

# Introduction to Unification Theory

## Speeding Up

Temur Kutsia

RISC, Johannes Kepler University of Linz, Austria  
`kutsia@risc.uni-linz.ac.at`



# Overview

Recursive Descent Algorithm is Expensive

Improvement 1: Linear Space, Exponential Time

Improvement 2. Linear Space, Quadratic Time

Improvement 3. Almost Linear Algorithm



# Outline

Recursive Descent Algorithm is Expensive

Improvement 1: Linear Space, Exponential Time

Improvement 2. Linear Space, Quadratic Time

Improvement 3. Almost Linear Algorithm



# Example from the Previous Lecture

## Example

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

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

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:

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

- ▶ Problem: Duplicate occurrences of the same variables 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

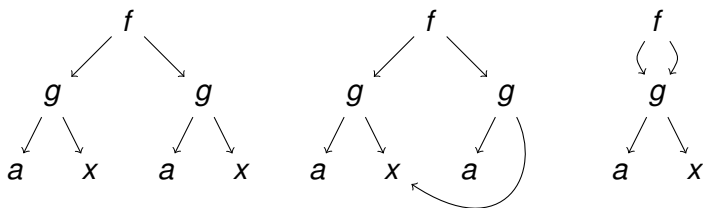
- ▶ Convention: Nodes and terms the term dags represent will not be distinguished.
- ▶ Example: “node”  $f(a, x)$  is a node labeled with  $f$  and having two arcs to  $a$  and to  $x$ .

# 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 for a given term in  $O(n)$  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.



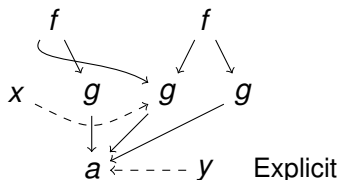
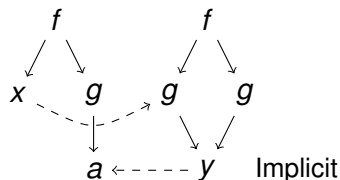
# Term Dags

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  $f(x, g(a))$  and  $f(g(y), g(y))$ , and two applications of the substitution  $\{x \mapsto g(a), y \mapsto a\}$  on it.



# 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.

# Outline

Recursive Descent Algorithm is Expensive

Improvement 1: Linear Space, Exponential Time

Improvement 2. Linear Space, Quadratic Time

Improvement 3. Almost Linear Algorithm



# Recursive Descent Algorithm (RDA) on Term Dags

## Assumptions:

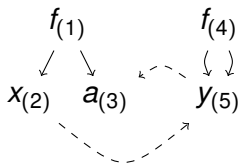
- ▶ Dags consist of nodes.
- ▶ Any node in a given dag defines a unique subdag (consisting of the nodes which can be reached from this node), and thus a unique subterm.
- ▶ Two different types of nodes: variable nodes and function nodes.
- ▶ Information at function nodes:
  - ▶ The name of the function 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

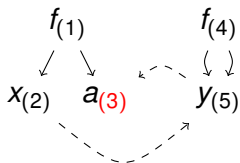


# Auxiliary procedures for the RDA on Term Dags

- ▶ `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)`

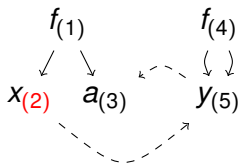


# Auxiliary procedures for the RDA on Term Dags

- ▶ `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)=`

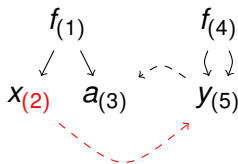


# Auxiliary procedures for the RDA on Term Dags

- ▶ `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)=`

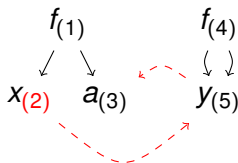


# Auxiliary procedures for the RDA on Term Dags

- ▶ `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)=`

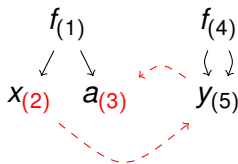


# Auxiliary procedures for the RDA on Term Dags

- ▶ `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  $v$ .

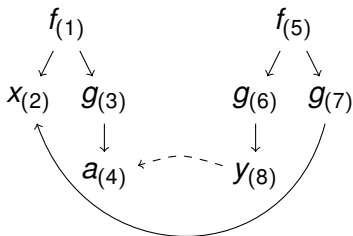


# Auxiliary procedures for the RDA on Term Dags

► `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



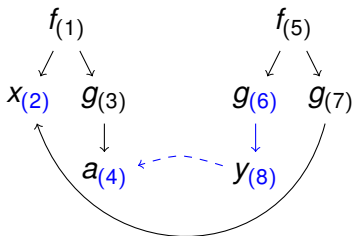
# Auxiliary procedures for the RDA on Term Dags

► 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

►  $\text{Occur}(2,6)=\text{False}$



# Auxiliary procedures for the RDA on Term Dags

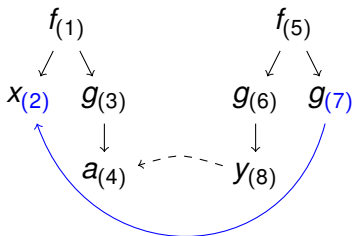
► `Occur`:

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

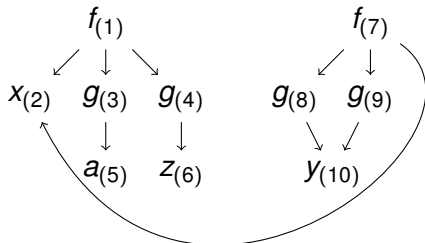
```
else if function-symbol(u)  $\neq$  function-symbol(v)  
then  
    return False;                                /* Symbol clash */  
else  
    n := arity(function-symbol(u));  
    (u1, ..., un) := succ-list(u);  
    (v1, ..., vn) := succ-list(v);  
    i := 0; bool := True;  
  
    while i ≤ n and bool do  
        i := i + 1; bool := Unify1(Find(ui), Find(vi));  
        /* Decomposition */  
    end  
    return bool
```

