Databases

http://www.dcs.fmph.uniba.sk/~plachetk

/TEACHING/DB2

Tomáš Plachetka, Ján Šturc

Faculty of mathematics, physics and informatics, Comenius University, Bratislava

Summer 2023–2024

Recommended reading

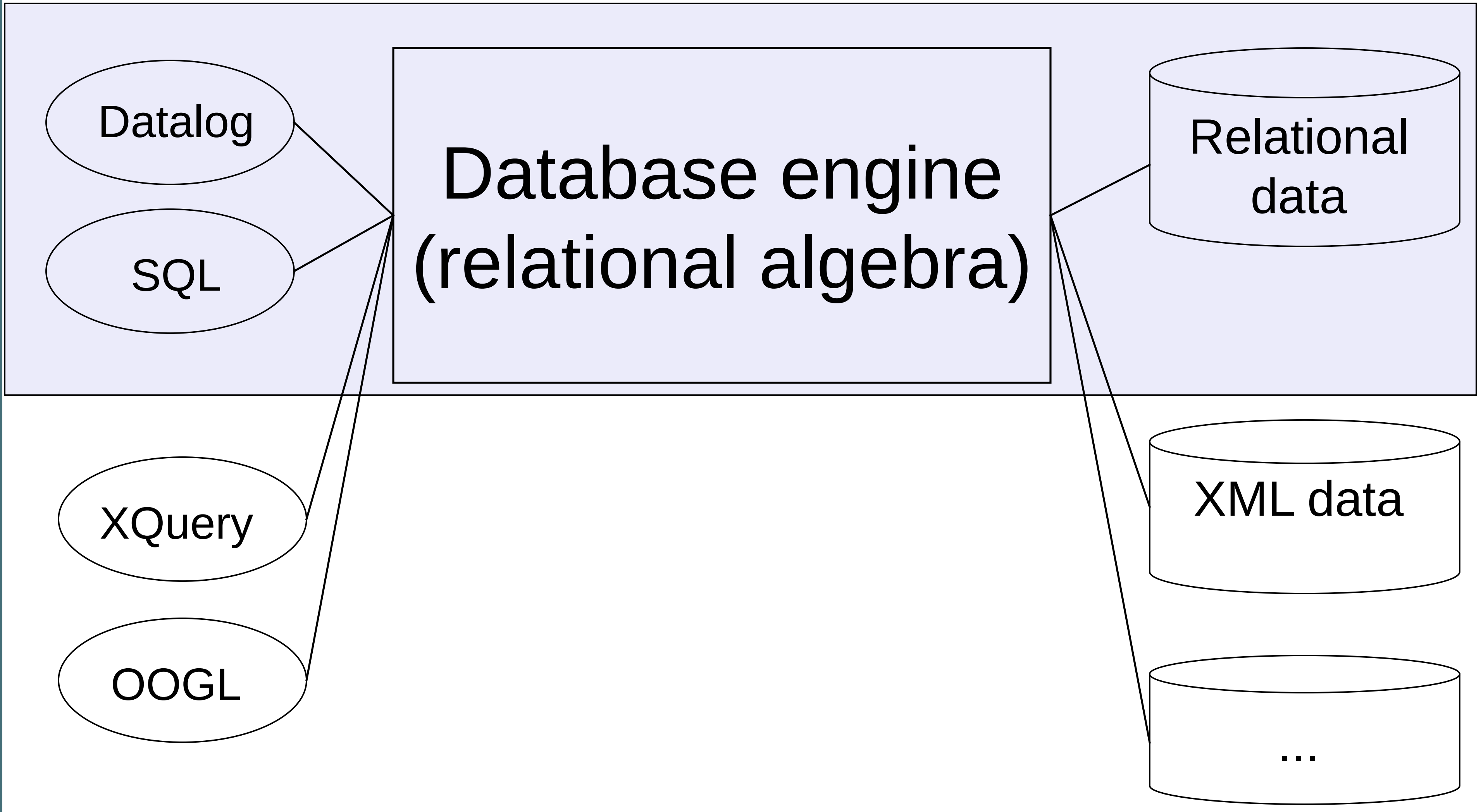
- H. Garcia-Molina, J.D. Ullman, J. Widom: Database Systems, The Complete Book, Prentice Hall 2003
- S. Abiteboul, R. Hull, V. Vianu: Foundations of Databases, Addison-Wesley, 1995
- C. Zaniolo: Advanced Database Systems, Morgan Kaufmann, 1997
- S. W. Dietrich, S. D. Urban: An Advanced Course in Database Systems (Beyond Relational Databases), Pearson Prentice Hall, 2005
- P.A. Bernstein, V. Hadzilacos, N. Goodman: Concurrency Control and Recovery in Database Systems, Addison-Wesley, 1987
- ...

Course outline in short

Implementation and optimisation of database systems with focus on relational (and deductive) databases

- Other kinds of databases are also deployed and studied: network, object, document, ... (Some trends may be just programmers’ escape from mathematics.)
- A graph is a binary **relation** $[N, E \subseteq N \times N]$. A tree is a special graph. Thus replacement of relations with e.g. trees concerns only the language used to talk about the data structures

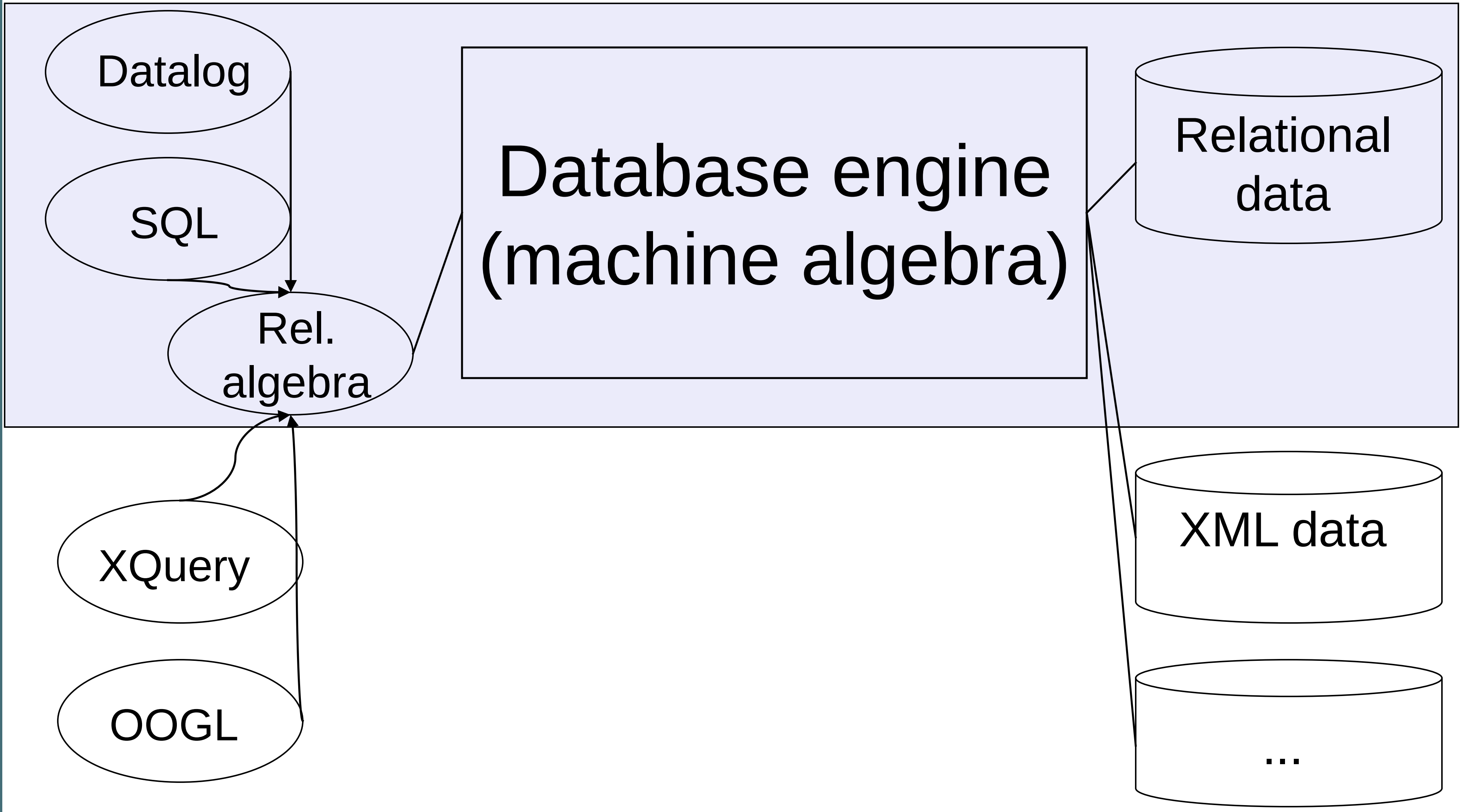
Structure of a modern database system



Frontend (user interface)

Backend

Structure of a modern database system



Frontend (user interface)

Backend

Subproblems

- **Semantics of query languages**, models of programs
- Addition of function symbols to Datalog (and other languages)
- **Computation of queries**: naïve evaluation, Pure Prolog
- Translation of Datalog and SQL to relational algebra, *shredding* (mapping of XML documents to relational schemas)
- **Optimisation of queries**:
  - in Datalog and SQL (naïve evaluation + recursion, indexing)
  - in relational algebra
  - in machine algebra

Function symbols (functors) in Datalog

• A function (functor) is fully described by a name and arity (number of arguments). Functions of arity 0 are "ordinary constants".

