Introduction to Unification Theory

Speeding Up

Temur Kutsia

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

Improving the Recursive Descent Algorithm

- Improvement 1: Linear Space, Exponential Time
- Improvement 2. Linear Space, Quadratic Time
- Improvement 3. Almost Linear Algorithm

Unification via $\mathcal{U}$: Exponential in Time and Space

Example

Unifying s and t, where

$$s = h(x_1, x_2, \ldots, x_n, f(y_0, y_0), f(y_1, y_1), \ldots, f(y_{n-1}, y_{n-1}), y_n)$$
$$t = h(f(x_0, x_0), f(x_1, x_1), \ldots, f(x_{n-1}, x_{n-1}), y_1, y_2, \ldots, y_n, x_n)$$

will create an mgu where each x_i and each y_i is bound to a term with $2^{i+1} - 1$ symbols:

$$\{x_1 \mapsto f(x_0, x_0), x_2 \mapsto f(f(x_0, x_0), f(x_0, x_0)), \ldots,$$
$$y_0 \mapsto x_0, y_1 \mapsto f(x_0, x_0), y_2 \mapsto f(f(x_0, x_0), f(x_0, x_0)), \ldots\}$$

Can we do better?

Unification via $\mathcal{U}$: Exponential in Time and Space

First idea: Use triangular substitutions.

Example

Triangular unifier of s and t from the previous example:

$$[y_0 \mapsto x_0; y_n \mapsto f(y_{n-1}, y_{n-1}); y_{n-1} \mapsto f(y_{n-2}, y_{n-2}); \ldots]$$

- Triangular unifiers are not larger than the original problem.
- However, it is not enough to rescue the algorithm:
  - Substitutions have to be applied to terms in the problem, that leads to duplication of subterms.
  - In the example, unifying x_n and y_n, which by then are bound to terms with $2^{n+1} - 1$ symbols, will lead to exponential number of decompositions.

Unification via $\mathcal{U}$: Exponential in Time and Space

- Problem: Duplicate occurrences of the same variable cause the explosion in the size of terms.
- Fix: Represent terms as graphs which share subterms.

Term Dags

Term Dag

A term dag is a directed acyclic graph such that

- its nodes are labeled with function symbols or variables,
- its outgoing edges from any node are ordered,
- outdegree of any node labeled with a symbol f is equal to the arity of f,
- nodes labeled with variables have outdegree 0.

Term Dags

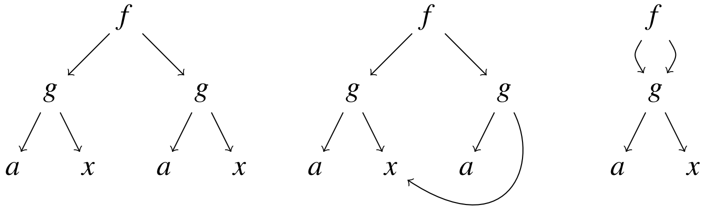
- Convention: Nodes and terms the term dags represent will not be distinguished.
- Example: “node” $f(a,x)$ is a node labeled with f and having two arcs to a and to x.

Term Dags

The only difference between various dags representing the same term is the amount of structure sharing between subterms.

Example

Three representations of the term $f(g(a,x), g(a,x))$:



Term Dags

- It is possible to build a dag with unique, shared variables for a given term in $O(n * \log(n))$ where n is the number of symbols in the term.
- There are subtle variations that can improve this result to $O(n)$.
- Assumption for the algorithm we plan to consider:
  - The input is a term dag representing the two terms to be unified, with unique, shared occurrences of all variables.

Term Dags

Representing substitutions involving only subterms of a term dag:

- Directly by a relation on the nodes of the dag, either
  - stored explicitly as a list of pairs, or
  - by storing a link ("substitution arcs") in the graph itself, and maintaining a list of variables (nodes) bound by the substitution.

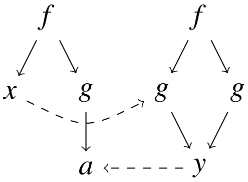
Term Dags

Substitution application.