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



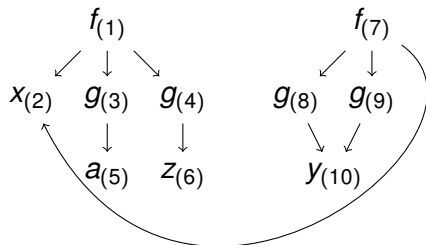
# RDA on Term Dags. Example 1

- ▶ 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

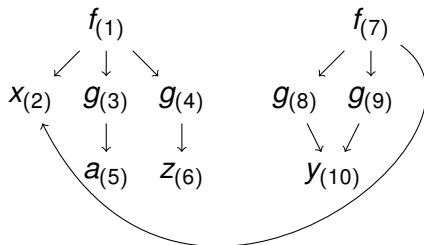
Algorithm run starts with  $\text{Unify}_1(1, 7)$  and continues:



# RDA on Term Dags. Example 1

Algorithm run starts with  $\text{Unify}_1(1, 7)$  and continues:

$\text{Unify}_1(\text{Find}(2), \text{Find}(8))$

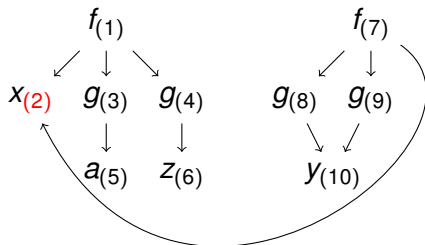


# RDA on Term Dags. Example 1

Algorithm run starts with  $\text{Unify}_1(1, 7)$  and continues:

$\text{Unify}_1(\text{Find}(2), \text{Find}(8))$

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



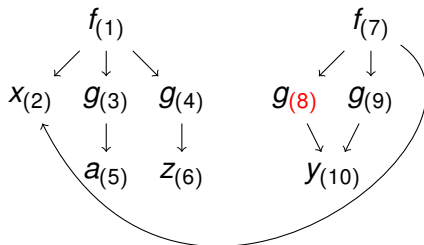
# RDA on Term Dags. Example 1

Algorithm run starts with  $\text{Unify}_1(1, 7)$  and continues:

$\text{Unify}_1(\text{Find}(2), \text{Find}(8))$

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

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



# RDA on Term Dags. Example 1

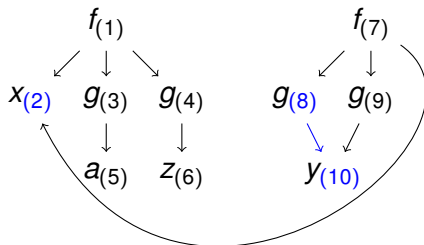
Algorithm run starts with  $\text{Unify}_1(1, 7)$  and continues:

$\text{Unify}_1(\text{Find}(2), \text{Find}(8))$

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

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

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



# RDA on Term Dags. Example 1

Algorithm run starts with  $\text{Unify}_1(1, 7)$  and continues:

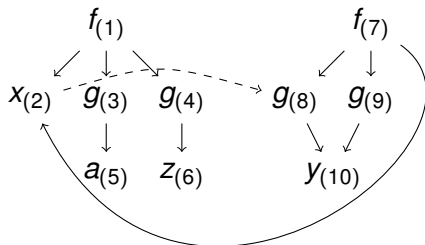
$\text{Unify}_1(\text{Find}(2), \text{Find}(8))$

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

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

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

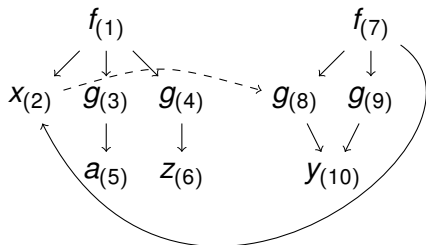
$\text{Union}(2, 8)$



# RDA on Term Dags. Example 1

Algorithm run starts with  $\text{Unify}_1(1, 7)$  and continues:

```
Unify1(Find(2), Find(8))  
  Find(2) = (2)  
  Find(8) = (8)  
  Occur(2, 8) = False  
  Union(2, 8)  
Unify1(Find(3), Find(9))
```



# RDA on Term Dags. Example 1

Algorithm run starts with  $\text{Unify}_1(1, 7)$  and continues:

$\text{Unify}_1(\text{Find}(2), \text{Find}(8))$

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

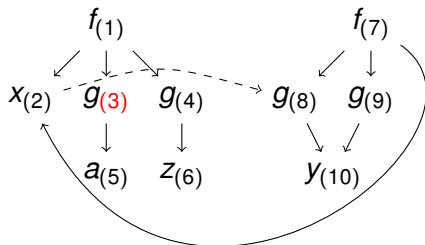
$\text{Find}(8) = (8)$

$\text{Occur}(2, 8) = \textit{False}$

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

$\text{Unify}_1(\text{Find}(3), \text{Find}(9))$

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



# RDA on Term Dags. Example 1

Algorithm run starts with  $\text{Unify}_1(1, 7)$  and continues:

$\text{Unify}_1(\text{Find}(2), \text{Find}(8))$

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

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

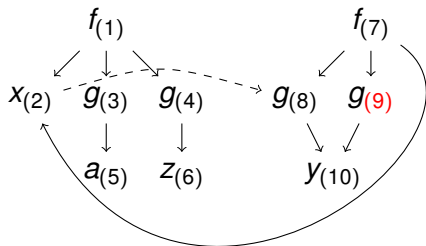
$\text{Occur}(2, 8) = \text{False}$

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

$\text{Unify}_1(\text{Find}(3), \text{Find}(9))$

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

$\text{Find}(9) = (9)$



# RDA on Term Dags. Example 1

Algorithm run starts with  $\text{Unify}_1(1, 7)$  and continues:

$\text{Unify}_1(\text{Find}(2), \text{Find}(8))$

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

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

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

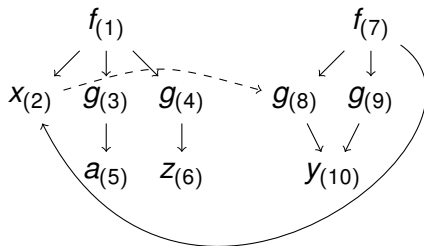
$\text{Union}(2, 8)$

$\text{Unify}_1(\text{Find}(3), \text{Find}(9))$

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

$\text{Find}(9) = (9)$

$\text{Unify}_1(\text{Find}(5), \text{Find}(10))$



# RDA on Term Dags. Example 1

Algorithm run starts with  $\text{Unify}_1(1, 7)$  and continues:

$\text{Unify}_1(\text{Find}(2), \text{Find}(8))$

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

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

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

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

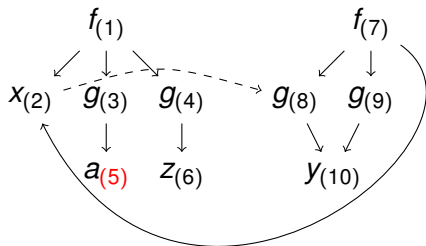
$\text{Unify}_1(\text{Find}(3), \text{Find}(9))$

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

$\text{Find}(9) = (9)$

$\text{Unify}_1(\text{Find}(5), \text{Find}(10))$

$\text{Find}(5) = 5$



# RDA on Term Dags. Example 1

Algorithm run starts with  $\text{Unify}_1(1, 7)$  and continues:

$\text{Unify}_1(\text{Find}(2), \text{Find}(8))$

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

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

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

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

$\text{Unify}_1(\text{Find}(3), \text{Find}(9))$

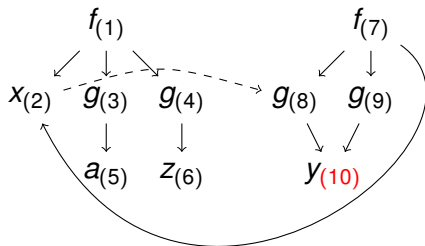
$\text{Find}(3) = (3)$

$\text{Find}(9) = (9)$

$\text{Unify}_1(\text{Find}(5), \text{Find}(10))$

$\text{Find}(5) = 5$

$\text{Find}(10) = 10$



# RDA on Term Dags. Example 1

Algorithm run starts with  $\text{Unify}_1(1, 7)$  and continues:

$\text{Unify}_1(\text{Find}(2), \text{Find}(8))$

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

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

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

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

$\text{Unify}_1(\text{Find}(3), \text{Find}(9))$

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

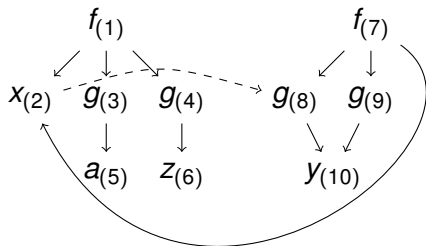
$\text{Find}(9) = (9)$

$\text{Unify}_1(\text{Find}(5), \text{Find}(10))$

$\text{Find}(5) = 5$

$\text{Find}(10) = 10$

$\text{orient}(10, 5)$



# RDA on Term Dags. Example 1

Algorithm run starts with  $\text{Unify}_1(1, 7)$  and continues:

$\text{Unify}_1(\text{Find}(2), \text{Find}(8))$

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

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

$\text{Occur}(2, 8) = \textit{False}$

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

$\text{Unify}_1(\text{Find}(3), \text{Find}(9))$

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

$\text{Find}(9) = (9)$

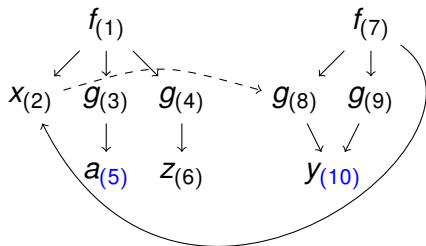
$\text{Unify}_1(\text{Find}(5), \text{Find}(10))$

$\text{Find}(5) = 5$

$\text{Find}(10) = 10$

$\textit{orient}(10, 5)$

$\text{Occur}(10, 5) = \textit{False}$



# RDA on Term Dags. Example 1

Algorithm run starts with  $\text{Unify}_1(1, 7)$  and continues:

$\text{Unify}_1(\text{Find}(2), \text{Find}(8))$

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

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

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

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

$\text{Unify}_1(\text{Find}(3), \text{Find}(9))$

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

$\text{Find}(9) = (9)$

$\text{Unify}_1(\text{Find}(5), \text{Find}(10))$

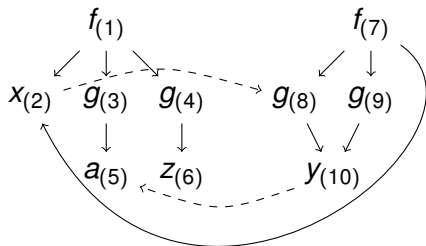
$\text{Find}(5) = 5$

$\text{Find}(10) = 10$

$\text{orient}(10, 5)$

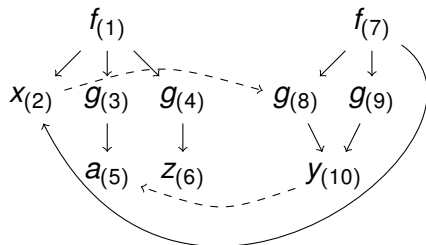
$\text{Occur}(10, 5) = \text{False}$

$\text{Union}(10, 5)$



# RDA on Term Dags. Example 1

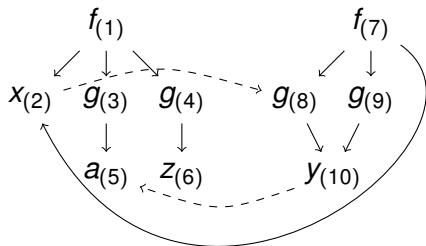
Algorithm run starts with  $\text{Unify}_1(1, 7)$  and continues:



# RDA on Term Dags. Example 1

Algorithm run starts with  $\text{Unify}_1(1, 7)$  and continues:

$\text{Unify}_1(\text{Find}(4), \text{Find}(2))$

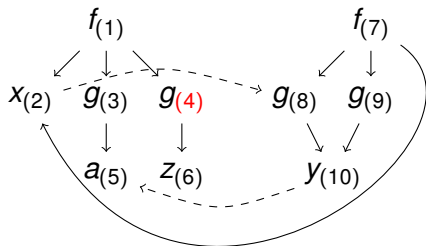


# RDA on Term Dags. Example 1

Algorithm run starts with  $\text{Unify}_1(1, 7)$  and continues:

$\text{Unify}_1(\text{Find}(4), \text{Find}(2))$

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



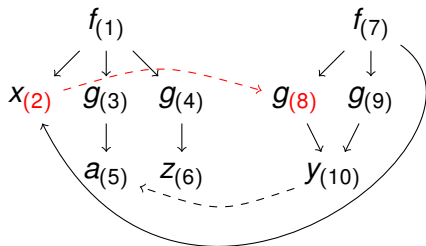
# RDA on Term Dags. Example 1

Algorithm run starts with  $\text{Unify}_1(1, 7)$  and continues:

$\text{Unify}_1(\text{Find}(4), \text{Find}(2))$

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

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



# RDA on Term Dags. Example 1

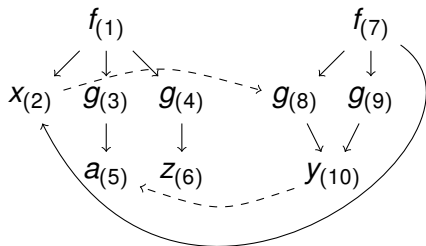
Algorithm run starts with  $\text{Unify}_1(1, 7)$  and continues:

$\text{Unify}_1(\text{Find}(4), \text{Find}(2))$

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

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

$\text{Unify}_1(4, 8)$



# RDA on Term Dags. Example 1

Algorithm run starts with  $\text{Unify}_1(1, 7)$  and continues:

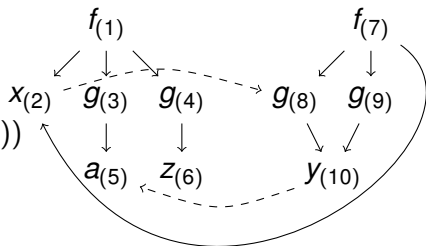
$\text{Unify}_1(\text{Find}(4), \text{Find}(2))$

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

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

$\text{Unify}_1(4, 8)$

$\text{Unify}_1(\text{Find}(6), \text{Find}(10))$



# RDA on Term Dags. Example 1

Algorithm run starts with  $\text{Unify}_1(1, 7)$  and continues:

$\text{Unify}_1(\text{Find}(4), \text{Find}(2))$

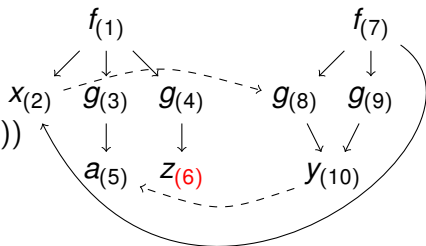
$\text{Find}(4) = 4$

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

$\text{Unify}_1(4, 8)$

$\text{Unify}_1(\text{Find}(6), \text{Find}(10))$

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



# RDA on Term Dags. Example 1

Algorithm run starts with  $\text{Unify}_1(1, 7)$  and continues:

$\text{Unify}_1(\text{Find}(4), \text{Find}(2))$

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

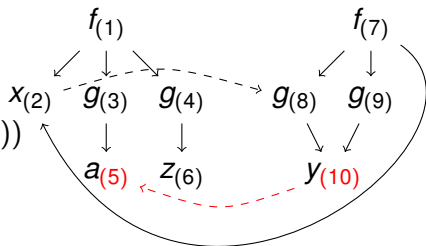
$\text{Find}(2) = 8$

$\text{Unify}_1(4, 8)$

$\text{Unify}_1(\text{Find}(6), \text{Find}(10))$

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

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





# RDA on Term Dags. Example 1

Algorithm run starts with  $\text{Unify}_1(1, 7)$  and continues:

$\text{Unify}_1(\text{Find}(4), \text{Find}(2))$

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

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

$\text{Unify}_1(4, 8)$

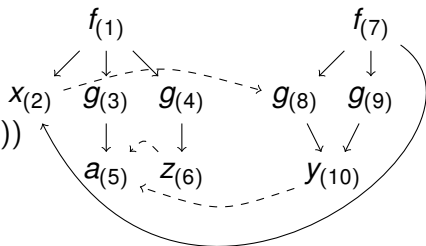
$\text{Unify}_1(\text{Find}(6), \text{Find}(10))$