• Mathematically, a function can be replaced with a characteristic predicate which “represents the graph of the function”:

$f(X_0, \ldots, X_{n-1}) = y \Leftrightarrow p_f(X_0, \ldots, X_{n-1}, Y)$

But we do not care about the values returned by functions

• We will use **function symbols to represent structured data**. It is worth mentioning that they can be used to **encode numbers** (natural, rational, …)

Function symbols: arithmetics on natural numbers

Arithmetics on natural numbers:

nat(0).

nat(s(X)) $\leftarrow$ nat(X).

add(0, X, X) $\leftarrow$ nat(X).

add(s(X), Y, s(Z)) $\leftarrow$ nat(X), nat(Y), nat(Z), add(X, Y, Z).

le(X, Y) $\leftarrow$ nat(X), nat(Y), add(X, _, Y).   /* less equal */

mul(0, X, 0) $\leftarrow$ nat(X).

mul(s(X), Y, Z) $\leftarrow$ nat(X), nat(Y), nat(Z), mul(X, Y, V), add(V, Y, Z).

Function symbols: arithmetics on natural numbers

Unary encoding of natural numbers:

nat(0). nat(s(X)) ← nat(X).

Natural numbers are represented as terms: **0, s(0), s(s(0)), ...**

nat(X) means that X is a natural number.

add(X, Y, Z) ≡ Z = X + Y

mult(X, Y, Z) ≡ Z = X . Y

le(X, Y) ≡ X <= Y

We can rely on built-in arithmetic predicates instead on the definitions above (built-in predicates are presumably more efficient). However, when details are important, we can define them from scratch

Function symbols: arithmetics

Exercise (related more to computability theory than databases):

Implement all elementary mathematical operations (+, -, ×, /) and functions (sqrt, sin, cos, log, exp) on natural numbers.

Propose a representation of whole and rational numbers

Function symbols: Encoding of data structures

Fragment of an XML document:

```
<addr>
<place>
    <street>Baker_Street</street>
    <nr>221B</nr>
</place>
<city>London
    <pcode>NW1_6XE</pcode>
</city>
</addr>
```

a(p(s(Baker_Street), nr(221B)), c(London, pc(NW1_6XE)))

Function symbols: Encoding of data structures

Binary tree:

- Predicates:

tree(T)	true when T is binary tree
label(L)	true when L is a label of a node

- Function symbols:

null	empty tree
node(L, T1, T2)	tree with a root L, left subtree T1, right subtree T2

- Rules:

tree(null).

tree(node(L, T1, T2)) ← label(L), tree(T1), tree(T2).

Terms

Definition:

• A variable is a **term**.

• Let t_0, t_{n-1} be terms. Let f be an n-ary functional symbol. Then $f(t_0, ..., t_{n-1})$ is a **term**.

Informally, a term is any text which can be used as an argument of a predicate

Note that constants such as 'a, 'john', '7', ... are terms as well. More specifically, they are functors with no arguments, i.e. functors of arity 0

Computation of Datalog programs with functors

Definition: **Herbrand universe** for a Datalog program P is a set of all ground terms which are compositions of functors which appear in P

Herbrand universum of an arbitrary program with only "ordinary constants" is finite (assuming that the extensional database is finite)

Herbrand universum of a program with function symbols can be infinite. Also the result of the naïve evaluation (fixpoint) can be infinite. E.g. the predicate nat(.) „comprises“ all natural numbers, the predicate tree(.) „comprises“ all binary trees etc.

Substitutions

Definition: **A substitution** is a function τ: V $\rightarrow$ T, where V is a set of variables and T is a set of terms. A substitution is applied to terms.

Substitutions are denoted as $\tau = [\{X_i \mapsto t_i\}_{i = 1, \ldots, n}]$

The result of an application of a substitution τ on term *t* is a term $s = t\,\tau$, obtained by replacement of variables common in both *t* and τ by terms determined by τ. The replacement is applied to **all the variables “in parallel” (i.e. once for each variable)**

Ordering and equivalence of terms

Definition: **A term *t* is at least as general as a term *s*** (denoted $t \sqsupseteq s$), if a substitution τ exists such that $s = t\tau$.

Definition: **Terms *t* a *s* are equivalent** (denoted $t \cong s$), if $t \sqsupseteq s$ and $s \sqsupseteq t$

It holds: Two terms are equivalent, $t \cong s$, if they differ only in naming of variables

This structure is called **term algebra**, although it is rather a lattice. Maxima are variables, minima are constant (ground) terms, i.e. terms which do not contain variables)

Composition of substitutions

Definition: **A substitution σ is a composition of substitutions ρ and τ** (denoted $\sigma = \rho \circ \tau$), if for each term t holds $t\sigma = (t\ \tau)\ \rho$

Definition: **A substitution τ is at least as general as a substitution σ** (denoted $\tau \sqsupseteq \sigma$), if a substitution ρ exists such that $\sigma = \tau \circ \rho$

Definition: **Substitutions τ a σ are equivalent** (denoted $\tau \cong \sigma$), if $\tau \sqsupseteq \sigma$ and $\sigma \sqsupseteq \tau$

Composition of substitutions

An empty substitution, (a partial) identical substitution and all substitutions which only permute variables’ names are equivalent

A substitution which is **not equivalent to an identical substitution** is a **specialisation**

Term matching

Term matching is a specialised unification problem (we will deal with general unification later): Consider a term *t* and a ground term *s* (i.e. term *s* does not contain variables). The task is to find whether $t \sqsupseteq s$; and if so, then **find a substitution τ such that $s = t\tau$**

Solution: recursive descent (into the structure of the terms). Begin with empty substitution τ and run the following **procedure match(s, t)**. When it returns TRUE, then the term *t* is at least as general as the term *s* and *τ* is the substitution which proves it (we say that the term *t* matches term *s*). When the procedure returns FALSE, then no solution exists (we say that the term *t* does not match the term *s*)

Term matching

```
boolean match(t, s) {
    if (t is a variable) {
        if ($\tau$(t) is undefined) {
            $\tau$(t) = s;
            return TRUE;
        }
        else
            return $\tau$(t) == s;
    }
    else if (t == f($t_0$, …, $t_{k-1}$) and s == f($s_0$, … $s_{k-1}$)) {
        for (i = 0; i < k; i++) {
            if (! match($t_i$, $s_i$))
                return FALSE;
        }
        return TRUE;
    }
    else
        return FALSE;
}
```

Unification: variants of tasks

Unification solves equations in term algebra. There are several variants:

1. Let t and s be terms. Find the most general substitutions τ and σ such that $t\ \tau = s\ \sigma$
2. Let t and s be terms. Find the most general substitution σ such that $t\ \sigma = s\ \sigma$. (This is a classical mathematical task on solving equations.)
3. Let t and s be terms. Find the most general substitutions ρ and σ such that $t\ \rho\ \sigma = s\ \sigma$. This variant is called weak unification.
4. If terms are bound by a more specialised algebra than equality of terms (on the level of texts), then we talk about paramodulation. „Modulo“ is understood as “with respect to a set of additional properties”. (For example, in a commutative algebra, 2+3 unifies with 3+2.)

