Unification Algorithms

Maribel Fernández

King's College London

December 2007

Motivations

Unification algorithms have become popular in recent years, due to their key role in the fields of logic programming and theorem proving.

Logic Programming Languages

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

Prolog is one of the most popular logic programming languages.
Prolog = Clausal Logic + Resolution + Control Strategy

Prolog

- Knowledge-based programming: the program just describes the problem.
- *Declarative* programming: the program says *what* should be computed, rather than *how* it is computed (although this is not true for impure languages).
- Precise and simple semantics.
- The same program can be used in many different ways, thanks to the use of UNIFICATION.

Example:
SWI Prolog (Free Software Prolog compiler) developed at the University of Amsterdam, `http://www.swi-prolog.org/`

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

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

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

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

In Prolog no specific alphabet is asssumed, the programmer can freely choose the names of functions (but there are some built-in functions with specific meanings, e.g. arithmetic operations).

Prolog Programs

Prolog programs are sets of *definite clauses* (or *Horn clauses*).
A *definite clause* is a disjunction of literals with at most one positive literal.
A *literal* is an atomic formula or a negated atomic formula.
To build atomic formulas we use terms and *predicate symbols* (with fixed arities):
If p is a predicate of arity n and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is an *atomic formula*, or simply an *atom.*

Example:

`value(number(1),1)`, $\neg$`raining`

are literals, where we use the binary predicate `value` and 0-ary predicate `raining`; `number` is a unary function.

Definite Clauses

A definite clause $P_1 \vee \neg P_2 \vee \ldots \vee \neg P_n$ (where P_1 is the only positive literal) will be written:

$$P_1 \;\texttt{:-}\; P_2, \ldots, P_n.$$

and we read it as: "P_1 if P_2 and ... and P_n"
If the clause contains just P_1 and no negative atoms, then we write

$$P_1.$$

Both kinds of clauses are called *Program Clauses*, and the second kind is called a *Fact*.
If the clause contains only negative literals, we call it a *Goal* or *Query* and write

$$\texttt{:-}P_2, \ldots, P_n.$$

Example - Horn Clauses

```
based(prolog,logic).
based(haskell,functions).
likes(claire,functions).
likes(max,logic).
likes(X,P) :- based(P,Y), likes(X,Y).
```

The first four clauses are facts, the last clause is a rule. The following is a goal:

```
:- likes(Z,prolog).
```

Prolog programs

A list of program clauses in Prolog can be seen as the definition of a series of predicates. For instance, in the program

```
based(prolog,logic).
based(haskell,maths).
likes(max,logic).
likes(claire,maths).
likes(X,P) :- based(P,Y), likes(X,Y).
```

we are defining the predicates `likes` and `based`.

Prolog programs

In the program

```
append([],L,L).
append([X|L],Y,[X|Z]) :- append(L,Y,Z).
```

the atomic formula `append(S,T,U)` expresses that the result of appending the list T onto the end of list S is the list U.
Any term of the form `[X|T]` denotes a list where the first element is X (*the head of the list*) and T is the rest of the list (also called *the tail of the list*). The constant `[]` denotes the empty list. We abbreviate `[X|[Y|[]]]` as `[X,Y]`.
Goals such as:

```
:- append([0],[1,2],U)
:- append(X,[1,2],U)
:- append([1,2],X,[0])
```

are questions to be solved using the program.

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

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

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

```
append([X|L],Y,[X|Z]) :- append(L,Y,Z).
```

The substitution
{`X` ↦ `0`, `L` ↦ `[]`, `Y` ↦`[1,2]`, `U` ↦ `[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` ↦ `[1,2]`}.
Thus, {`U` ↦ `[0,1,2]`} is an **answer substitution**.

This method is based on the Principle of Resolution.

Unification is a key step in the Principle of Resolution.
History:
The unification algorithm was first sketched by Jacques Herbrand in his thesis (in 1930).

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

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

The version of the unification algorithm that we present is based on work by Martelli and Montanari (1982).

Unification

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

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

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

Martelli and Montanari's algorithm finds the most general unifier for a unification problem if a solution exists, otherwise it fails, indicating that there are no solutions.

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

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

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.

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

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:

We start with $\{f(a,a) = f(X,a)\}$:

- using rule (1) it rewrites to $\{a = X, a = a\}$,
- using rule (4) we get $\{X = a, a = a\}$,
- using rule (1) again we get $\{X = a\}$.