- Implicit: Identifies two nodes connected with a substitution arc, without actually moving any of the subterm links.

Example

A term dag for the terms $f(x, g(a))$ and $f(g(y), g(y))$, with the implicit application of their mgu $\{x \mapsto g(a), y \mapsto a\}$.



Term Dags

- With implicit application, the binding for a variable can be determined by traversing the graph depth first, left to right.

Improvement 1: Linear Space, Exponential Time

Assumptions:

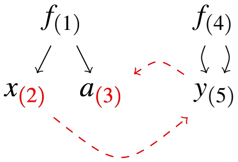
- Dags consist of nodes.
- Any node in a given dag defines a unique subdag (consisting of the nodes which can be reached from this node), and thus a unique subterm.
- Two different types of nodes: variable nodes and function nodes.
- Information at function nodes:
  - The name of the function symbol.
  - The arity n of this symbol.
  - The list (of length n) of successor nodes (corresponds to the argument list of the function)
- Both function and variable nodes may be equipped with one extra pointer (dashed arrow in diagrams) to another node.

Auxiliary procedures for Unification on Term Dags

- `Find`:
  Takes a node of a dag as input, and follows the additional pointers until it reaches a node without such a pointer. This node is the output of `Find`.

Example

- `Find`(3)=(3)
- `Find`(2)= (3)



Auxiliary procedures for Unification on Term Dags

- `Union`:
  Takes as input a pair of nodes u, v that do not have additional pointers and creates such a pointer from u to v.

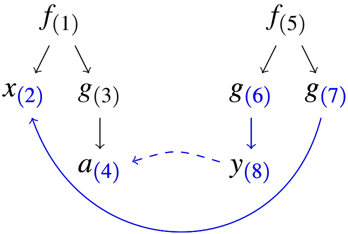
Auxiliary procedures for Unification on Term Dags

- `Occur`:
  Takes as input a variable node u and another node v (both without additional pointers) and performs the occur check, i.e. it tests whether the variable is contained in the term corresponding to v. The test is performed on the virtual term expressed by the additional pointer structure, i.e. one applies `Find` to all nodes that are reached during the test.

Auxiliary procedures for Unification on Term Dags

- `Occur`

Example



- `Occur`(2,6)=False
- `Occur`(2,7)=True

Unification of Term Dags

Input: A pair of nodes k_1 and k_2 in a dag
Output: *True* if the terms corresponding to k_1 and k_2 are unifiable. *False* Otherwise.
Side Effect: A pointer structure which allows to read off an mgu and the unified term.

Procedure Unify1. Unification of term dags.

(Continues on the next slide)

Unification of Term Dags

```
Unify1 (k1, k2)

if k1 = k2 then return True                          /* Trivial */

else
  if function-node(k2) then
  |  u := k1; v := k2
  else
  |  u := k2; v := k1                                /* Orient */
  end
```

Procedure Unify1. Unification of term dags.

(Continues on the next slide)

Unification of Term Dags

```
if variable-node(u) then
|   if Occurs (u, v)                                 /* Occur-check */
|    then
|    |   return False
|   else
|    |   Union(u, v)                                 /* Variable elimination */
|    |   return True
|   end
else if function-symbol(u) ≠ function-symbol(v)
 then
|   return False                                     /* Symbol clash */
```

Procedure Unify1. Unification of term dags. Continued.

(Continues on the next slide)

Unification of Term Dags

else

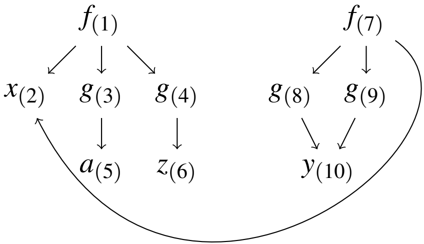
$n :=$ *arity*(*function-symbol*(u))
($u_1, \ldots, u_n$) $:=$ *succ-list*(u)
($v_1, \ldots, v_n$) $:=$ *succ-list*(v)
$i := 0$; $bool :=$ *True*

while $i \leq n$ **and** $bool$ **do**
  $i := i + 1$; $bool :=$ `Unify1`(`Find`(u_i), `Find`(v_i))
   /* Decomp. */
end
return $bool$

Procedure Unify1. Unification of term dags. Finished.