Unification: variants of tasks

We will only consider variant 2:

Let t and s be terms. Find the most general substitution σ such that $t\,\sigma = s\,\sigma$

Notes on the other variants:

- The most general solution of variant 3 is obtained so that ρ renames the variables in *t* in such a way that they are distinct from variables in *s*. We denote $t' = t\,\rho$ and solve variant 2: $t'\,\sigma = s\,\sigma$
- Solution to variant 1 can be obtained from the solution of variant 3 by setting $\tau = \rho \circ \sigma$
- Solution to variant 1 with the use of the previous two statements is the most general solution to variant 1

Unification: algorithms

Two main approaches to unification (this is only a selection of representative algorithms):

- **Manipulation of the set of equations**: J.Herbrand (1930), A.Martelli, U.Montanari (1982)
- **Manipulation of trees or dags which represent terms**: J.A.Robinson (1965), J.Corbin, M.Bidoit (1983), *G.Escalada-Imaz, M.Ghallab (1988)*, I.Prívara, P.Ružička (1989)

Unification: Herbrand? (1930), Martelli-Montanari (1982)

Let E be a system of equations. Let σ be initially an empty substitution.

```
while ($E \neq \emptyset$) {
    select an equation $e \in E$;
    $E := E - \{e\}$;
    if (e is $X = t$ or e is $t = X$) {
        if (X occurs in t)
            return solution does not exist;
        else {
            $\sigma = \sigma \circ [x \mapsto t]$;
            $E = E[x \mapsto t]$;
        }
    else
        if (e is of the form $f(s_0, \dots, s_{n-1}) = f(t_0, \dots, t_{n-1})$)
            $E = E \cup \{s_0 = t_0, \dots, s_{n-1} = t_{n-1}\}$;
        else return(solution does not exist)
}
```

Unification: Escalada-Ghallab (1988)

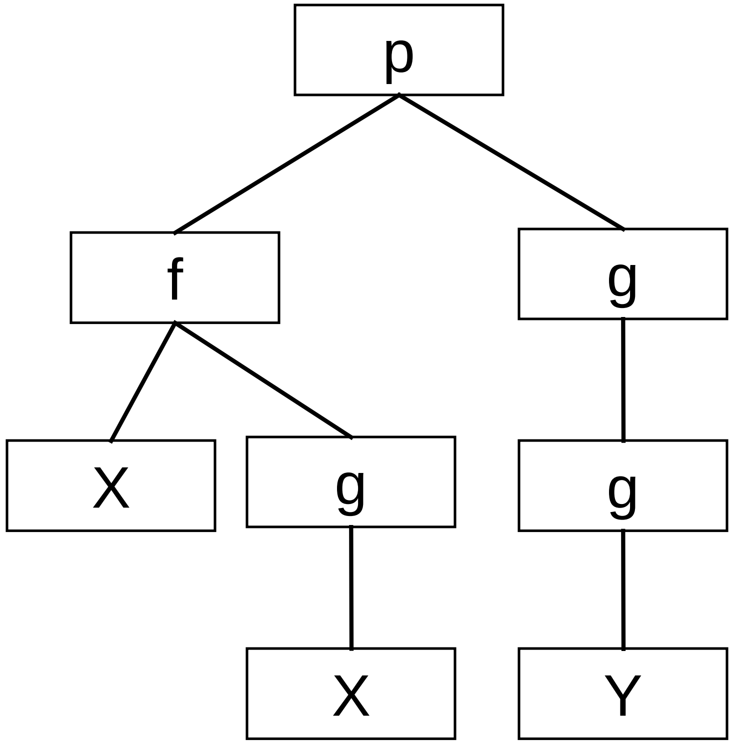
Both the terms are represented with trees or (better) dags

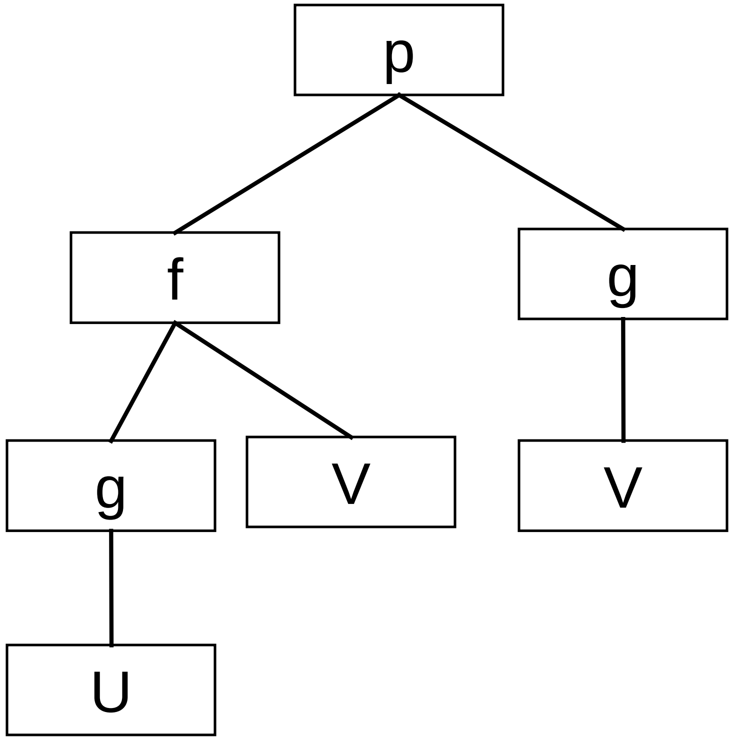
1. Define equivalence of nodes as follows:
    - Roots of the terms are equivalent
    - If two nodes are equivalent, then their sons are also equivalent (in order)
    - Leafs denoted by the same variable or the same constant are equivalent
2. Compute symmetric, reflective and transitive closure from these equivalence classes
3. Assign substitution to a class of equivalence:
    - If a class contains two nodes with different functional symbols, then solution does not exist
    - If a class contains only variables, then choose one and map the others to the chosen one
    - If a class contains variables as well as functors, then map all those variables to the node with the functor
4. Apply occurs-check

Unification: Escalada-Ghallab (1988)

Example:

p(f(X, g(X)), g(g(Y))) = p(f(g(U), V), g(V))

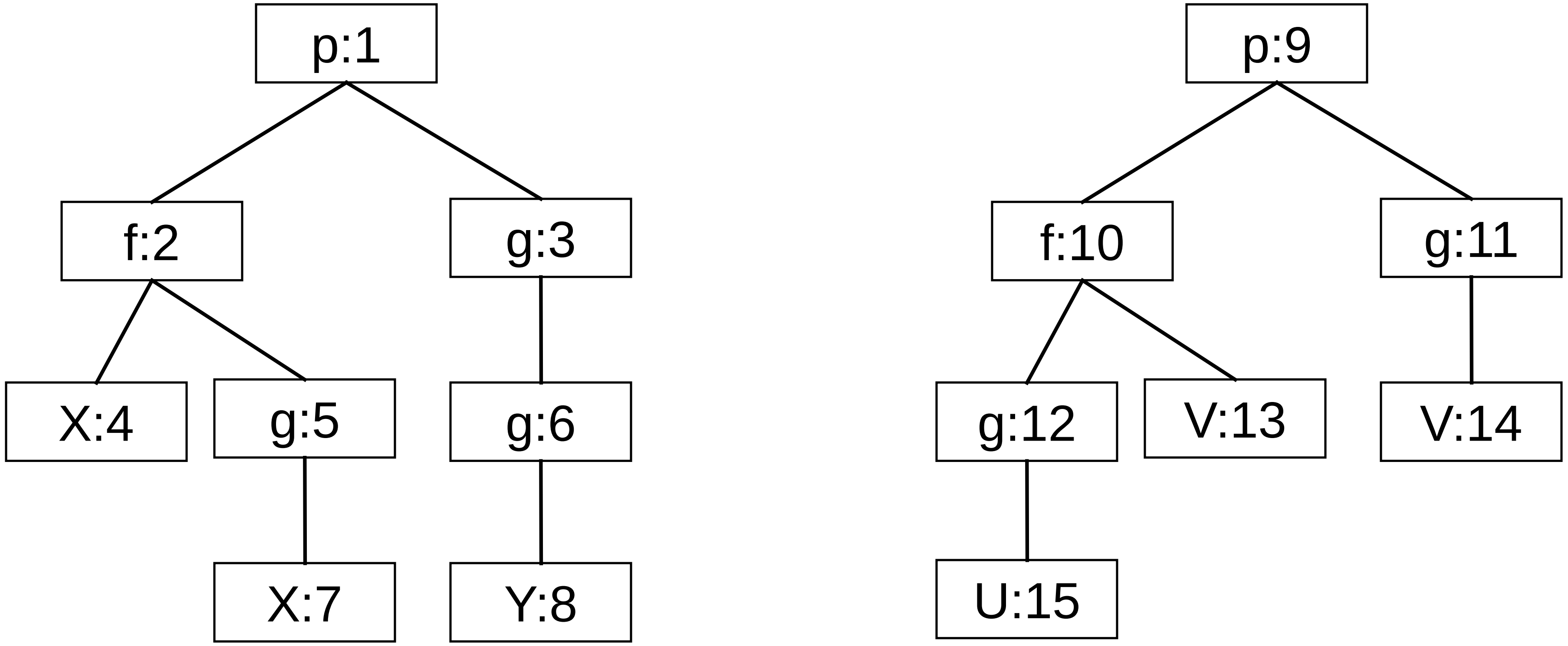




Unification: Escalada-Ghallab (1988)