Now no rule can be applied, the algorithm terminates with the most general unifier $\{X \mapsto a\}$

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]}`

The Principle of Resolution

In order to solve a query
`:-` $A_1, \ldots, A_n$
with respect to a set P of program clauses, resolution seeks to show that P, $\neg A_1, \ldots, \neg A_n$ leads to a contradiction. It is based on *refutation*.

A contradiction is obtained when a literal and its negation are stated at the same time: A, $\neg A$.

If a contradiction does not arise directly, new literals are derived by resolution using the clauses, until a contradiction arises (or the search continues forever). The derived literals are called *resolvents*.

Computing Resolvents with SLD-Resolution:

If we have a query $\quad$ `:-` $\mathtt{a}(u_1, \ldots, u_n)$
and a program clause $\quad$ $\mathtt{a}(t_1, \ldots, t_n)$ `:-` $S_1, \ldots, S_m$
such that $\mathtt{a}(t_1, \ldots, t_n)$ and $\mathtt{a}(u_1, \ldots, u_n)$ are **unifiable** with mgu σ, then we obtain a resolvent: $\quad$ `:-` $S_1\sigma, \ldots, S_m\sigma$.

In general, if the query has several atoms
`:-` $A_1, \ldots, A_k$
the *resolvent* is computed between the *first* atom in the goal (A_1) and a program clause, and we obtain
`:-` $S_1\sigma, \ldots, S_m\sigma, A_2\sigma, \ldots, A_k\sigma$
Note that when we compute a resolvent using a fact ($m = 0$), the atom disappears from the query.
An empty resolvent indicates a contradiction, denoted by $\diamondsuit$. The substitution that has been computed is the answer to the original goal. The idea is to continue generating resolvents until we obtain an empty one.

Each resolution step computes a resolvent between the last resolvent obtained and a clause in the program. Prolog uses the clauses in the program in the order they are written.

When an empty resolvent is generated, the composition of all the substitutions (mgu) applied at each resolution step, restricted to the variables of the query, is the *answer* to the query.

We represent each resolution step graphically as follows:

$$\begin{array}{c} \text{Query} \\ \big|\ \text{mgu} \\ \text{Resolvent} \end{array}$$

Since there might be several clauses in the program that can be used to generate a resolvent, we obtain an *SLD-resolution tree.*

Example

Program:

```
based(prolog,logic).
based(haskell,maths).
likes(max,logic).
likes(claire,maths).
likes(X,P) :- based(P,Y), likes(X,Y).
```

Query:

```
:- likes(Z,prolog).
```

Using the last clause, and the mgu $\{X \mapsto Z, P \mapsto prolog\}$ we obtain the resolvent

```
:- based(prolog,Y), likes(Z,Y).
```

Now using the first clause and the mgu $\{Y \mapsto logic\}$ we obtain the new resolvent

```
:- likes(Z,logic).
```

We can now unify with `likes(max,logic)` using $\{Z \mapsto max\}$, and we obtain an empty resolvent (success). Answer to the initial query: $\{Z \mapsto max\}$

Example

Graphically, the SLD-resolution tree for this query contains:

```
likes(Z,prolog)
```

$| \{X \mapsto Z, P \mapsto prolog\}$

```
based(prolog,Y), likes(Z,Y)
```

$| \{Y \mapsto logic\}$

```
likes(Z,logic).
```

$\{Z \mapsto max\} \; / \quad \backslash \; \{X' \mapsto Z, P' \mapsto logic\}$

$\diamondsuit$ `based(logic,Y'),likes(Z,Y')`

Failure

SLD-resolution:

SLD-resolution using unification is complete (if there is an answer, it will eventually be generated), although Prolog's implementation is not complete (due to the use of a depth first search strategy, for efficiency reasons).

Horn clauses use first-order terms — simple but not very expressive. Extensions of the language involve extending the unification algorihtm.

More expressive languages

How do we represent binding operations? Informally:

- Operational semantics:

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

Renaming of bound variables (α-equality) is implicit.

More expressive languages

How do we represent binding operations? Informally:

- Operational semantics:

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

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

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

Renaming of bound variables (α-equality) is implicit.

More expressive languages

How do we represent binding operations? Informally:

- Operational semantics:

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

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

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

- π-calculus: $P \mid \nu a.Q \to \nu a.(P \mid Q) \qquad (a \notin \mathrm{fv}(P))$

Renaming of bound variables (α-equality) is implicit.