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

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

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

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



# RDA on Term Dags. Example 1

Algorithm run starts with  $\text{Unify}_1(1, 7)$  and continues:

$\text{Unify}_1(\text{Find}(4), \text{Find}(2))$

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

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

$\text{Unify}_1(4, 8)$

$\text{Unify}_1(\text{Find}(6), \text{Find}(10))$

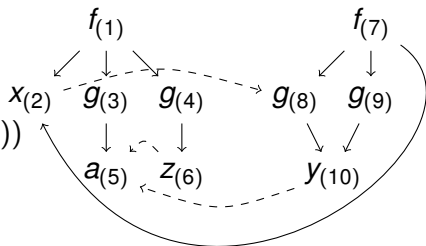
$\text{Find}(6) = 6$

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

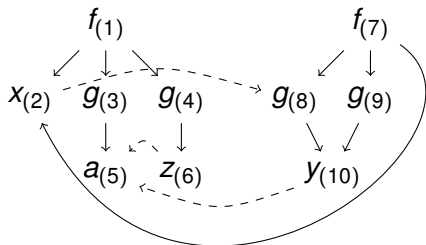
$\text{Occur}(6, 5) = \text{False}$

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

*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.



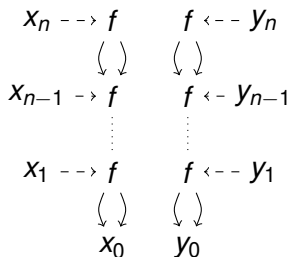
## RDA on Term Dags. Example 2

Consider again the problem:

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

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

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



Exponential number of recursive calls.

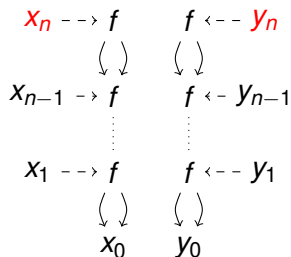
## RDA on Term Dags. Example 2

Consider again the problem:

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

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

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



Exponential number of recursive calls.

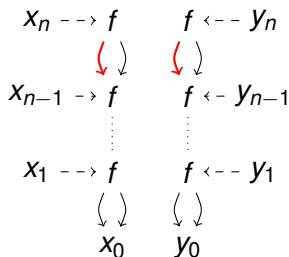
## RDA on Term Dags. Example 2

Consider again the problem:

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

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

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



Exponential number of recursive calls.

## RDA on Term Dags. Example 2

Consider again the problem:

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

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

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

$$\begin{array}{ccc} x_n \dashrightarrow f & & f \dashleftarrow y_n \\ & \left( \downarrow \right) & \left( \downarrow \right) \\ x_{n-1} \rightarrow f & & f \leftarrow y_{n-1} \\ & \vdots & \vdots \\ x_1 \dashrightarrow f & & f \dashleftarrow y_1 \\ & \left( \downarrow \right) & \left( \downarrow \right) \\ & x_0 & y_0 \end{array}$$

Exponential number of recursive calls.

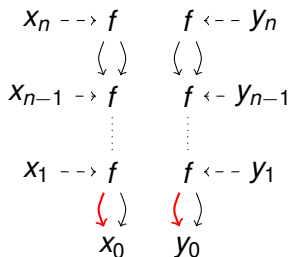
## RDA on Term Dags. Example 2

Consider again the problem:

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

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

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



Exponential number of recursive calls.

## RDA on Term Dags. Example 2

Consider again the problem:

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

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

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

$$\begin{array}{ccc} x_n \dashrightarrow f & & f \dashleftarrow y_n \\ \downarrow \downarrow & & \downarrow \downarrow \\ x_{n-1} \rightarrow f & & f \leftarrow y_{n-1} \\ \vdots & & \vdots \\ x_1 \dashrightarrow f & & f \dashleftarrow y_1 \\ \downarrow \downarrow & & \downarrow \downarrow \\ & & x_0 \rightarrow y_0 \end{array}$$

Exponential number of recursive calls.



## RDA on Term Dags. Example 2

Consider again the problem:

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

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

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

$$\begin{array}{ccc} x_n \dashrightarrow f & & f \dashleftarrow y_n \\ \downarrow \downarrow & & \downarrow \downarrow \\ x_{n-1} \rightarrow f & & f \leftarrow y_{n-1} \\ \vdots & & \vdots \\ x_1 \dashrightarrow f & & f \dashleftarrow y_1 \\ \downarrow \downarrow & & \downarrow \downarrow \\ & & x_0 \rightarrow y_0 \end{array}$$

Exponential number of recursive calls.

## RDA on Term Dags. Example 2

Consider again the problem:

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

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

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

$$\begin{array}{ccc} x_n \dashrightarrow f & & f \dashleftarrow y_n \\ & \left( \downarrow \right) & \left( \downarrow \right) \\ x_{n-1} \rightarrow f & & f \leftarrow y_{n-1} \\ & \vdots & \vdots \\ x_1 \dashrightarrow f & & f \dashleftarrow y_1 \\ & \left( \downarrow \right) & \left( \downarrow \right) \\ & x_0 \rightarrow y_0 & \end{array}$$

Exponential number of recursive calls.

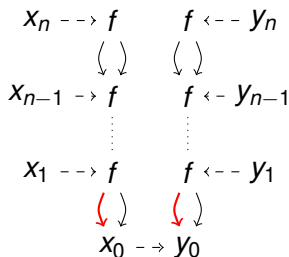
## RDA on Term Dags. Example 2

Consider again the problem:

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

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

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



Exponential number of recursive calls.