Example:

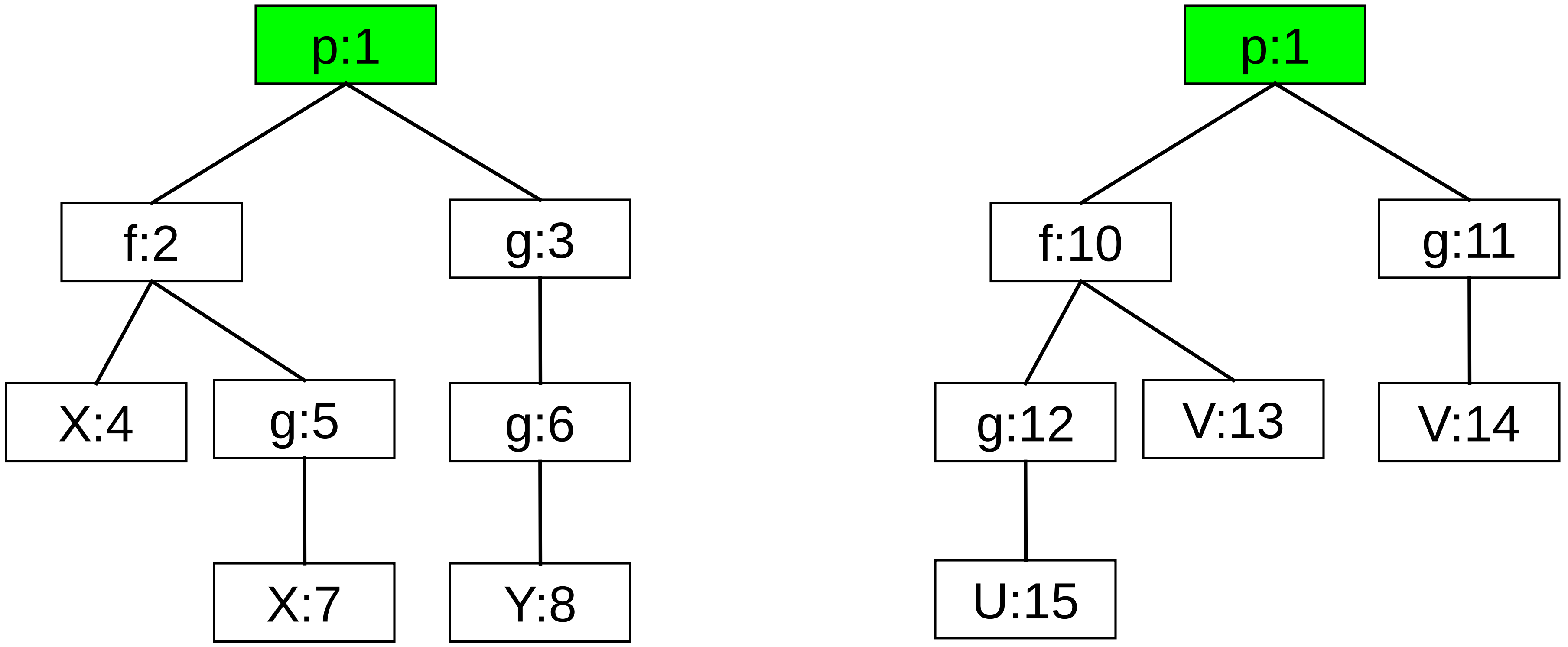
p(f(X, g(X)), g(g(Y))) = p(f(g(U), V), g(V))



Unification: Escalada-Ghallab (1988)

Example:

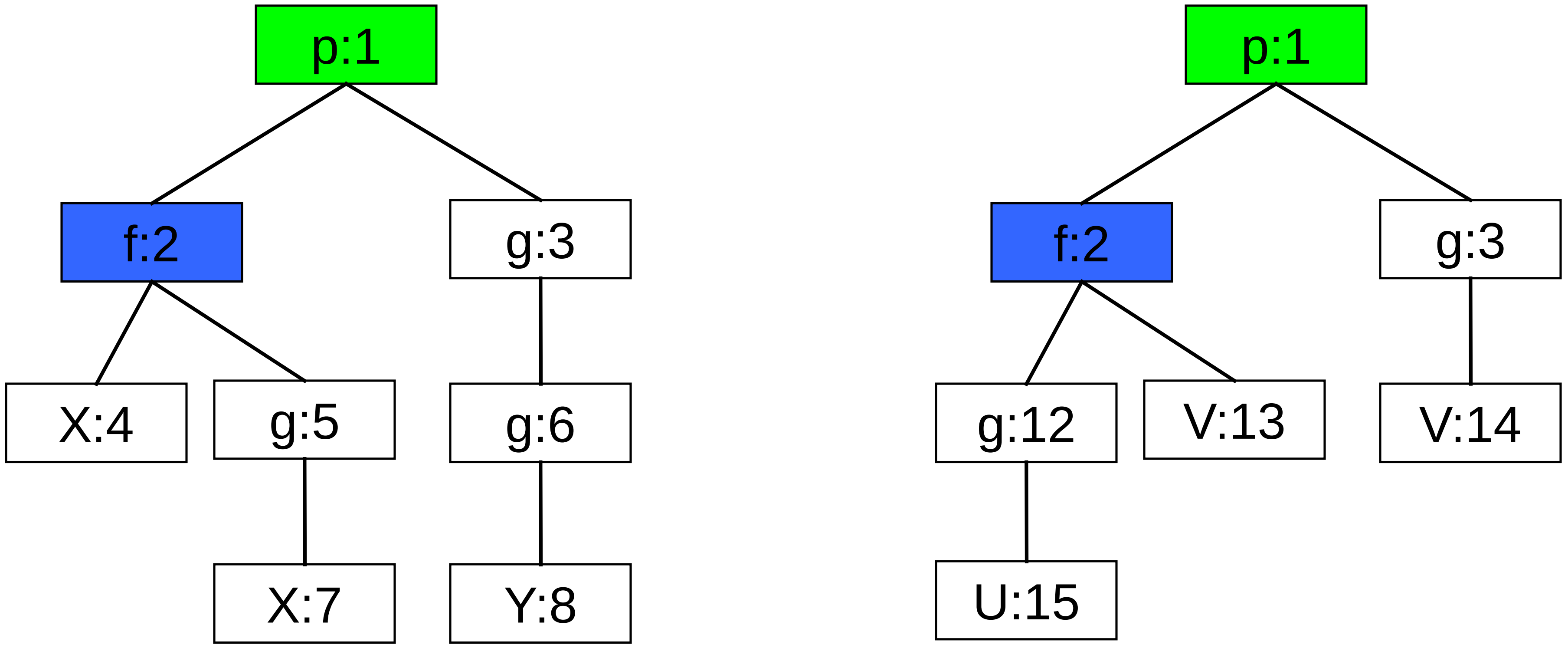
p(f(X, g(X)), g(g(Y))) = p(f(g(U), V), g(V))



Unification: Escalada-Ghallab (1988)

Example:

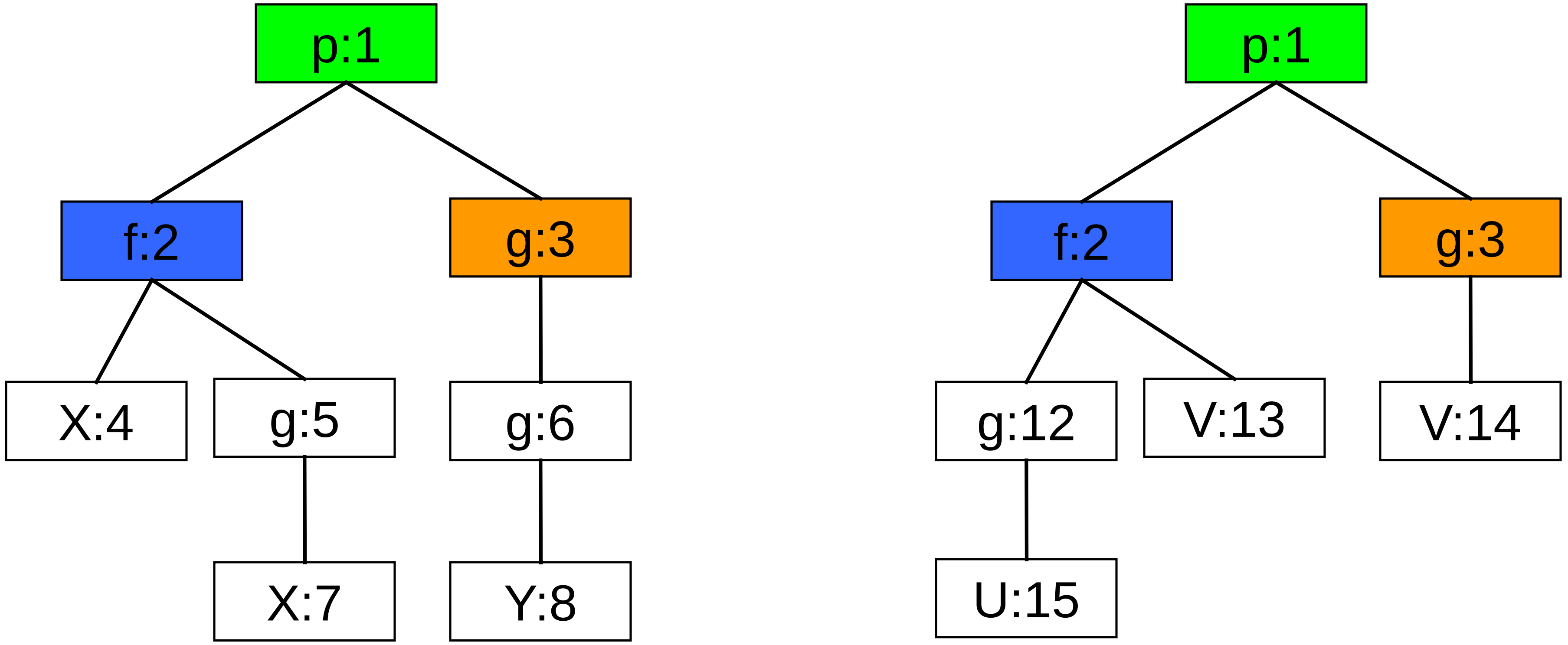
p(f(X, g(X)), g(g(Y))) = p(f(g(U), V), g(V))



Unification: Escalada-Ghallab (1988)

Example:

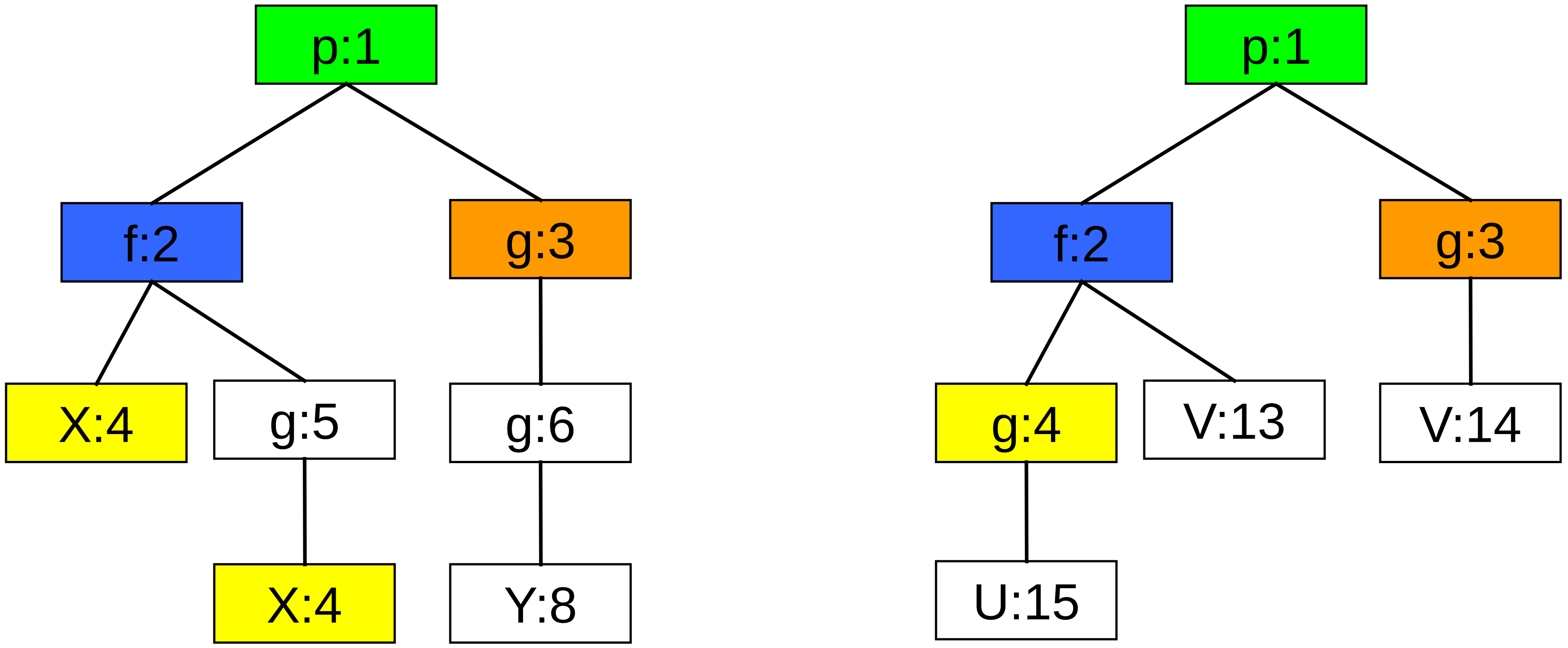
p(f(X, g(X)), g(g(Y))) = p(f(g(U), V), g(V))



Unification: Escalada-Ghallab (1988)

Example:

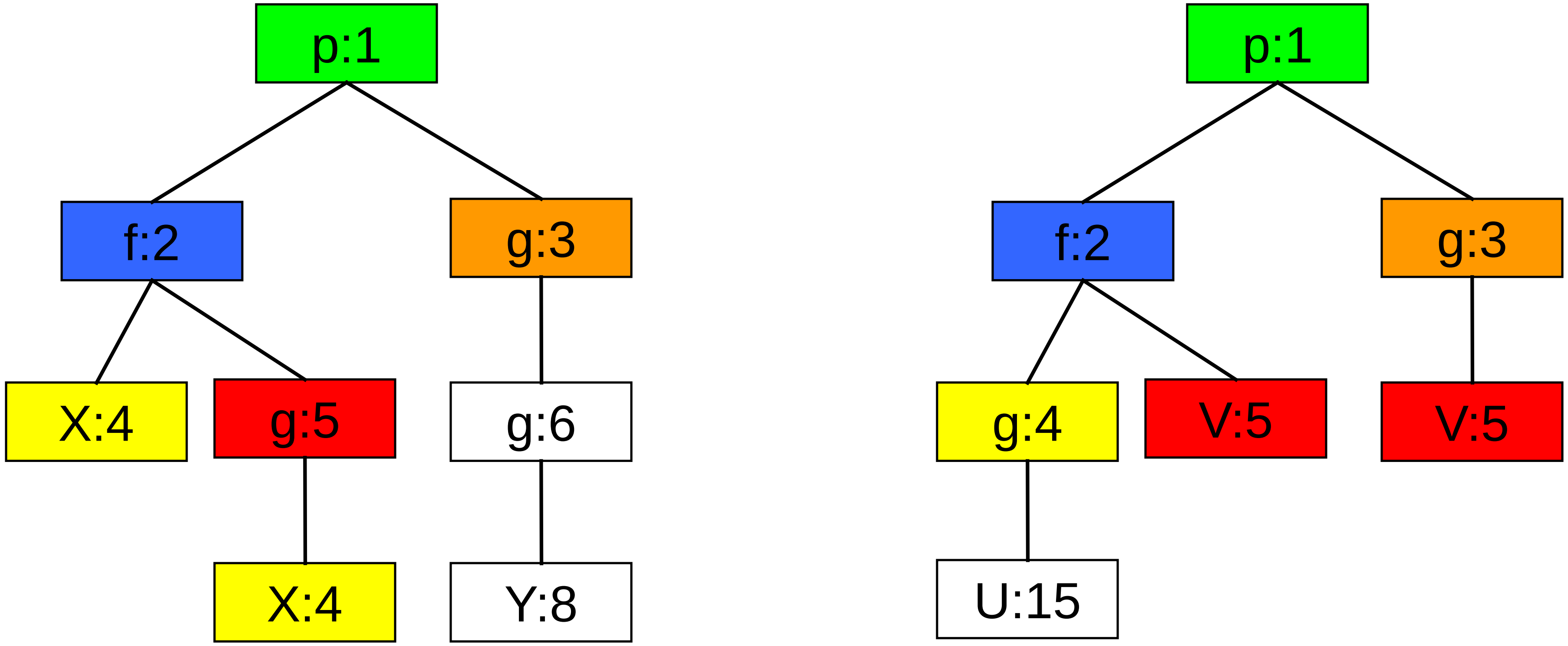
p(f(X, g(X)), g(g(Y))) = p(f(g(U), V), g(V))