More expressive languages

How do we represent binding operations? Informally:

- Operational semantics:

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

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

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

- π-calculus: $P \mid \nu a.Q \to \nu a.(P \mid Q) \qquad (a \notin \mathrm{fv}(P))$
- Logic equivalences:

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

Renaming of bound variables (α-equality) is implicit.

There are several alternatives.

- Use a first-order system and encode α-equality.

Formally: Rewrite Systems

There are several alternatives.

- Use a first-order system and encode α-equality.

  $\Rightarrow$ No binders. (-)

There are several alternatives.

- Use a first-order system and encode α-equality.
  - No binders. (-)

  $\Rightarrow$ First-order matching: we need to 'specify' α-conversion. (-)

Formally: Rewrite Systems

There are several alternatives.

- Use a first-order system and encode α-equality.
  - No binders. (-)
  - First-order matching: we need to 'specify' α-conversion. (-)
  - $\Rightarrow$ Simple notion of substitution. $(+)$

Formally: Rewrite Systems

There are several alternatives.

- Use a first-order system and encode α-equality.
  - No binders. (-)
  - First-order matching: we need to 'specify' α-conversion. (-)
  - Simple notion of substitution. $(+)$
- Algebraic λ-calculi: First-order rewriting $+$ λ-calculus

Formally: Rewrite Systems

There are several alternatives.

- Use a first-order system and encode α-equality.
  - No binders. (-)
  - First-order matching: we need to 'specify' α-conversion. (-)
  - Simple notion of substitution. $(+)$
- Algebraic λ-calculi: First-order rewriting $+$ λ-calculus

  $\Rightarrow$ λ is a binder. $(+)$

There are several alternatives.

- Use a first-order system and encode α-equality.
  - No binders. (-)
  - First-order matching: we need to 'specify' α-conversion. (-)
  - Simple notion of substitution. $(+)$
- Algebraic λ-calculi: First-order rewriting $+$ λ-calculus
  - λ is a binder. $(+)$

  $\Rightarrow$ First-order matching: we need to 'specify' α-conversion. (-)

There are several alternatives.

- Use a first-order system and encode α-equality.
  - No binders. (-)
  - First-order matching: we need to 'specify' α-conversion. (-)
  - Simple notion of substitution. (+)
- Algebraic λ-calculi: First-order rewriting + λ-calculus
  - λ is a binder. (+)
  - First-order matching: we need to 'specify' α-conversion. (-)

  $\Rightarrow$ Simple notion of substitution. (+)

Higher-order frameworks

- Higher-order rewrite systems (CRS, HRS, etc.) include a general binding construct.
  Example: β-rule

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

  Then $app(lam([a]f(a, g(a)), b) \to f(b, g(b))$
  using higher-order matching.

Higher-order frameworks

- Higher-order rewrite systems (CRS, HRS, etc.) include a general binding construct.
  Example: β-rule

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

  Then $app(lam([a]f(a, g(a)), b) \to f(b, g(b))$
  using higher-order matching.
- Higher-Order Abstract Syntax:

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

Higher-order frameworks

- Higher-order rewrite systems (CRS, HRS, etc.) include a general binding construct.
  Example: β-rule

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

  Then $app(lam([a]f(a, g(a)), b) \to f(b, g(b))$
  using higher-order matching.
- Higher-Order Abstract Syntax:

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

$\Rightarrow$ Terms with binders. $(+)$

Higher-order frameworks

- Higher-order rewrite systems (CRS, HRS, etc.) include a general binding construct.
  Example: β-rule

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

  Then $app(lam([a]f(a, g(a)), b) \to f(b, g(b))$
  using higher-order matching.
- Higher-Order Abstract Syntax:

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

  - Terms with binders. $(+)$

  $\Rightarrow$ Implicit α-equivalence. $(+)$

Higher-order frameworks

- Higher-order rewrite systems (CRS, HRS, etc.) include a general binding construct.
  Example: β-rule

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

  Then $app(lam([a]f(a, g(a)), b) \to f(b, g(b))$
  using higher-order matching.
- Higher-Order Abstract Syntax:

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

  - Terms with binders. $(+)$
  - Implicit α-equivalence. $(+)$

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

Higher-order frameworks

- Higher-order rewrite systems (CRS, HRS, etc.) include a general binding construct.
  Example: β-rule