## RDA on Term Dags. Example 2

Consider again the problem:

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

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

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

$$\begin{array}{ccc} x_n \dashrightarrow f & & f \dashleftarrow y_n \\ \downarrow \downarrow & & \downarrow \downarrow \\ x_{n-1} \rightarrow f & & f \leftarrow y_{n-1} \\ \vdots & & \vdots \\ x_1 \dashrightarrow f & & f \dashleftarrow y_1 \\ \downarrow \downarrow & & \downarrow \downarrow \\ & & x_0 \rightarrow y_0 \end{array}$$

Exponential number of recursive calls.

## RDA on Term Dags. Example 2

Consider again the problem:

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

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

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

$$\begin{array}{ccc} x_n \dashrightarrow f & & f \dashleftarrow y_n \\ & \left( \downarrow \right) & \left( \downarrow \right) \\ x_{n-1} \rightarrow f & & f \leftarrow y_{n-1} \\ & \vdots & \vdots \\ x_1 \dashrightarrow f & & f \dashleftarrow y_1 \\ & \left( \downarrow \right) & \left( \downarrow \right) \\ & x_0 \rightarrow y_0 & \end{array}$$

Exponential number of recursive calls.

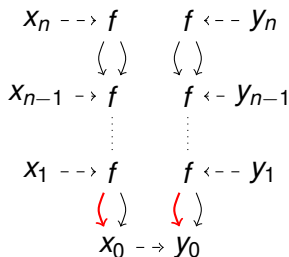
## RDA on Term Dags. Example 2

Consider again the problem:

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

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

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



Exponential number of recursive calls.



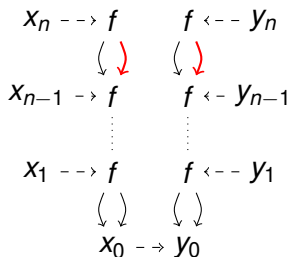
## RDA on Term Dags. Example 2

Consider again the problem:

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

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

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



Exponential number of recursive calls.

## RDA on Term Dags. Example 2

Consider again the problem:

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

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

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

$$\begin{array}{ccc} x_n \dashrightarrow f & & f \dashleftarrow y_n \\ & \left( \downarrow \right) & \left( \downarrow \right) \\ x_{n-1} \rightarrow f & & f \leftarrow y_{n-1} \\ & \vdots & \vdots \\ x_1 \dashrightarrow f & & f \dashleftarrow y_1 \\ & \left( \downarrow \right) & \left( \downarrow \right) \\ & x_0 \rightarrow y_0 & \end{array}$$

Exponential number of recursive calls.

## RDA on Term Dags. Example 2

Consider again the problem:

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

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

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

$$\begin{array}{ccc} x_n \dashrightarrow f & & f \dashleftarrow y_n \\ \downarrow \downarrow & & \downarrow \downarrow \\ x_{n-1} \rightarrow f & & f \leftarrow y_{n-1} \\ \vdots & & \vdots \\ x_1 \dashrightarrow f & & f \dashleftarrow y_1 \\ \downarrow \downarrow & & \downarrow \downarrow \\ & & x_0 \rightarrow y_0 \end{array}$$

Exponential number of recursive calls.

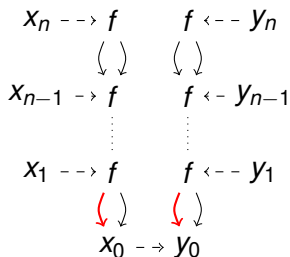
## RDA on Term Dags. Example 2

Consider again the problem:

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

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

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



Exponential number of recursive calls.



## RDA on Term Dags. Example 2

Consider again the problem:

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

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

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

$$\begin{array}{ccc} x_n \dashrightarrow f & & f \dashleftarrow y_n \\ \downarrow \downarrow & & \downarrow \downarrow \\ x_{n-1} \rightarrow f & & f \leftarrow y_{n-1} \\ \vdots & & \vdots \\ x_1 \dashrightarrow f & & f \dashleftarrow y_1 \\ \downarrow \downarrow & & \downarrow \downarrow \\ & & x_0 \rightarrow y_0 \end{array}$$

Exponential number of recursive calls.

# Correctness of RDA for Term Dags

- ▶ Proof is similar as for the RDA. These two algorithm 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.

# 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.



# 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.





# Outline

Recursive Descent Algorithm is Expensive

Improvement 1: Linear Space, Exponential Time

Improvement 2. Linear Space, Quadratic Time

Improvement 3. Almost Linear Algorithm



## 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 function-symbol(u)  $\neq$  function-symbol(v)  
then  
    return False;                                     /* Symbol clash */  
else  
    n := arity(function-symbol(u));  
    (u1, ..., un) := succ-list(u);  
    (v1, ..., vn) := succ-list(v);  
    i := 0; bool := True;  
  
    Union(u,v);  
while i ≤ n and bool do  
    i := i + 1; bool := Unify2(Find(ui), Find(vi));  
    /* Decomposition */  
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:

$$\begin{array}{ccc} x_n & \dashrightarrow & f \\ & \left( \downarrow \right) & \\ x_{n-1} & \dashrightarrow & f \\ & \vdots & \\ x_1 & \dashrightarrow & f \\ & \left( \downarrow \right) & \\ & x_0 & \end{array} \quad \begin{array}{ccc} f & \dashleftarrow & y_n \\ & \left( \downarrow \right) & \\ f & \dashleftarrow & y_{n-1} \\ & \vdots & \\ f & \dashleftarrow & y_1 \\ & \left( \downarrow \right) & \\ & y_0 & \end{array}$$



# Quadratic Algorithm. Example

The same example that revealed exponential behavior of RDA:

$$\begin{array}{ccc} x_n & \dashrightarrow & f \\ & \left( \downarrow \right) & \\ x_{n-1} & \dashrightarrow & f \\ & \vdots & \\ x_1 & \dashrightarrow & f \\ & \left( \downarrow \right) & \\ & x_0 & \end{array} \quad \begin{array}{ccc} f & \dashleftarrow & y_n \\ & \left( \downarrow \right) & \\ f & \dashleftarrow & y_{n-1} \\ & \vdots & \\ f & \dashleftarrow & y_1 \\ & \left( \downarrow \right) & \\ & y_0 & \end{array}$$