Unification: Escalada-Ghallab (1988)

Example:

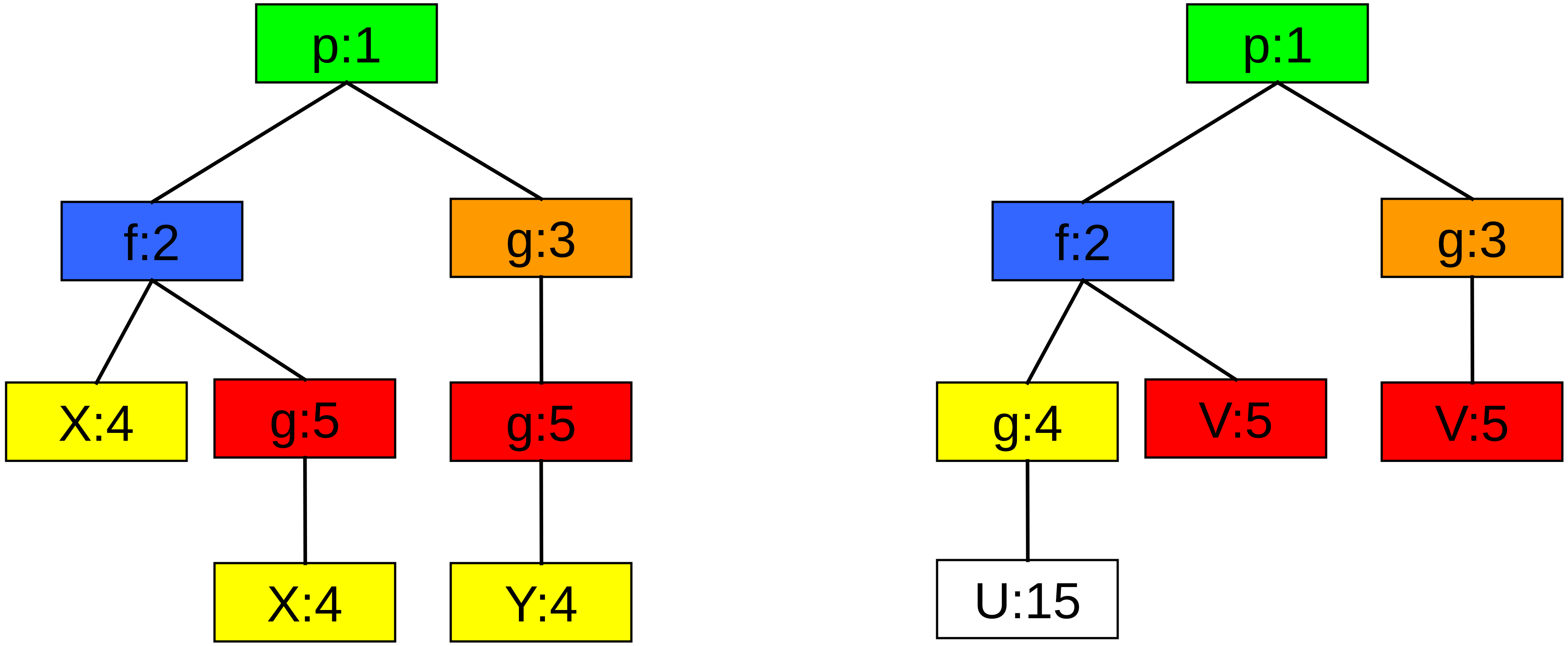
p(f(X, g(X)), g(g(Y))) = p(f(g(U), V), g(V))



Unification: Escalada-Ghallab (1988)

Example:

p(f(X, g(X)), g(g(Y))) = p(f(g(U), V), g(V))

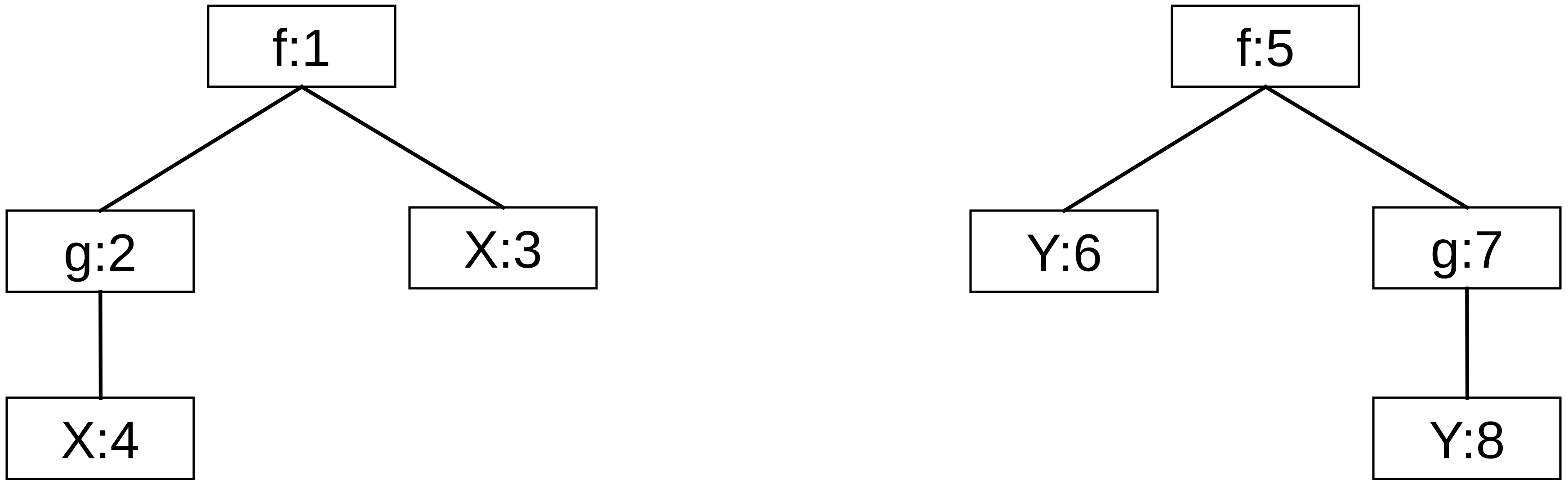


Construction of substitution (mgu): **Y ↦ g(U), X ↦ g(U), V ↦ g(X)**

Explicit solution: X = g(U), Y = g(U), V = g(g(U))

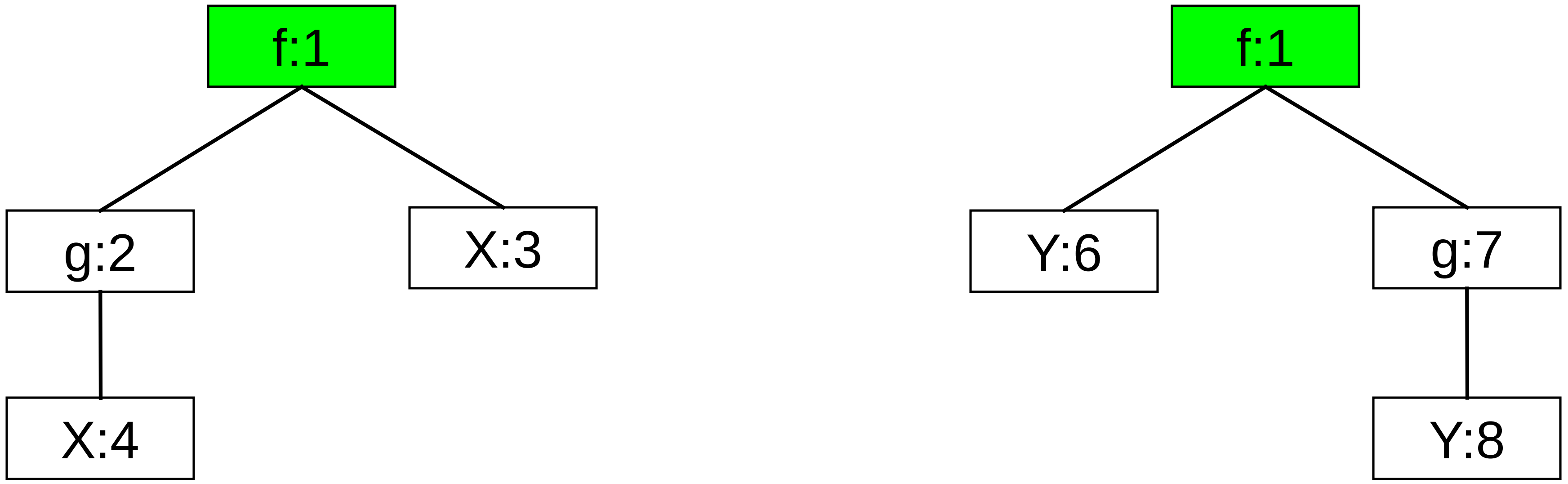
Unification: Escalada-Ghallab (1988)

f(g(X), X) = f(Y, g(Y))