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

  Then $app(lam([a]f(a, g(a)), b) \to f(b, g(b))$
  using higher-order matching.
- Higher-Order Abstract Syntax:

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

  - Terms with 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

- Higher-order rewrite systems (CRS, HRS, etc.) include a general binding construct.
  Example: β-rule

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

  Then $app(lam([a]f(a, g(a)), b) \to f(b, g(b))$
  using higher-order matching.
- Higher-Order Abstract Syntax:

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

  - Terms with 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

- Higher-order rewrite systems (CRS, HRS, etc.) include a general binding construct.
  Example: β-rule

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

  Then $app(lam([a]f(a, g(a)), b) \to f(b, g(b))$
  using higher-order matching.
- Higher-Order Abstract Syntax:

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

  - Terms with 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. $\forall x.P$).

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

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

$\Rightarrow$ Terms with binders.

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

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

- Terms with binders.

$\Rightarrow$ Built-in α-equivalence.

Nominal Rewriting

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

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

- Terms with binders.
- Built-in α-equivalence.

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

Nominal Rewriting

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

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

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

$\Rightarrow$ Dependencies of terms on names are implicit.

Nominal Rewriting

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

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

- Terms with binders.
- Built-in α-equivalence.
- Simple notion of substitution (first order).
- Dependencies of terms on names are implicit.

$\Rightarrow$ Easy to express conditions such as $a \notin \mathrm{fv}(M)$

Nominal Syntax

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

Nominal Syntax

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

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

$Id \cdot X$ written as X.

Nominal Syntax

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

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

  $Id \cdot X$ 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

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

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

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

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

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

Freshness

Also defined by induction:

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

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

Checking α-equivalence of terms

The syntax-directed derivation rules above suggest an algorithm to check α-equivalence, using transformation rules:

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

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

Checking α-equivalence of terms

The relation $\Longrightarrow$ is confluent and strongly normalising; i.e. the simplification process terminates and the result is unique: $\langle Pr \rangle_{nf}$. If $\langle Pr \rangle_{nf}$ is a consistent freshness context, Pr is valid.

To solve equations we need to add instantiation rules:
$Pr, \pi \cdot X \approx_\alpha t \Longrightarrow Pr\{X \mapsto \pi^{-1} \cdot t\}$ if X in Pr and X not in t.

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

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

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

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

- 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

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

- 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.

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

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

- 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 is decidable, and linear in time [Calves, Fernandez 07].

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

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

- 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 is decidable, and
  linear in time [Calves, Fernandez 07].
- Nominal unification is decidable and polynomial. A solvable unification problem has a unique most general solution [Urban, Pitts, Gabbay 04].

Implementing Nominal Unification — First Approach: MAUDE

MAUDE is based on rewriting. Example program:

```
fmod LAMBDA is
sorts Var Term .
subsorts Var < Term .

op var : String -> Var .
op lam : Var Term -> Term .
op app : Term Term -> Term .

var x : Var .
var t1 t2 : Term .

rl [beta] : app(lam(x,t1),t2) => t1[t2/x]
endfm
```

sorts Var Atom Perm Term .
subsorts Atom Var $<$ Term .

op _^_ : Perm Var $->$ Term .
op [_]_ : Atom Term $->$ Term .

eq perm1 ^ (perm2 ^ var) $=$ (perm1 $\circ$ perm2) ^ var .
eq $a \# f(t) = a \# t$.

- natural choice of language since the algorithm is specified as a set of rewrite rules

Maude implementation

- natural choice of language since the algorithm is specified as a set of rewrite rules
- very easy to code (direct translation)

Maude implementation

- natural choice of language since the algorithm is specified as a set of rewrite rules
- very easy to code (direct translation)
- easy to maintain

- natural choice of language since the algorithm is specified as a set of rewrite rules
- very easy to code (direct translation)
- easy to maintain
- but it is inefficient

Maude implementation

- natural choice of language since the algorithm is specified as a set of rewrite rules
- very easy to code (direct translation)
- easy to maintain
- but it is inefficient
- even standard (first-order) unification is exponential on trees, due to copying: $f(X, X) \; {}_?{\approx}_? \; t, \quad X \; {}_?{\approx}_? \; u$

