# Introduction to Unification Theory

## Applications

Temur Kutsia

RISC, Johannes Kepler University of Linz, Austria
kutsia@risc.jku.at

# Outline

Theorem Proving

Programming

Program Transformation

Computational Linguistics

# Outline

Theorem Proving

Programming

Program Transformation

Computational Linguistics

# Theorem Proving

- Robinson's unification algorithm was introduced in the context of theorem proving.
- Unification: Computational mechanism behind the resolution inference rule.

# Resolution

- Resolution is a rule of logical inference that allows one from "*A or B*" and "*not-A or C*" to conclude that "*B or C*".
- Logically

$$\frac{A \vee B \qquad\qquad \neg A \vee C}{B \vee C}$$

- For instance, from the two sentences
  - *it rains or it is sunny,*
  - *it does not rain or trees are wet*
    (this is the same as *if it rains then trees are wet*)

  one concludes that
  - *it is sunny or trees are wet*.
- Just take $A$ for *it rains*, $B$ for *it is sunny*, and $C$ for *trees are wet*.

# Resolution

- Resolution for first-order clauses:

$$\frac{A_1 \vee B \qquad\qquad \neg A_2 \vee C}{B\sigma \vee C\sigma},$$

  where $\sigma = mgu(A_1, A_2)$.
- For instance, from the two sentences
  - *Every number is less than its successor.*
  - *If $y$ is less than $x$ then $y$ is less than the successor of $x$.*

  one concludes that
  - *every number is less than the successor of its successor.*
- How?

# Resolution

- Let's write the sentences as logical formulae.
- *Every number is less than its successor:*
  $\forall x\ number(x) \Rightarrow less\_than(x, s(x))$
- *If $y$ is less than $x$ then $y$ is less than the successor of $x$:*
  $\forall y \forall x\ less\_than(y, x) \Rightarrow less\_than(y, s(x))$
- Write these formulae in disjunctive form and strip off the quantifiers:
  $\neg number(x) \vee less\_than(x, s(x))$
  $\neg less\_than(y, x) \vee less\_than(y, s(x))$

# Resolution

- Prepare for the resolution step. Make the clauses variable disjoint:
  $\neg number(x) \vee less\_than(x, s(x))$
  $\neg less\_than(y, x') \vee less\_than(y, s(x'))$
- Unify $less\_than(x, s(x))$ and $less\_than(y, x')$. The mgu $\sigma = \{x \mapsto y, x' \mapsto s(y)\}$
- Perform the resolution step and obtain the resolvent: $\neg number(y) \vee less\_than(y, s(s(y)))$.
- What would go wrong if we did not make the clauses variable disjoint?

# Factoring

- Another rule in resolution calculus that requires unification.
- Factoring

$$\frac{A_1 \vee A_2 \vee C}{A_1\sigma \vee C\sigma}$$

  where $\sigma = mgu(A_1, A_2)$.

# Resolution and Factoring in Action

Given:

- If $y$ is less than $x$ then $y$ is less than the successor of $x$.
- If $x$ is not less than a successor of some $y$, than 0 is less than $x$.

Prove:

- 0 is less than its successor.

# Resolution and Factoring in Action

Translating into formulae.

Given:

- $\neg less\_than(y, x) \vee less\_than(y, s(x))$.
- $less\_than(x, s(y)) \vee less\_than(0, x)$.

Prove:

- $less\_than(0, s(0))$

# Resolution and Factoring in Action

Negate the goal and try to derive the contradiction:

1. $\neg less\_than(y, x) \vee less\_than(y, s(x))$.
2. $less\_than(x, s(y)) \vee less\_than(0, x)$.
3. $\neg less\_than(0, s(0))$.
4. $less\_than(0, s(x)) \vee less\_than(x, s(y))$,
   (Resolvent of the renamed copy of 1
   $\neg less\_than(y', x') \vee less\_than(y', s(x'))$ and 2, obtained by unifying $less\_than(y', x')$ and $less\_than(0, x)$ with $\{y' \mapsto 0, x' \mapsto x\}$.
5. $less\_than(0, s(0))$
   (Factor of 4 with $\{x \mapsto 0, y \mapsto 0\}$
6. $\square$
   (Contradiction, resolvent of 3 and 5).