# Quadratic Algorithm. Example

The same example that revealed exponential behavior of RDA:

$$\begin{array}{ccc} x_n & \dashrightarrow f & \dashrightarrow f \dashleftarrow y_n \\ & \left( \downarrow \right) & \left( \downarrow \right) \\ x_{n-1} & \dashrightarrow f & f \dashleftarrow y_{n-1} \\ & \vdots & \vdots \\ x_1 & \dashrightarrow f & f \dashleftarrow y_1 \\ & \left( \downarrow \right) & \left( \downarrow \right) \\ & x_0 & y_0 \end{array}$$

# Quadratic Algorithm. Example

The same example that revealed exponential behavior of RDA:

$$\begin{array}{ccc} x_n & \dashrightarrow f & \dashrightarrow f \dashleftarrow y_n \\ & \downarrow \downarrow & \downarrow \downarrow \\ x_{n-1} & \dashrightarrow f & f \dashleftarrow y_{n-1} \\ & \vdots & \vdots \\ x_1 & \dashrightarrow f & f \dashleftarrow y_1 \\ & \downarrow \downarrow & \downarrow \downarrow \\ & x_0 & y_0 \end{array}$$



# Quadratic Algorithm. Example

The same example that revealed exponential behavior of RDA:

$$\begin{array}{ccc} x_n & \dashrightarrow f & \dashrightarrow f \dashleftarrow y_n \\ & \left( \downarrow \right) & \left( \downarrow \right) \\ x_{n-1} & \dashrightarrow f & \dashrightarrow f \dashleftarrow y_{n-1} \\ & \vdots & \vdots \\ x_1 & \dashrightarrow f & \dashrightarrow f \dashleftarrow y_1 \\ & \left( \downarrow \right) & \left( \downarrow \right) \\ & x_0 & y_0 \end{array}$$

# Quadratic Algorithm. Example

The same example that revealed exponential behavior of RDA:

$$\begin{array}{ccc} x_n & \dashrightarrow f & \dashrightarrow f \dashleftarrow y_n \\ & \left( \downarrow \right) & \left( \downarrow \right) \\ x_{n-1} & \dashrightarrow f & \dashrightarrow f \dashleftarrow y_{n-1} \\ & \vdots & \vdots \\ x_1 & \dashrightarrow f & \dashrightarrow f \dashleftarrow y_1 \\ & \left( \downarrow \right) & \left( \downarrow \right) \\ & x_0 & y_0 \end{array}$$

# Quadratic Algorithm. Example

The same example that revealed exponential behavior of RDA:

$$\begin{array}{ccccc} x_n & \dashrightarrow & f & \dashrightarrow & f & \dashleftarrow & y_n \\ & & \left( \downarrow \right) & & \left( \downarrow \right) & & \\ x_{n-1} & \dashrightarrow & f & \dashrightarrow & f & \dashleftarrow & y_{n-1} \\ & & \vdots & & \vdots & & \\ x_1 & \dashrightarrow & f & \dashrightarrow & f & \dashleftarrow & y_1 \\ & & \left( \downarrow \right) & & \left( \downarrow \right) & & \\ & & x_0 & & y_0 & & \end{array}$$

# Quadratic Algorithm. Example

The same example that revealed exponential behavior of RDA:

$$\begin{array}{ccccc} x_n & \dashrightarrow & f & \dashrightarrow & f & \dashleftarrow & y_n \\ & & \downarrow & & \downarrow & & \\ x_{n-1} & \dashrightarrow & f & \dashrightarrow & f & \dashleftarrow & y_{n-1} \\ & & \vdots & & \vdots & & \\ x_1 & \dashrightarrow & f & \dashrightarrow & f & \dashleftarrow & y_1 \\ & & \downarrow & & \downarrow & & \\ & & x_0 & & y_0 & & \end{array}$$

# Quadratic Algorithm. Example

The same example that revealed exponential behavior of RDA:

$$\begin{array}{ccccc} x_n & \dashrightarrow & f & \dashrightarrow & f & \dashleftarrow & y_n \\ & & \left( \downarrow \right) & & \left( \downarrow \right) & & \\ x_{n-1} & \dashrightarrow & f & \dashrightarrow & f & \dashleftarrow & y_{n-1} \\ & & \vdots & & \vdots & & \\ x_1 & \dashrightarrow & f & \dashrightarrow & f & \dashleftarrow & y_1 \\ & & \left( \downarrow \right) & & \left( \downarrow \right) & & \\ & & x_0 & \dashrightarrow & y_0 & & \end{array}$$



# Quadratic Algorithm. Example

The same example that revealed exponential behavior of RDA:

$$\begin{array}{ccccc} x_n & \dashrightarrow & f & \dashrightarrow & f & \dashleftarrow & y_n \\ & & \left( \downarrow \right) & & \left( \downarrow \right) & & \\ x_{n-1} & \dashrightarrow & f & \dashrightarrow & f & \dashleftarrow & y_{n-1} \\ & & \vdots & & \vdots & & \\ x_1 & \dashrightarrow & f & \dashrightarrow & f & \dashleftarrow & y_1 \\ & & \left( \downarrow \right) & & \left( \downarrow \right) & & \\ & & x_0 & \dashrightarrow & y_0 & & \end{array}$$



