Introduction to Unification Theory

Speeding Up

Temur Kutsia

RISC, Johannes Kepler University of Linz, Austria
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

Example from the Previous Lecture

Example

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

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

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

- 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

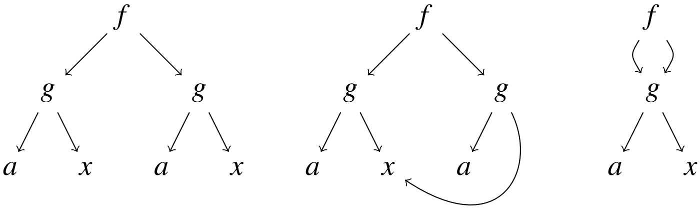
- Convention: Nodes and terms the term dags represent will not be distinguished.
- Example: “node” $f(a, x)$ is a node labeled with f and having two arcs to a and to x.

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.
- 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.

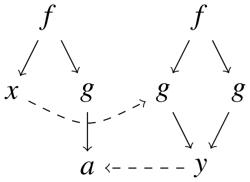
Term Dags

Substitution application. Two alternatives:

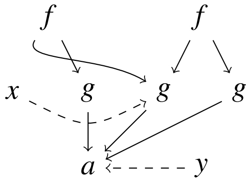
1. Implicit: Identifies two nodes connected with a substitution arc, without actually moving any of the subterm links.
2. Explicit: Expresses the substitution by moving any arc (subterm or substitution) pointing to a variable to point to a binding.

Example

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



Implicit



Explicit

Term Dags

- With implicit application, the binding for a variable can be determined by traversing the graph depth first, left to right.
- Explicit application represents a substitution in a direct way.

Recursive Descent Algorithm (RDA) on Term Dags

Assumptions:

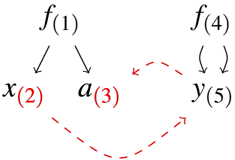
- Dags consist of nodes.
- Any node in a given dag defines a unique subdag (consisting of the nodes which can be reached from this node), and thus a unique subterm.
- Two different types of nodes: variable nodes and function nodes.
- Information at function nodes:
  - The name of the function symbol.
  - 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 additional pointer (displayed as a dashed arrow in diagrams) to another node.

Auxiliary procedures for the RDA 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 the RDA on Term Dags

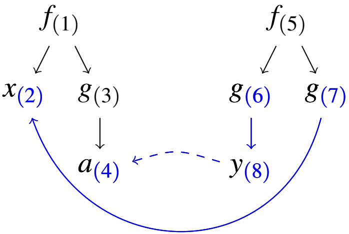
- `Union`:
  Takes as input a pair of nodes u, v that do not have additional pointers and creates such a pointer from u to v.

Auxiliary procedures for the RDA 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.

Example

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



RDA on 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.

`Unify1` (k_1, k_2)
if $k_1 = k_2$ **then return** *True*; /* Trivial */
else
  if *function-node*(k_2) **then**
    $u := k_1; v := k_2$
  else
    $u := k_2; v := k_1$; /* Orient */
  end

Procedure `Unify1`. Recursive descent algorithm on term dags.

(Continues on the next slide)