# Outline

Theorem Proving

Programming

Program Transformation

Computational Linguistics

# Logic Programming

- Unification plays a crucial role in logic programming.
- Used to perform execution steps.

# Logic Programming

- Logic programs consist of (nonnegative) clauses, written:

$$A \leftarrow B_1, \ldots, B_n,$$

  where $n \geq 0$ and $A, B_i$ are atoms.
- Example:
  - $likes(john, X) \leftarrow likes(X, wine).$
    John likes everybody who likes wine.
  - $likes(john, wine).$
    John likes wine.
  - $likes(mary, wine).$
    Marry likes wine.

# Logic Programming

- Goals are negative clauses, written

$$\leftarrow D_1, \ldots, D_m$$

  where $m \geq 0$.
- Example:
  - $\leftarrow likes(john, X)$.
    Who (or what) does John like?
  - $\leftarrow likes(X, marry), likes(X, wine)$.
    Who likes both marry and wine?
  - $\leftarrow likes(john, X), likes(Y, X)$.
    Find such $X$ and $Y$ that both John and $Y$ like $X$.

# Logic Programming

Inference step:

$$\frac{\leftarrow D_1, \ldots, D_m}{\leftarrow D_1\sigma, \ldots, D_{i-1}\sigma, B_1\sigma, \ldots, B_n\sigma, D_{i+1}\sigma, \ldots, D_m\sigma}$$

where $\sigma = mgu(D_i, A)$ for (a renamed copy of) some program clause $A \leftarrow B_1, \ldots, B_n$.

# Logic Programming

## Example

Program:

$$likes(john, X) \leftarrow likes(X, wine).$$
$$likes(john, wine).$$
$$likes(mary, wine).$$

Goal:

$$\leftarrow likes(X, marry), likes(X, wine).$$

Inference:

- Unifying *likes(X,marry)* with *likes(john,X')* gives $\{X \mapsto john, X' \mapsto marry\}$
- New goal: $\leftarrow likes(marry, wine), likes(john, marry).$

# Prolog

- Prolog: Most popular logic programming language.
- Unification in Prolog is nonstandard: Omits occur-check.
- Result: Prolog unifies terms $x$ and $f(x)$, using the substitution $\{x \mapsto f(f(f(\dots)))\}$.
- Because of that, sometimes Prolog might draw conclusions the user does not expect:

```
less(X, s(X)).
foo :- less(s(Y), Y).
?- foo.
yes.
```

- Infinite terms in a theoretical model for real Prolog implementations.

# Higher-Order Logic Programming

## Example

A $\lambda$-Prolog program:

```
(age bob 24).
(age sue 23).
(age ned 23).
(mappred P nil nil).
(mappred P (X::L) (Y::K)):- (P X Y), (mappred P L K).
```

mappred maps the predicate P on the lists (X::L) and (Y::K).

The goal (mappred x\y\(age x y) L (23::24::nil)) returns two answers:

L = (sue::bob::nil)

L = (ned::bob::nil)

# Higher-Order Logic Programming

## Example (Cont.)

- On the previous slide, the goal was unified with the head of (a copy of) the second mappred clause by the substitution

$$\{P \mapsto x\backslash y\backslash(age\ x\ y), L \mapsto (X :: L'), Y \mapsto 23, K \mapsto (24 :: nil)\}$$

  x\y\(age x y) is the $\lambda$-Prolog notation for $\lambda x.\lambda y.\ (age\ x\ y)$.
- It made the new goal

  (age X 23), (mappred x\y\(age x y) L' (24::nil))).

etc.

# Higher-Order Logic Programming

- The fragment of higher-order unification used in $\lambda$-prolog is unification for higher-order patterns.
- Higher-order pattern is a $\lambda$-term where arguments of free variables are distinct bound variables.
- Higher-order pattern unification is unitary.