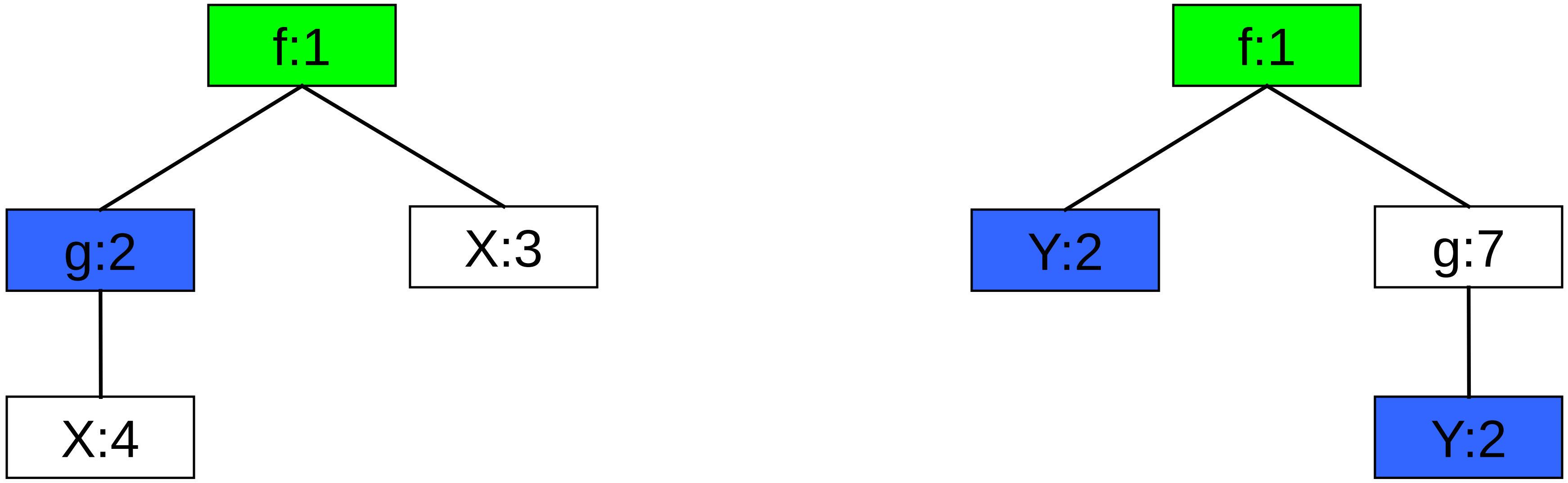
Unification: Escalada-Ghallab (1988)

f(g(X), X) = f(Y, g(Y))



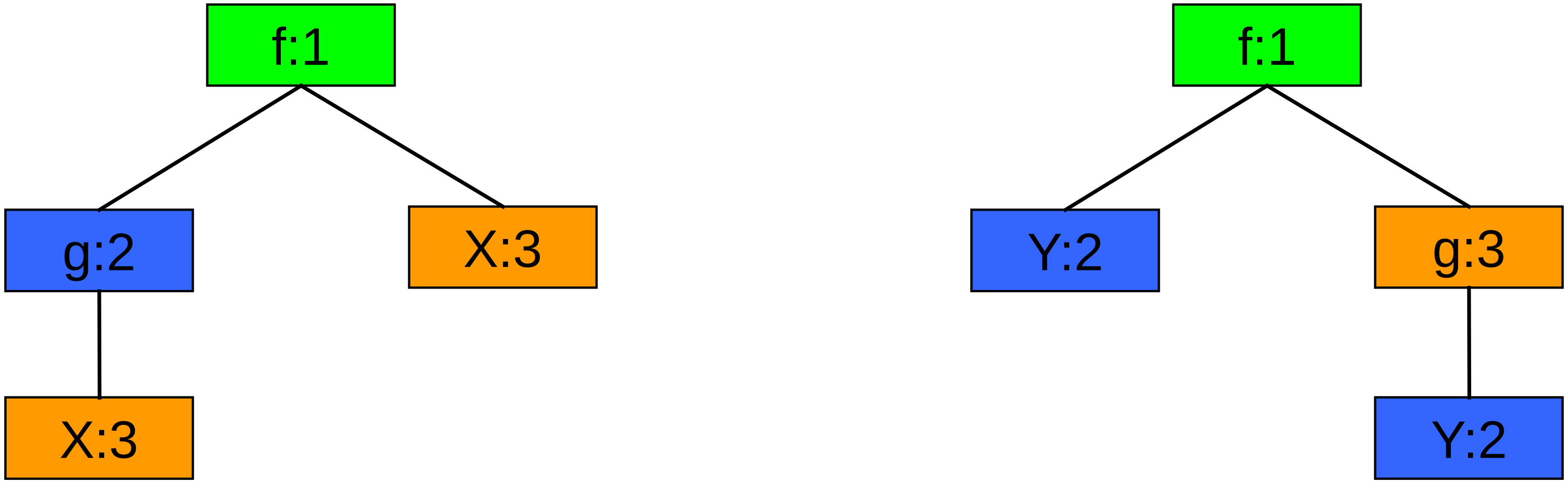
Unification: Escalada-Ghallab (1988)

f(g(X), X) = f(Y, g(Y))



Unification: Escalada-Ghallab (1988)

f(g(X), X) = f(Y, g(Y))



Construction of substitution (mgu): **Y ↦ g(X), X ↦ g(Y)** **CYCLE!**

Explicit solution does not exist: X = g(g(...g(U)...)), Y = g(g(...g(U)...))?

Unification: Escalada-Ghallab (1988)

Asymptotical complexity of the algorithm:

- Traversing both terms creates equivalence classes such that each node is equivalent with itself. ***O(n)***
- Synchronous traversal:
  Find n1
  Find n2
  Union n1 n2
- Complexity of the union-find task (amortised for n operations) is ***$O(n\alpha(n))$***, where α is the inverse Ackermann function, $\alpha(D(n)) = n$
- Complexity of occurs-check test is ***O(n)*** e.g. using topological sorting

```
A(m, n) = if (m == 0) return n + 1;
          else if (n == 0) return A(m - 1, 1);
          else return A(m - 1, A(m, n - 1));
D(n) = A(n, n)
```

Unification: Implementation

Tricks leading to an efficient implementation of unification:

• Represent a term as a DAG (Directed Acyclic Graph)

• During the synchronous traversal, maintain equivalence classes for nodes and equivalence classes for variables

• Whenever Find operation fails, establish a new equivalence class

• If a functor does not match the functor already stored in a class, then immediately terminate. If a variable differs from the variable already stored in a class, make these variables equivalent

• Use a modified occurs-check: test whether a variable in the resulting substitution is equivalent to a variable in the same equivalence class

Unification: explicit expression of the solution

The result of unification is a substitution (mgu)

Beware! An attempt to explicitly express the solution can lead to exponential complexity!

Example:

$f(x_0, x_1, x_2, \ldots, x_{n-1}) = f(g(x_1, x_1), g(x_2, x_2), \ldots, g(x_n, x_n))$

Solution:

$x_0 \mapsto g(x_1, x_1), x_1 \mapsto g(x_2, x_2), x_2 \mapsto g(x_3, x_3), \ldots, x_{n-1} \mapsto g(x_n, x_n)$

Explicit solutions computed backwards:

$x_{n-1} = g(x_n, x_n)$,

$x_{n-2} = g(g(x_n, x_n), g(x_n, x_n))$,

$x_{n-3} = g(g(g(x_n, x_n), g(x_n, x_n)), g(g(x_n, x_n), g(x_n, x_n))$ ...

...

Lengths of terms: 1, 6, 16, 36, 76, 156, 316, ...

Length of i-th term: $L_i = 2L_{i-1}+4$

Translation of Datalog to relational algebra

We want to compute Datalog programs using the following mapping to the relational algebra:

Datalog	**Relational algebra**
predicate	relation
predicate definition ($\leftarrow$)	assignment to a relation (:=), with union ($\cup$)
constant (ground term)	constant string
,	join ($\bowtie$)
not	antijoin ($\triangleright$)
recursion	iterative fixpoint operator (Φ)

Terms in Datalog are used only in arguments of predicates (this includes built-in predicates such as “<“, “>”, “=“, “+”, “-”, …)

However, arguments of predicates can now be complex terms

Datalog to relational algebra: conversion of arguments

If an argument of a predicate is a single variable, it can be directly mapped to an attribute of a relation. However, arguments in a Datalog rule can be more complex, e.g.:

p(f(X, g(X))) ← r(f(X, Y)), r(g(f(Y, g(Y)))).

