$.

NML Pattern Semantics

Given valuation ρ whose domain includes the free variables and free names of ϕ:

$$
\begin{array}{rcl}
\overline{\rho}(x:\tau) & = & \{\rho(x)\} \\
\overline{\rho}(\mathsf{a}:\alpha) & = & \{\rho(\mathsf{a})\} \\
\overline{\rho}(\sigma(\phi_1,\ldots,\phi_n)) & = & \overline{\sigma_M}(\overline{\rho}(\phi_1),\ldots,\overline{\rho}(\phi_n)) \\
\overline{\rho}(\phi_1 \wedge \phi_2) & = & \overline{\rho}(\phi_1) \cap \overline{\rho}(\phi_2) \\
\overline{\rho}(\neg\phi) & = & M_\tau - \overline{\rho}(\phi) \\
\overline{\rho}(\exists x:\tau.\phi) & = & \bigcup\limits_{a \in M_\tau} \overline{\rho[a/x]}(\phi) \\
\overline{\rho}(\mathsf{N}\mathsf{a}:\alpha.\phi) & = & \bigcup\limits_{a \in \mathbb{A}_\alpha - supp(\rho)} \{v \in \overline{\rho[a/\mathsf{a}]}(\phi) \mid a \notin supp(v)\}
\end{array}
$$

In the interpretation of the N pattern, ρ is extended by assigning to a any fresh element a of $\mathbb{A}_\alpha$

Consider three possible rules representing eta-equivalence for the lambda-calculus

$$
\begin{array}{rcl}
x : Exp & = & lam([\mathrm{a}]app(x, var(\mathrm{a}))) \\
x : Exp & = & lam(\exists a.[a]app(x, var(a))) \\
x : Exp & = & lam(\text{И}\mathrm{a}.[\mathrm{a}]app(x, var(\mathrm{a})))
\end{array}
$$

Only the third one is correct.

Applications: The λ-calculus in NML

To reason about the typed lambda-calculus we use sorts *Exp* (expressions), *Ty* (types), and *Var* (variables, a name-sort) interpreted as nominal sets M_{Var}, M_{Exp}, and M_{Ty} satisfying the following equations:

$$M_{Exp} = M_{Var} + (M_{Exp} \times M_{Exp}) + [M_{Var}]M_{Exp}$$

$$M_{Ty} = 1 + M_{Ty} \times M_{Ty} + \cdots$$

We assume at least one constant type (e.g. *int* or *unit*) and a binary constructor $fn : Ty \times Ty \to Ty$ for function types

M_{Exp} is the set of lambda-terms quotiented by alpha-equivalence.
We fix M_{Λ} as the standard model obtained taking M_{Exp} and M_{Ty} as defined above.

The λ-calculus in NML

In NML we can axiomatize substitution equationally (no side condition)

$$\begin{aligned}
subst(var(a), a, z) &= z \\
subst(var(a), \neg a, z) &= var(a) \\
subst(app(x_1, x_2), y, z) &= app(subst(x_1, y, z), subst(x_2, y, z)) \\
subst(lam(x), y, z) &= lam(\text{И}\mathsf{a}.[\mathsf{a}]subst(x@\mathsf{a}, y, z))
\end{aligned}$$

The λ-calculus in NML

Induction principle using $\text{\reflectbox{N}}$ avoiding freshness constraints

$$
\begin{array}{rl}
(\forall x : Var.P(var(x))) & \Rightarrow \\
(\forall t_1 : Exp, t_2 : Exp.P(t_1) \wedge P(t_2) \Rightarrow P(app(t_1, t_2))) & \Rightarrow \\
(\forall t : [Var]Exp.\text{\reflectbox{N}}\mathrm{a} : Var.P(t@\mathrm{a}) \Rightarrow P(lam(t)) & \Rightarrow \\
\forall t : Exp.P(t) &
\end{array}
$$

Substitution Lemma (with just one freshness condition, formalizing the usual side-condition in textbooks)

$$
\begin{array}{rl}
\mathrm{a} \# z' \quad \Rightarrow & subst(subst(x, \mathrm{a}, z), \mathrm{b}, z') = \\
& subst(subst(x, \mathrm{b}, z'), \mathrm{a}, subst(z, \mathrm{b}, z'))
\end{array}
$$

Rewriting Logic

A rewrite theory is a tuple

$$\mathcal{R} = (\Sigma, E, \phi, R)$$

where

- (Σ, E) is an equational theory with order-sorted signature Σ consisting of sorts $(S, <)$ and function symbols F, and Σ-equations E,
- R is a set of (possibly conditional) rewrite rules,
- $\phi : \Sigma \to \mathbb{N}^*$ is a so-called *frozenness map* indicating, for each function symbol $f \in \Sigma$, its *frozen* argument positions, where rewriting with rules R is forbidden.

Two requirements: (i) *countably infinite* supply of names (ii) an equality predicate

Specification in Maude:
NAME (conditional) equational theory with *initiality constraints* on subtheories [1]

theory $NAME$ **protects** $NAT, BOOL$
sort $Name$
functions : $i : Name \rightarrow Nat, \quad j : Nat \rightarrow Name, \quad \_ . = . \_ : Name \ Name \rightarrow Bool$
vars $a, b : Name, \quad n : Nat$
equations :
$a . = . a = true, \quad a . = . b = true \Rightarrow a = b, \quad j(i(a)) = a, \quad i(j(n)) = n$
endtheory

[1] the *reduct* $\mathbb{A}|_{\Sigma_0}$ of a *NAME*-algebra $\mathbb{A}$ to any subtheory $T_0 = (\Sigma_0, E_0)$ of it having an initiality constraint must be isomorphic to the initial T_0-algebra $\mathbb{T}_{\Sigma_0/E_0}$

Definition

A *Binder Signature* is a pair of an order-sorted signature $\Sigma = ((S, <), F)$ and a function β with domain F.
$\beta(f)$ gives *binding information*: which *argument positions* bind which other *argument positions* in f.