Unification of Term Dags. Example 1

- Unify $f(x, g(a), g(z))$ and $f(g(y), g(y), x)$.
- First, create dags.
- Numbers indicate nodes.



Unification of Term Dags. Example 1

Algorithm run starts with $\texttt{Unify1}(1, 7)$ and continues:

$\texttt{Unify1}(\texttt{Find}(2), \texttt{Find}(8))$

$\texttt{Find}(2) = (2)$

$\texttt{Find}(8) = (8)$

$\texttt{Occur}(2, 8) =$ *False*

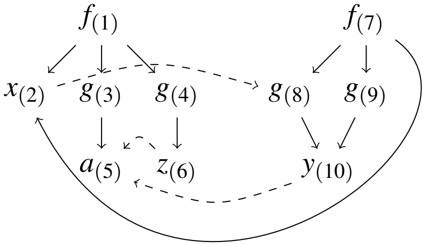
$\texttt{Union}(2, 8)$

$\texttt{Unify1}(\texttt{Find}(3), \texttt{Find}(9))$

$\texttt{Find}(3) = (3)$

$f_{(1)}$ $f_{(7)}$

Unification of Term Dags. Example 1 (Cont.)



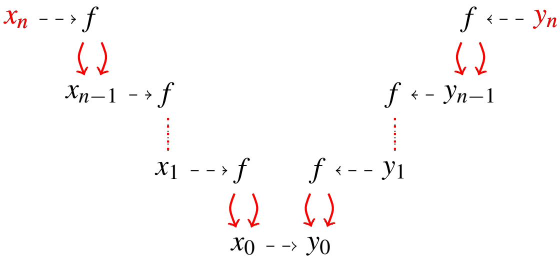
- From the final dag one can read off:
  - The unified term $f(g(a), g(a), g(a))$.
  - The mgu in triangular form $[x \mapsto g(y); y \mapsto a; z \mapsto a]$.
- No new nodes. Only one extra pointer for each variable node.
- Needs linear space.
- Time is still exponential. See the next example.

Unification of Term Dags. Example 2

Consider again the problem $s \doteq^{?} t$, where

$$s = h(x_1, x_2, \dots, x_n, f(y_0, y_0), f(y_1, y_1), \dots, f(y_{n-1}, y_{n-1}), y_n)$$
$$t = h(f(x_0, x_0), f(x_1, x_1), \dots, f(x_{n-1}, x_{n-1}), y_1, y_2, \dots, y_n, x_n)$$

A dag representation of the term bound to x_n and y_n:



Exponential number of recursive calls.

Unification of Term Dags: Correctness

`Unify1` can be simulated by $\mathcal{U}$ such that

- If the call to `Unify1` ends in failure, then the corresponding transformation sequence in $\mathcal{U}$ ends in $\bot$.
- If the call to `Unify1` terminates with success, with a substitution σ read from the pointer structure, then the corresponding transformation sequence $\mathcal{U}$ ends in $\emptyset; S$ where $\sigma_S = \sigma$.

Unification of Term Dags: Complexity

- Linear space: terms are not duplicated anymore.
- Exponential time: Calls `Unify1` recursively exponentially often.
- Fortunately, with an easy trick one can make the running time quadratic.
- Idea: Keep from revisiting already-solved problems in the graph.
- The algorithm of Corbin and Bidoit:

J. Corbin and M. Bidoit.
A rehabilitation of Robinson's unification algorithm.
In R. Mason, editor, *Information Processing 83*, pages 909–914. Elsevier Science, 1983.