# Quadratic Algorithm. Example

The same example that revealed exponential behavior of RDA:

$$\begin{array}{ccccc} x_n & \dashrightarrow & f & \dashrightarrow & f & \dashleftarrow & y_n \\ & & \left( \downarrow \right) & & \left( \downarrow \right) & & \\ x_{n-1} & \dashrightarrow & f & \dashrightarrow & f & \dashleftarrow & y_{n-1} \\ & & \vdots & & \vdots & & \\ x_1 & \dashrightarrow & f & \dashrightarrow & f & \dashleftarrow & y_1 \\ & & \left( \downarrow \right) & & \left( \downarrow \right) & & \\ & & x_0 & \dashrightarrow & y_0 & & \end{array}$$



# Quadratic Algorithm. Example

The same example that revealed exponential behavior of RDA:

$$\begin{array}{ccccc} x_n & \dashrightarrow & f & \dashrightarrow & f & \dashleftarrow & y_n \\ & & \downarrow & & \downarrow & & \\ & & \downarrow & & \downarrow & & \\ x_{n-1} & \dashrightarrow & f & \dashrightarrow & f & \dashleftarrow & y_{n-1} \\ & & \vdots & & \vdots & & \\ & & \vdots & & \vdots & & \\ x_1 & \dashrightarrow & f & \dashrightarrow & f & \dashleftarrow & y_1 \\ & & \downarrow & & \downarrow & & \\ & & \downarrow & & \downarrow & & \\ & & x_0 & \dashrightarrow & y_0 & & \end{array}$$

# Quadratic Algorithm. Example

The same example that revealed exponential behavior of RDA:

$$\begin{array}{ccccc} x_n & \dashrightarrow & f & \dashrightarrow & f & \dashleftarrow & y_n \\ & & \left( \downarrow \right) & & \left( \downarrow \right) & & \\ x_{n-1} & \dashrightarrow & f & \dashrightarrow & f & \dashleftarrow & y_{n-1} \\ & & \vdots & & \vdots & & \\ x_1 & \dashrightarrow & f & \dashrightarrow & f & \dashleftarrow & y_1 \\ & & \left( \downarrow \right) & & \left( \downarrow \right) & & \\ & & x_0 & \dashrightarrow & y_0 & & \end{array}$$



# 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.



# Outline

Recursive Descent Algorithm is Expensive

Improvement 1: Linear Space, Exponential Time

Improvement 2. Linear Space, Quadratic Time

**Improvement 3. Almost Linear Algorithm**

# 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.

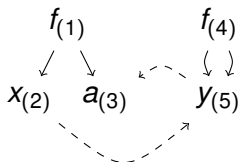
*J. ACM*, 22(2):215–225, 1975.



# 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 `Find( $k$ )` is reached.
  - ▶ In addition, `Collapsing-find` relocates the pointer of all the nodes reached during this process to `Find( $k$ )`.

## Example

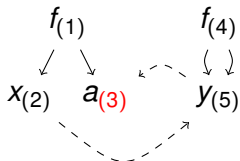


# Auxiliary Procedures for the Almost Linear Algorithm

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

## Example

- ▶  $CF(3)=(3)$

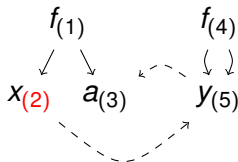


# Auxiliary Procedures for the Almost Linear Algorithm

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

## Example

- ▶  $CF(3)=(3)$
- ▶  $CF(2)=$

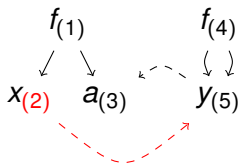


# Auxiliary Procedures for the Almost Linear Algorithm

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

## Example

- ▶  $CF(3)=(3)$
- ▶  $CF(2)=$

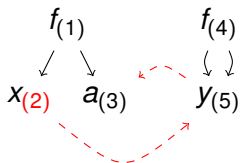


# Auxiliary Procedures for the Almost Linear Algorithm

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

## Example

- ▶  $CF(3)=(3)$
- ▶  $CF(2)=$

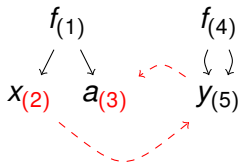


# Auxiliary Procedures for the Almost Linear Algorithm

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

## Example

- ▶  $CF(3)=(3)$
- ▶  $CF(2)=(3)$

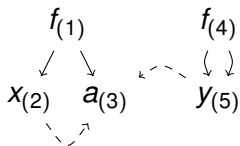


# Auxiliary Procedures for the Almost Linear Algorithm

- ▶ Collapsing-find:
  - ▶ Like `Find` it takes a node  $k$  of a dag as input, and follows the additional pointers until the node `Find( $k$ )` is reached.
  - ▶ In addition, `Collapsing-find` relocates the pointer of all the nodes reached during this process to `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  $\{k \mid \text{Find}(k) = u\}$  larger than the set  $\{k \mid \text{Find}(k) = v\}$  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** *Cyclic-unify*( $k_1, k_2$ ) and *Not-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 ( $k_1, k_2$ )  
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** *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 function-symbol(u)  $\neq$  function-symbol(v)  
then  
    return False;                               /* Symbol clash */  
else  
    n := arity(function-symbol(u));  
    (u1, ..., un) := succ-list(u);  
    (v1, ..., vn) := 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(ui)  
                             Collapsing-find(vi)); /* 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  
 $1, 2, \dots, \omega$ .

Thèse d'État, Université de Paris VII, 1976.



# Complexity

- ▶ The algorithm is almost linear in the number of symbols in original terms:

# 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.



# 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`.



# 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  $\alpha$  is an extremely slowly growing function (functional inverse of Ackerman's function) never exceeding 5 for practical input.



# 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  $\alpha$  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.



# 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  $\alpha$  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

- ▶ 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  $\alpha$  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.



# 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  $\alpha$  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.