Maude implementation

- natural choice of language since the algorithm is specified as a set of rewrite rules
- very easy to code (direct translation)
- easy to maintain
- but it is inefficient
- even standard (first-order) unification is exponential on trees, due to copying: $f(X, X) \;_?{\approx}_?\; t, \quad X \;_?{\approx}_?\; u$
- we need sharing

$\Rightarrow$ atoms, variables and 1 are represented as leaves

Implementing nominal unification on DAGs

- atoms, variables and 1 are represented as leaves

$\Rightarrow$ a tuple $(t_1, \ldots, t_n)$ is represented as a node () with n children

Implementing nominal unification on DAGs

- atoms, variables and 1 are represented as leaves
- a tuple $(t_1, \ldots, t_n)$ is represented as a node () with n children

$\Rightarrow$ $f(t)$ is represented as a node f with one child.

Implementing nominal unification on DAGs

- atoms, variables and 1 are represented as leaves
- a tuple $(t_1, \ldots, t_n)$ is represented as a node () with n children
- $f(t)$ is represented as a node f with one child.

$\Rightarrow$ $[a]t$ is represented as a node [] with two children (a and t)

$\Rightarrow$ Equivalence and Freshness constraints are also represented with DAGs

- Equivalence and Freshness constraints are also represented with DAGs

$\Rightarrow$ $t \;{}_?{\approx}_?\; u$ is represented as a node ${}_?{\approx}_?$ with two children

Equivalence and Freshness in DAGs

- Equivalence and Freshness constraints are also represented with DAGs
- $t \; {}_?{\approx}_? \; u$ is represented as a node ${}_?{\approx}_?$ with two children

$\Rightarrow$ $a \; \#_? \; t$ is represented as a node $\#_?$ with two children

Equivalence and Freshness in DAGs

- Equivalence and Freshness constraints are also represented with DAGs
- $t \,{}_?{\approx}_?\, u$ is represented as a node ${}_?{\approx}_?$ with two children
- $a \#_? t$ is represented as a node $\#_?$ with two children

$\Rightarrow$ a whole unification problem is represented as a DAG

Optimisation techniques for equivalence

$\Rightarrow$ the nodes $t \ {}_?{\approx}_? \ t$ are erased

Optimisation techniques for equivalence

- the nodes $t \;{}_?{\approx}_?\; t$ are erased

$\Rightarrow$ $\pi \circ t \;{}_?{\approx}_?\; \pi' \circ t$ is replaced by $ds(\pi, \pi') \;\#_?\; t$

Optimisation techniques for equivalence

- the nodes $t \ {}_?{\approx}_? \ t$ are erased
- $\pi \circ t \ {}_?{\approx}_? \ \pi' \circ t$ is replaced by $ds(\pi, \pi') \ \#_? \ t$

$\Rightarrow$ $t \ {}_?{\approx}_? \ X$ or $X \ {}_?{\approx}_? \ t$, if $X \notin Var(t)$, each pointer to X is replaced by a pointer to t

Optimisation techniques for equivalence

- the nodes $t \;{}_?{\approx}_?\; t$ are erased
- $\pi \circ t \;{}_?{\approx}_?\; \pi' \circ t$ is replaced by $ds(\pi, \pi') \;\#_?\; t$
- $t \;{}_?{\approx}_?\; X$ or $X \;{}_?{\approx}_?\; t$, if $X \notin Var(t)$, each pointer to X is replaced by a pointer to t

$\Rightarrow$ $t \;{}_?{\approx}_?\; u$ not of the previous forms, unification rules are applied and each pointer to u is replaced by a pointer to t.

- the nodes $t \; {}_?{\approx}_? \; t$ are erased
- $\pi \circ t \; {}_?{\approx}_? \; \pi' \circ t$ is replaced by $ds(\pi, \pi') \; \#_? \; t$
- $t \; {}_?{\approx}_? \; X$ or $X \; {}_?{\approx}_? \; t$, if $X \notin Var(t)$, each pointer to X is replaced by a pointer to t
- $t \; {}_?{\approx}_? \; u$ not of the previous forms, unification rules are applied and each pointer to u is replaced by a pointer to t.