Improvement 2. Linear Space, Quadratic Time

Input: A pair of nodes k_1 and k_2 in a dag.
Output: *True* if the terms corresponding to k_1 and k_2 are unifiable. *False* Otherwise.
Side Effect: A pointer structure which allows to read off an mgu and the unified term.

Procedure Unify2. Quadratic Algorithm.
(No difference from `Unify1` so far. Continues on the next slide)

Quadratic Algorithm

```
Unify2 (k1, k2)
if k1 = k2 then return True                         /* Trivial */

else
  if function-node(k2) then
    u := k1; v := k2
  else
    u := k2; v := k1                                 /* Orient */
  end
```

Procedure Unify2. Quadratic Algorithm.

(No difference from `Unify1` so far. Continues on the next slide)

Quadratic Algorithm

```
if variable-node(u) then
    if Occurs (u, v)                          /* Occur-check */
     then
        return False
    else
        Union(u, v)                           /* Variable elimination */
        return True
    end
else if function-symbol(u) ≠ function-symbol(v) then
    return False                              /* Symbol clash */
```

Procedure Unify2. Quadratic Algorithm. Continued.

(No difference from `Unify1` so far. Continues on the next slide)

Quadratic Algorithm

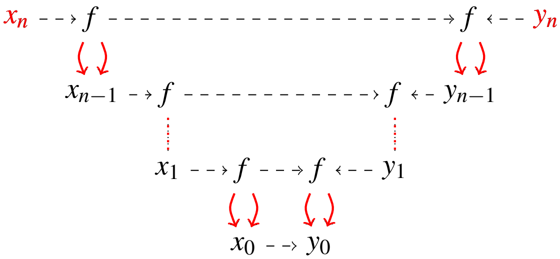
```
else
   n := arity(function-symbol(u))
   (u_1, ..., u_n) := succ-list(u)
   (v_1, ..., v_n) := succ-list(v)
   i := 0;  bool := True

   Union(u,v)
   while i ≤ n and bool do
      i := i + 1;  bool := Unify2(Find(u_i), Find(v_i))
        /* Decomp. */
   end
   return bool
```

Procedure Unify2. Quadratic Algorithm. Finished.

(The only difference from `Unify1` is `Union(u,v)`.)

Quadratic Algorithm. Example

The same example that revealed exponential behavior of RDA:



Why is it Quadratic?

- The algorithm is quadratic in the number of symbols in original terms:
  - Each call of `Unify2` either returns immediately, or makes one more node unreachable for the `Find` operation.
  - Therefore, there can be only linearly many calls of `Unify2`.
  - Quadratic complexity comes from the fact that `Occur` and `Find` operations are linear.

Improvement 3. Almost Linear Algorithm

How to eliminate two sources of nonlinearity of `Unify2`?

- `Occur`: Just omit the occur check during the execution of the algorithm.
  - Consequence: The data structure may contain cycles.
  - Since the occur-check failures are not detected immediately, at the end an extra check has to be performed to find out whether the generated structure is cyclic or not.
  - Detecting cycles in a directed graph can be done by linear search.
- `Find`: Use more efficient union-find algorithm from

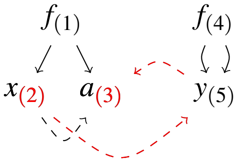
R. Tarjan.
Efficiency of a good but not linear set union algorithm.
J. ACM, 22(2):215–225, 1975.

Auxiliary Procedures for the Almost Linear Algorithm

- `Collapsing-find`:
  - Like `Find` it takes a node k of a dag as input, and follows the additional pointers until the node $\texttt{Find}(k)$ is reached.
  - In addition, `Collapsing-find` relocates the pointer of all the nodes reached during this process to $\texttt{Find}(k)$.

Example

- `CF`(3)=(3)
- `CF`(2)= (3)



Auxiliary Procedures for the Almost Linear Algorithm

- `Union-with-weight`:
  - Takes as input a pair of nodes u, v that do not have additional pointers.
  - If the set $\{k \mid \texttt{Find}(k) = u\}$ larger than the set $\{k \mid \texttt{Find}(k) = v\}$ then it creates an additional pointer from v to u.
  - Otherwise, it creates an additional pointer from u to v.
  - Hence, the link is created from the smaller tree to the larger one, increasing the path to the root (the result of `Find`) for fewer nodes.