# Programming in Mathematica

- Mathematica is a symbolic computation system, a product of Wolfram Research, Inc.
- It comes with a rule based programming language.
- An example of Mathematica code to compute factorial:

```
f[0] := 1
f[n_] := n * f[n - 1]/; n > 0
```

- To compute `f[5]`, it first tries to match 0 with 5, which fails.
- Next, `n` matches 5 with the substitution $\mathtt{n} \mapsto \mathtt{5}$, the condition $5 > 0$ is satisfied and the next goal becomes `5*f[4]`.
- `n_` indicates that `n` is a variable that can match an expression.
- Matching is a special case of unification: $s \doteq^{?} t$ is a matching problem if $t$ is ground.

# Programming in Mathematica

- Mathematica has a special kind of variable, called sequence variable.
- Sequence variables can be instantiated by finite sequences.
- Convenient to write short, elegant programs.
- Unification with sequence variables is decidable and infinitary, matching is finitary.

# Programming in Mathematica

- An example of Mathematica code for bubble sort:

  $$\mathtt{sort}[\{\mathtt{x}\_\_\_,\mathtt{u}\_,\mathtt{y}\_\_\_,\mathtt{v}\_,\mathtt{z}\_\_\_\}] := \mathtt{sort}[\{\mathtt{x},\mathtt{v},\mathtt{y},\mathtt{u},\mathtt{z}\}]/;\mathtt{u} > \mathtt{v}$$
  $$\mathtt{sort}[\{\mathtt{x}\_\_\_\}] := \{\mathtt{x}\}$$

- $\mathtt{x}\_\_\_$ indicates that $\mathtt{x}$ is a sequence variable.
- $\mathtt{sort}[\{\mathtt{x}\_\_\_,\mathtt{u}\_,\mathtt{y}\_\_\_,\mathtt{v}\_,\mathtt{z}\_\_\_\}]$ matches $\mathtt{sort}[\{1,2,3,4,1\}]$ in various ways.
- The one that satisfies the condition $\mathtt{u} > \mathtt{v}$ is

  $$\{\mathtt{x} \mapsto 1, \mathtt{u} \mapsto 2, \mathtt{y} \mapsto (3,4), \mathtt{v} \mapsto 1, \mathtt{z} \mapsto ()\}$$

- The next goal becomes $\mathtt{sort}[\{1,1,3,4,2\}]$, and so on.

# Outline

Theorem Proving

Programming

Program Transformation

Computational Linguistics

# Program Transformation

- Program transformation is the act of changing one program into another.
- Some techniques describe transformation as rewriting systems for program schemas, together with constraints on the instances of the schemas that must be met in order for the transformation to be valid.
- When a rewriting rule is applied to a particular program, the schema in the left hand side of the rule should match the program.
- Usually schemas are expressed in a higher-order language.
- Leads to higher-order matching.

# Program Transformation

## Example (Schema Matching)

- Schema:

$$F(x) \Leftarrow \textbf{if } A(x) \textbf{ then } B(x) \textbf{ else } H(D(x), F(E(x))).$$

- Instance program:

$$\mathsf{fact}(x) \Leftarrow \textbf{if } x = 0 \textbf{ then } 1 \textbf{ else } x * \mathsf{fact}(x - 1)$$

- The schema matches the instance with the substitution:

$$\{F \mapsto \lambda x.\mathsf{fact}(x), A \mapsto \lambda x.x = 0, B \mapsto \lambda x.1,$$
$$H \mapsto \lambda x.\lambda y.x * y, D \mapsto \lambda x.x, E \mapsto \lambda x.x - 1\}$$

# Program Transformation

## Example (Schema Matching)

The same schema, different instance.