$\Rightarrow$ permutations on terms are evaluated 'by need': push one level down, only when needed to be able to apply a transformation rule (use a 'neutralising' permutation if necessary)

Optimisation techniques for equivalence

- the nodes $t \,{}_?{\approx}_?\, t$ are erased
- $\pi \circ t \,{}_?{\approx}_?\, \pi' \circ t$ is replaced by $ds(\pi, \pi') \,\#_?\, t$
- $t \,{}_?{\approx}_?\, X$ or $X \,{}_?{\approx}_?\, t$, if $X \not\in Var(t)$, each pointer to X is replaced by a pointer to t
- $t \,{}_?{\approx}_?\, u$ not of the previous forms, unification rules are applied and each pointer to u is replaced by a pointer to t.
- permutations on terms are evaluated 'by need': push one level down, only when needed to be able to apply a transformation rule (use a 'neutralising' permutation if necessary)

$\Rightarrow$ the graph is kept in canonical form: after each application of a unification rule, we compress consecutive permutation nodes, etc.

Complexity

$\Rightarrow$ The size of the problem (without counting freshness constraints) does not grow.

- The size of the problem (without counting freshness constraints) does not grow.

$\Rightarrow$ The number of constraints generated is linear in the size of the problem.

Complexity

- The size of the problem (without counting freshness constraints) does not grow.
- The number of constraints generated is linear in the size of the problem.

$\Rightarrow$ The number of transformation steps for each unification constraint is polynomial

Freshness

$\Rightarrow$ An efficient implementation should avoid computing freshness of an atom on a node several times.
Example:
$a \,\#_? \, f(X, X) \qquad X \;{}_?{\approx}_? \; t$

Freshness

- An efficient implementation should avoid computing freshness of an atom on a node several times.
  Example:
  $a \#_? f(X, X) \qquad X \; {}_?{\approx}_? \; t$

$\Rightarrow$ Need to remember which atoms have been already tested for freshness on a term, so each node t is tagged by a set A of atoms

Freshness

- An efficient implementation should avoid computing freshness of an atom on a node several times.
  Example:
  $a \#_? f(X, X) \qquad X \;{}_?{\approx}_?\; t$
- Need to remember which atoms have been already tested for freshness on a term, so each node t is tagged by a set A of atoms

$\Rightarrow$ On $a \#_? t$, if a is not in A, the transformation rule (depending on the form of t) is applied and a is added to A

Freshness

- An efficient implementation should avoid computing freshness of an atom on a node several times.
  Example:
  $a \#_? f(X, X) \qquad X \, {}_?{\approx}_? \, t$
- Need to remember which atoms have been already tested for freshness on a term, so each node t is tagged by a set A of atoms
- On $a \#_? t$, if a is not in A, the transformation rule (depending on the form of t) is applied and a is added to A

$\Rightarrow$ $a \#_? \pi \circ t$ is replaced by $\pi^{-1}(a) \#_? t$

- Nominal Terms: first-order syntax, with a notion of matching modulo α.

Conclusion

- Nominal Terms: first-order syntax, with a notion of matching modulo α.
- Higher-order substitutions are easy to define using freshness.

Conclusion

- Nominal Terms: first-order syntax, with a notion of matching modulo α.
- Higher-order substitutions are easy to define using freshness.
- Nominal matching is decidable and linear in time.

Conclusion

- Nominal Terms: first-order syntax, with a notion of matching modulo α.
- Higher-order substitutions are easy to define using freshness.
- Nominal matching is decidable and linear in time.
- Nominal rewriting has the expressive power of higher-order rewriting.

Conclusion

- Nominal Terms: first-order syntax, with a notion of matching modulo α.
- Higher-order substitutions are easy to define using freshness.
- Nominal matching is decidable and linear in time.
- Nominal rewriting has the expressive power of higher-order rewriting.
- Nominal unification is polynomial (unknown lower bound).

Conclusion

- Nominal Terms: first-order syntax, with a notion of matching modulo α.
- Higher-order substitutions are easy to define using freshness.
- Nominal matching is decidable and linear in time.
- Nominal rewriting has the expressive power of higher-order rewriting.
- Nominal unification is polynomial (unknown lower bound).
- Nominal unificaiton is used in the language α-Prolog [Cheney and Urban]

Conclusion

- Nominal Terms: first-order syntax, with a notion of matching modulo α.
- Higher-order substitutions are easy to define using freshness.
- Nominal matching is decidable and linear in time.
- Nominal rewriting has the expressive power of higher-order rewriting.
- Nominal unification is polynomial (unknown lower bound).
- Nominal unificaiton is used in the language α-Prolog [Cheney and Urban]
- Type systems for nominal terms are available.

That's all!

Questions ?