Weighted union does not apply when we have a variable node and a function node.

Almost Linear Algorithm

One more auxiliary procedure:

- `Not-cyclic`:
  - Takes a node k as input, and tests the graph which can be reached from k for cycles.
  - The test is performed on the virtual graph expressed by the additional pointer structure, i.e. one first applies `Collapsing-find` to all nodes that are reached during the test.

Almost Linear Algorithm

Input: A pair of nodes k_1 and k_2 in a directed graph.
Output: *True* if k_1 and k_2 correspond unifiable terms. *False* Otherwise.
Side Effect: A pointer structure which allows to read off an mgu and the unified term.

```
Unify3 (k1, k2)
if Cyclic-unify (k1, k2) and Not-cyclic (k1) then
  | return True
else
  | return False
end
```

Procedure Unify3. Almost Linear Algorithm.

(Continues on the next slide)

Almost Linear Algorithm

```
Cyclic-unify (k1, k2)
if k1 = k2 then return True                          /* Trivial */

else
    if function-node(k2) then
        u := k1; v := k2
    else
        u := k2; v := k1                             /* Orient */
    end
```

Procedure Cyclic-unify.

(Continues on the next slide)

Almost Linear Algorithm

```
Cyclic-unify (s,t)
if variable-node(u) then
    if variable-node(v) then
        Union-with-weight(u, v)
    else
        Union(u, v)    /* No occur-check. Variable elimination */
        return True
    end
else if function-symbol(u) ≠ function-symbol(v) then
    return False                               /* Symbol clash */
```

Procedure Cyclic-unify.

(Continues on the next slide)

Almost Linear Algorithm

```
else
    n := arity(function-symbol(u))
    (u_1, ..., u_n) := succ-list(u)
    (v_1, ..., v_n) := succ-list(v)
    i := 0;  bool := True

    Union-with-weight (u,v)
    while i ≤ n and bool do
        i := i + 1
        bool :=
            Cyclic-unify(Collapsing-find(u_i)
              Collapsing-find(v_i))    /* Decomposition */
    end
    return bool
```

Procedure Cyclic-unify. Finished.

Almost Linear Algorithm

The algorithm is very similar to the one described in Gerard Huet's thesis:

G. Huet.
Résolution d'Équations dans des Langages d'ordre $1, 2, \ldots, \omega$.
Thèse d'État, Université de Paris VII, 1976.

Complexity

- The algorithm is almost linear in the number of symbols in original terms:
  - Each call of `Cyclic-unify` either returns immediately, or makes one more node unreachable for the `Collapsing-find` operation.
  - Therefore, there can be only linearly many calls of `Cyclic-unify`.
  - A sequence of n `Collapsing-find` and `Union-with-weight` operations can be done in $O(n * \alpha(n))$ time, where α is an extremely slowly growing function (functional inverse of Ackerman's function) never exceeding 5 for practical input.
  - The use of nonoptimal `Union` can increase the time complexity at most by a summand $O(m)$ where m is the number of different variable nodes.
  - Therefore, complexity of `Cyclic-unify` is $O(n * \alpha(n))$.
  - Complexity of `Not-cyclic` is linear.
  - Hence, complexity of `Unify3` is $O(n * \alpha(n))$.

Implementation: Matching vs. Unification

- Unlike matching, efficient unification algorithms require sophisticated data structures.
- When efficiency is an issue, matching should be implemented separately from unification.

Summary

- Recursive Descent Algorithm for unification is exponential in time and space.
- Using term dags reduces space complexity to linear.
- Making the union pointer between function nodes before unifying their arguments reduces time complexity to quadratic.
- Using collapsing-find and union-with-weight further reduces time complexity to almost linear.