- Schema:

$$F(x) \Leftarrow \textbf{if } A(x) \textbf{ then } B(x) \textbf{ else } H(D(x), F(E(x)))$$

- Instance:

$$\mathsf{rev}(x) \Leftarrow \textbf{if } \mathsf{Null}(x) \textbf{ then } x \textbf{ else } \mathsf{app}(\mathsf{rev}(\mathsf{Cdr}(x)), \mathsf{Cons}(\mathsf{Car}(x), \mathsf{nil}))$$

- Matching substitution:

$$\{F \mapsto \lambda x.\mathsf{rev}(x), A \mapsto \lambda x.\mathsf{Null}(x), B \mapsto \lambda x.x,$$
$$H \mapsto \lambda x.\lambda y.\mathsf{app}(y, x), D \mapsto \lambda x.\mathsf{Cons}(\mathsf{Car}(x), \mathsf{nil}),$$
$$E \mapsto \lambda x.\mathsf{Cdr}(x)\}$$

# Outline

# Ellipsis Resolution

- An elliptical construction involves two phrases (clauses) that are parallel in structure in some sense.
- The source clause is complete.
- The target clause is missing material found in the source.
- Goal: To recover the property of the parallel element in the target the missing material stands for.

# Ellipsis Resolution

## Example

- Dan likes golf, and George does too.
- "Dan" and "George" are parallel elements.
- Semantic interpretation of "Dan likes golf": $like(\underline{dan}, golf)$.
- $\underline{dan}$ is called a primary occurrence.
- To interpret "George does too", we require the property $P$ such that, when applied to the interpretation of the subject of "Dan likes golf", i.e. $dan$, gives the interpretation of "Dan likes golf".
- Find $P$ such that $P(dan) \doteq^{?} like(\underline{dan}, golf)$.
- $\sigma_1 = \{P \mapsto \lambda x.like(\underline{dan}, golf)\}$, $\sigma_2 = \{P \mapsto \lambda x.like(x, golf)\}$.
- Constraint: Solution should not contain the primary occurrence. Hence, $\sigma_2$ is the only solution.
- Interpretation: $like(dan, golf) \wedge P(george)\sigma_2$ that gives $like(dan, golf) \wedge like(george, golf)$.

# Ellipsis Resolution

- Higher-order unification generates multiple solutions.
- Leads to multiple readings.
- Constraints help to filter out some.
- Still, several may remain.
- Strict and sloppy reading.

# Ellipsis Resolution

## Example

- Dan likes his wife, and George does too.
- Semantic interpretation of "Dan likes his wife":
  $like(\underline{dan}, wife\text{-}of(dan))$.
- $\underline{dan}$ is a primary occurrence, $dan$ is secondary, because it came from the pronoun which is not a parallel element.
- Find $P$ such that $P(dan) \doteq^{?} like(\underline{dan}, wife\text{-}of(dan))$.
- $\sigma_1 = \{P \mapsto \lambda x.like(\underline{dan}, wife\text{-}of(dan))\}$,
  $\sigma_2 = \{P \mapsto \lambda x.like(\underline{dan}, wife\text{-}of(x))\}$,
  $\sigma_3 = \{P \mapsto \lambda x.like(x, wife\text{-}of(dan))\}$,
  $\sigma_4 = \{P \mapsto \lambda x.like(x, wife\text{-}of(x))\}$
- Constraint: Solution should not contain the primary occurrence. Hence, $\sigma_1$ and $\sigma_2$ are discarded.
- Strict reading: $P(george)\sigma_3 = like(george, wife\text{-}of(dan))$.
- Sloppy reading: $P(george)\sigma_4 = like(george, wife\text{-}of(george))$.

# Brief Summary of the Course

- First-order syntactic unification
  - Most general unifier.
  - Unification algorithm.
  - Improvements of the algorithm.
- First-order equational unification
  - Minimal complete set of unifiers.
  - Decidability/Undecadibility, Unification type.
  - Results for particular theories.
  - Universal E-unification procedure.
  - Narrowing.
- Higher-order unification
  - Undecidability.
  - Unification type (zero).
  - Preunification procedure.
- Applications related to logic, language, and information
  - Theorem proving.
  - Programming, program transformation.
  - Ellipsis resolution.

# Open Problems

The RTA list of open problems:

`http://www.win.tue.nl/rtaloop/`