Recursive Descent Algorithm on 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
```

Procedure `Unify1`. Recursive descent algorithm on term dags. Continued.

(Continues on the next slide)

Recursive Descent Algorithm on Term Dags

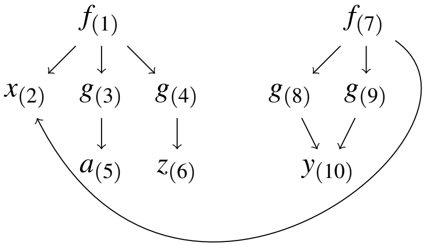
else if $\mathit{function\text{-}symbol}(u) \neq \mathit{function\text{-}symbol}(v)$
then
  return *False*; $/\star$ Symbol clash $\star/$
else
  $n := \mathit{arity}(\mathit{function\text{-}symbol}(u))$;
  $(u_1, \ldots, u_n) := \mathit{succ\text{-}list}(u)$;
  $(v_1, \ldots, v_n) := \mathit{succ\text{-}list}(v)$;
  $i := 0$; $\mathit{bool} := \mathit{True}$;

  while $i \leq n$ **and** *bool* **do**
    $i := i + 1$; $\mathit{bool} := \texttt{Unify1}(\texttt{Find}(u_i), \texttt{Find}(v_i))$;
    $/\star$ Decomposition $\star/$
  end
  return *bool*

Procedure `Unify1`. Recursive descent algorithm on term dags.
Finished.

RDA on Term Dags. Example 1

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



RDA on Term Dags. Example 1

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

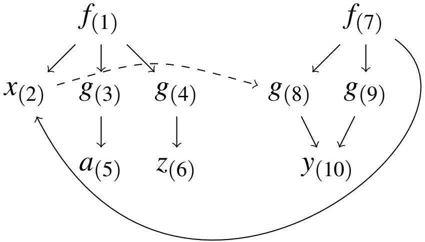
$\texttt{Unify1}(\texttt{Find}(2), \texttt{Find}(8))$

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

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

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

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



RDA on Term Dags. Example 1 (Cont.)

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

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

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

$\quad\texttt{Find}(9) = (9)$

$\quad\texttt{Unify1}(\texttt{Find}(5),\texttt{Find}(10))$

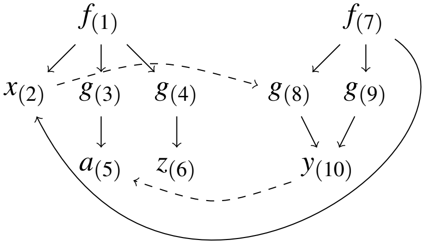
$\quad\quad\texttt{Find}(5) = 5$

$\quad\quad\texttt{Find}(10) = 10$

$\quad\quad \mathit{orient}(10,5)$

$\quad\quad\texttt{Occur}(10,5) = \mathit{False}$

$\quad\quad\texttt{Union}(10,5)$



RDA on Term Dags. Example 1 (Cont.)

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

$\texttt{Unify1}(\texttt{Find}(4), \texttt{Find}(2))$

  $\texttt{Find}(4) = 4$

  $\texttt{Find}(2) = 8$

$\texttt{Unify1}(4,8)$

  $\texttt{Unify1}(\texttt{Find}(6), \texttt{Find}(10))$

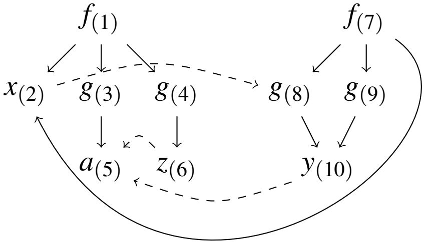
    $\texttt{Find}(6) = 6$

    $\texttt{Find}(10) = 5$

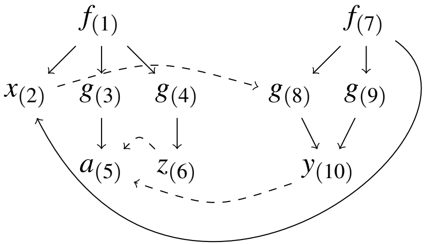
    $\texttt{Occur}(6,5) = \textit{False}$

    $\texttt{Union}(6,5)$

True



RDA on Term Dags. Example 1 (Cont.)



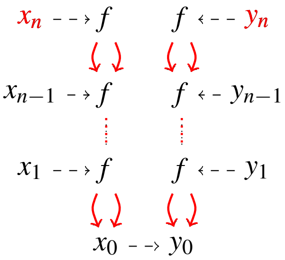
- From the final dag one can read off:
  - The unified term $f(g(a), g(a), g(a))$.
  - The mgu in triangular form $[x \mapsto g(y); y \mapsto a; z \mapsto a]$.
- The algorithm does not create new nodes. Only one extra pointer for each variable node.
- Needs linear space.
- Time is still exponential. See the next example.

RDA on Term Dags. Example 2

Consider again the problem:

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

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



Exponential number of recursive calls.

Correctness of RDA for Term Dags

- Proof is similar as for the RDA. These two algorithms differ only by the data structure they operate on.