For example, the *in* operator in the π-calculus binds any occurrence of the name given as second argument within the third argument, so that $\beta(in) = (2, 3)$. Similarly, in the λ-calculus $\beta(\lambda\_.\_) = (1, 2)$. For non-binding operators like *out* in the π-calculus we have $\beta(out) = \epsilon$.

Constraints on Beta

The signature is *parametric* on one or more copies of the $NAME$ parameter theory: $Name_1, \ldots, Name_k$ are the corresponding parameter sorts in those copies of $NAME$.

Three kinds of binding relationships: (i) binding a *single* name;
(ii) binding a *tuple* of names; and
(iii) binding a *non-empty* (Ne) *list* of names.
$Name_i < m.Tuple_i < NeList_i < List_i$

Any calculus $\mathscr{C}$ with binders has an associated *structural congruence*

$$E_{\mathscr{C}} = E_{\mathscr{C}}^{\alpha} \cup E_{\mathscr{C}}^{cs} \cup E_{\mathscr{C}}^{aux}$$

where the equations
$E_{\mathscr{C}}^{\alpha}$ define a calculus-generic α-equivalence relation,
$E_{\mathscr{C}}^{cs}$ are *calculus-specific* equivalences,
$E_{\mathscr{C}}^{aux}$ are other calculus-generic equations defining *auxiliary functions*, e.g,. name swapping, a freshness predicate, renaming or substitution operations

Not all calculi need all these auxiliary equations. For example, in the π-calculus *renaming* (as opposed to substitution) equations are needed.

Examples

Swapping:

$$(a\ b) \cdot f(t_1, \ldots, t_n) = f((a\ b) \cdot t_1, \ldots, (a\ b) \cdot t_n)$$

Freshness:
$\_ \# \_ : Name_i\ B \to Bool$ indicates whether a in $Name_i$ is fresh in a term of sort B.
There are three cases: the term in the second argument is a name b in $Name_i$, is rooted by a binding operator (wlg assume $f : List_1\ \bar{B}_1 \ldots List_k\ \bar{B}_k\ \bar{B}_{k+1} \to C$, where for $1 \leq i \leq k$, each $List_i$ is a name-list sort, which binds all sorts in the next sequence of sorts $\bar{B}_i$, and that all neither bound nor binding sorts are exactly those in the sort list $\bar{B}_{k+1}$) or by a non-binding operator g (including constants g such as names in $Name_j$ with $i \neq j$):

$$\begin{array}{lcl}
a \# b & = & not(a .=. b) \\
a \# f(L_1, \bar{t_1}, \ldots, L_k, \bar{t_k}, \bar{u}) & = & (a \in L_1 \vee a \# \bar{t_1}) \wedge \ldots \\
 & & \wedge(a \in L_k \vee a \# \bar{t_k}) \wedge a \# \bar{u} \\
a \# g(\bar{u}) & = & a \# \bar{u}
\end{array}$$

Specified by a rewrite relation $\rightarrow^{\phi}_{R/E_{\mathscr{C}}}$ on $\Sigma_{\mathscr{C}}$-terms: rewriting *modulo* the equations $E_{\mathscr{C}}$, forbidding reductions at certain *frozen* positions.

Definition

$u \rightarrow^{\phi}_{R/E_{\mathscr{C}}} v$ iff there exist u', v' such that:
(i) $u =_{E_{\mathscr{C}}} u'$ and $v =_{E_{\mathscr{C}}} v'$, and
(ii) $u' \rightarrow^{\phi}_{R} v'$, where the relation $\rightarrow^{\phi}_{R}$ restricts the standard *term-rewriting* relation $\rightarrow_R$ by forbidding rewriting with R at frozen positions (i.e., if f is a function symbol at position p and $i \in \phi(f)$ then rewriting is forbidden at any position piq)

Example, in the π-calculus the react rule cannot apply inside a prefix *in*, so $\phi(in) = \{1, 2, 3\}$.
For executability: Matching modulo $E_{\mathscr{C}}$ is required (cf nominal AC matching).
For verification tasks: Unification modulo $E_{\mathscr{C}}$ is required

Conclusions

Summary:

- Nominal Rewriting Systems [PPDP 2004]:
  first-order rewriting modulo α, based on Nominal Logic
- Closed NRS $\Leftrightarrow$ higher-order rewriting systems
  Capture-avoiding atom substitution easy to define.
- Nominal matching is linear, equivariant matching is linear with closed rules
- Nominal unification is quadratic (unknown lower bound) [LOPSTR 2010]
- Hindley-Milner style types: principal types, α-equivalence preserves types.
  Sufficient conditions for Subject Reduction.
- Applications: functional and logic programming languages, theorem provers, model checkers
  FreshML, AlphaProlog, AlphaCheck, Nominal package in Isabelle-HOL, ...

Conclusions

- Extensions: Nominal E-Unification, Nominal Narrowing, Nominal C-Unification [LOPSTR 2017,2019]
- Being first-order, nominal logic is a natural candidate for supporting binding in
  - Matching Logic (K) - see [PPDP 2022]
  - Rewriting Logic (Maude) - uses E-unification (A, C, AC,...)

Implementations/formalisations

Nominal Datatype Package for Haskell (Jamie Gabbay):
`https://github.com/bellissimogiorno/nominal`

Nominal Project, University of Brasilia: `http://nominal.cic.unb.br`

alpha-Prolog (James Cheney, Christian Urban):
`https://homepages.inf.ed.ac.uk/jcheney/programs/aprolog/`

Nominal Isabelle (Christian Urban)

Questions?