Subgoal r(f(X, Y)) has **one argument**, which depends on **two variables**.

Subgoal r(g(f(Y, g(Y)))) has one **argument**, which **depends on one variable**.

Arguments of subgoals and the head have a structure determined by functors

How to compute the join r(f(X, Y)), r(g(f(Y, g(Y)))), if predicate r is mapped to the one-attribute relation R?

How to assign the result of the join to the

one-attribute relation P corresponding to the predicate p?

We will extend the relational algebra with two conversion operators:

atov: „arguments to variables“

vtoa: „variables to arguments“

Datalog to relational algebra: conversion of arguments

atov: „arguments to variables“

Let G be an m-ary relation corresponding to subgoal $g(a_1, \dots, a_m)$, with variables $X_1, \dots, X_n$. The contents of the variables must fit the structure of the arguments of the subgoal. The result of atov(g, G) is an **n-ary** relation B. It is a selection (combined with a projection) of values from G which match the arguments of the subgoal $g(a_1, \dots, a_m)$

```
relation atov($g(a_1, \dots, a_m)$, G) {
    $B = \emptyset$;
    for (each tuple $[t_1, \dots, t_m] \in G$) {
        if (match($g(a_1, \dots, a_m)$, $g(t_1, \dots, t_m)$)) {
            /* let $\rho$ denote the matching substitution */
            $B = B \cup [X_1 \rho, \dots, X_n \rho]$;
        }
    return B;
}
```

Datalog to relational algebra: conversion of arguments

vtoa: „variables to arguments“

Let $B(X_1, ..., X_n)$ be **n-ary** relation resulting from the computation of the body of a rule $h(a_1, \ldots, a_m) \leftarrow$ <body>. The contents of the relation B must fit the structure of the head $h(a_1, \ldots, a_m)$. This is computed as $H = vtoa(h(a_1, \ldots, a_m), B)$. Values from B are simply substituted to the variables in the head

```
relation vtoa($h(a_1, \ldots, a_m)$, B) {
    $H = \emptyset$;
    for (each tuple $t \in B$) {
        $\sigma = \emptyset$;
        for ($i = 1$; $i < n$; $i$++)
            $\sigma = \sigma \cup [X_i \mapsto t_i]$;
        $H = H \cup [a_1, \ldots, a_m]\, \sigma$;
    }
    return H;
}
```

Datalog to relational algebra: conversion of arguments

Back to the example

p(f(X, g(X))) ← r(f(X, Y)), r(g(f(Y, g(Y)))).

Predicate r has a corresponding one-attribute relation R, predicate p has a corresponding one-attribute relation P

Translation of the rule to relational algebra:

P = vtoa(p(f(X, g(X))), atov(r(f(X, Y)), R) ⋈ atov(r(g(f(Y, g(Y))), R)))

For example, if R = {f(0, 1), g(f(1, g(1)))}, the computation proceeds as follows:

B1(X, Y) = atov(r(f(X, Y)), R) = {[0, 1]}   /* first record matches */

B2(Y) = atov(r(g(f(Y, g(Y)))), R) = {1}   /* second record matches */

B(X, Y) = B1(X, Y) ⋈ B2(Y) = {[0, 1]}   /* natural join */

P(X) = vtoa(p(f(X, g(X)), B(X, Y)) = {[f(0, g(0))]}   /* conversion to head */

Datalog to relational algebra: conversion of arguments

General scheme of computation of a safe Datalog rule (without negation)

Consider a rule $h \leftarrow g_1, \ldots, g_k$.

Let $G_1, \ldots, G_k$ be relations corresponding to predicates of subgoals $g_1, \ldots, g_k$

1. For each subgoal g_i compute relation B_i = atov(g_i, G_i). (Relation B_i contains all values for variables of subgoal g_i, which satisfy the subgoal g_i.)
2. Compute $B = B_1 \bowtie \ldots \bowtie B_k$. (Relation B contains all candidate values for variables of the rule body, which satisfy all the subgoals.)
3. Convert the result to arguments of the head: H = vtoa(h, B)

We can further optimise the computation in step 2 by early projecting out variables which appear neither in the rest of the join, nor in the head

Computation of programs without neg.: naïve iteration

We can compute single rules (with functors). If there is more than one rule defining the same predicate, then compute its resulting relation as a union of the relations corresponding to the rules

If the predicate dependency graph is acyclic, topological sorting determines the order in which intensional predicates (relations) are computed

If the predicate dependency graph is cyclic, naive iteration can be used to compute the intensional predicates (relations). (Naive iteration is universal, i.e. it can as well be used when the dependency graph is cyclic.)

Computation of programs without neg.: naïve iteration

```
do {
    for (all IDB predicates $p_i$) {
        $P_i = \emptyset$;
    }
    change = FALSE;
    for (all IDB predicates $p_i$) {
        old_$P_i$ = $P_i$;   /* old_$P_i$ is $P_i$ from previous iteration */
        for (all rules r with head $p_i(\ldots)$) {
            /* compute rule r */
            $P_i = P_i \cup$
            vtoa($p_i(\ldots)$, atov(., .) $\bowtie$ atov(., .) $\bowtie$ …);
        }
        if (old_$P_i$ != $P_i$)
            change = TRUE; /* a relation has changed */
    }
} while (change);
```

Computation of programs without neg.: semi-naïve iter.

Differential scheme:

• Each intensional predicate p_i is assigned a difference Δp_i corresponding to a relation ΔP_i (initially an empty set)

• Individual rules are computed as in naïve iteration, but in the body of the rule of p_i, differences Δp_i(...) are used instead of subgoals p_i(...) (other subgoals are unchanged). Results are stored to both new ΔP_i as well as new P_i. However, only a difference against the old P_i is stored into ΔP_i:

$\Delta P_i := \Delta P_i - P_i$

$P_i := P_i \cup \Delta P_i$

• This is iterated until all ΔP_i computed in the last iteration are empty (i.e. no tuple was added)

This scheme can be further optimised by finding an appropriate ordering of rules in the iteration (and using additional differentials in the body of the rules). [R.Ramakrishnan, D.Srivastava, S.Sudarshan: Rule Ordering in Bottom-Up Fixpoint Evaluation in Logic Programs, 1990]

Computation of programs without neg.: semi-naïve iter.

```
for (all IDB predicates $p_i$) {
$P_i = \emptyset$;
$\Delta P_i = \emptyset$;
}

do {
for (all IDB predicates $p_i$) {
    for (all rules r with head $p_i(\dots)$) {
        /* compute rule r */
        $\Delta P_i = \Delta P_i \cup$
        vtoa($p_i(\dots)$, (atov(., $\Delta$.) $\bowtie$ atov(., $\Delta$.) $\bowtie$ …) $- P_i$;
        $P_i = P_i \cup \Delta P_i$;
    }
    if ($\Delta P_i \neq \emptyset$)
        change = TRUE; /* a relation has changed */
}
} while (change);
```

Computation of programs with neg.: semijoin, antijoin

The **natural join, $\bowtie$**, behaves as the **cartesian product when the joined relations have no common attribute**

We will now introduce the **semijoin operator, $\ltimes$** (which will later be needed in some optimalisation techniques):
$r \ltimes s = \pi_{X1, \ldots, Xm} (r \bowtie s)$, where $X_1, \ldots, X_m$ are the attributes of r

Antijoin, $\triangleright$, generalises the set difference operator –:
$r \triangleright s = r - (r \ltimes s)$
Hence, the result of the antijoin are tuples of r which do not join with any tuple of s

We are now ready to compute Datalog programs (possibly with functional symbols) with negation. Negated subgoals are antijoined with the positive part of a rule. The result of the naive (or seminaive) iteration is called a **fixpoint**

Computation of programs with neg.: naïve iteration

```
for (all IDB predicates $p_i$)
$P_i = \varnothing$;
do {
change = FALSE;
for (all IDB predicates $p_i$) {
    old_$P_i$ = $P_i$;   /* old_$P_i$ is $P_i$ from previous iteration */
    for (all rules r with head $p_i$(…)) { /* compute rule r */
        $P_i = \cup_r$
        vtoa($p_i$(...), atov(., .) $\bowtie$ atov(., .) $\bowtie$ …/*positive subgoals*/
         $\triangleright$ atov(., .) $\triangleright$ atov(., .) $\triangleright$ …); /*negated subgoals*/
    }
    if (old_$P_i$ != $P_i$)
        change = TRUE; /* a relation has changed */
}
} while (change);
```