Complexity of RDA for Term Dags

- 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.

Quadratic Algorithm on 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.

`Unify2` (k_1, k_2)
if $k_1 = k_2$ **then return** *True*; /* Trivial */
else
  if *function-node*(k_2) **then**
    $u := k_1; v := k_2$
  else
    $u := k_2; v := k_1$; /* 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

Procedure `Unify2`. Quadratic Algorithm. Continued.

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

Quadratic Algorithm

else if $\mathit{function\text{-}symbol}(u) \neq \mathit{function\text{-}symbol}(v)$
then
  return *False*; $/\star$ Symbol clash $\star/$
else
  $n := \mathit{arity}(\mathit{function\text{-}symbol}(u))$;
  $(u_1, \ldots, u_n) := \mathit{succ\text{-}list}(u)$;
  $(v_1, \ldots, v_n) := \mathit{succ\text{-}list}(v)$;
  $i := 0; \quad \mathit{bool} := \mathit{True}$;

  `Union`(u,v);
  while $i \leq n$ **and** *bool* **do**
    $i := i + 1; \; \mathit{bool} := \mathtt{Unify2}(\mathtt{Find}(u_i), \mathtt{Find}(v_i))$;
    $/\star$ Decomposition $\star/$
  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:

$$\begin{tikzcd}
x_n \arrow[r, dashed] & f \arrow[r, dashed] \arrow[d, bend right] \arrow[d, bend left] & f \arrow[d, bend right] \arrow[d, bend left] & y_n \arrow[l, dashed] \\
x_{n-1} \arrow[r, dashed] & f \arrow[r, dashed] & f & y_{n-1} \arrow[l, dashed] \\
 & \vdots & \vdots & \\
x_1 \arrow[r, dashed] & f \arrow[r, dashed] \arrow[d, bend right] \arrow[d, bend left] & f \arrow[d, bend right] \arrow[d, bend left] & y_1 \arrow[l, dashed] \\
 & x_0 \arrow[r, dashed] & y_0 &
\end{tikzcd}$$

Properties of the Quadratic Algorithm

- Correctness can be shown in the similar way as for the RDA.
- 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.

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

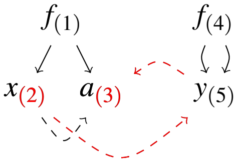
R. Tarjan.
Efficiency of a good but not linear set union algorithm.
J. ACM, 22(2):215–225, 1975.

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.

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` (k_1, k_2)
if $\mathit{Cyclic\text{-}unify}(k_1, k_2)$ `and` $\mathit{Not\text{-}cyclic}(k_1)$ **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

```
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
```

Procedure `Cyclic-unify`.

(Continues on the next slide)

Almost Linear Algorithm

else if $\mathit{function\text{-}symbol}(u) \neq \mathit{function\text{-}symbol}(v)$
then
  return *False*; $/\star$ Symbol clash $\star/$
else
  $n := \mathit{arity}(\mathit{function\text{-}symbol}(u))$;
  $(u_1, \ldots, u_n) := \mathit{succ\text{-}list}(u)$;
  $(v_1, \ldots, v_n) := \mathit{succ\text{-}list}(v)$;
  $i := 0$; $\mathit{bool} := \mathit{True}$;

  `Union-with-weight` (u,v);
  while $i \leq n$ **and** *bool* **do**
    $i := i + 1$;
    $\mathit{bool} :=$ `Cyclic-unify`(`Collapsing-find`(u_i)
      `Collapsing-find`(v_i)); $/\star$ Decomposition $\star/$
  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 $O(n * \alpha(n))$.

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.