

# **Lecture 8: Constraint Semantics & Resolution**

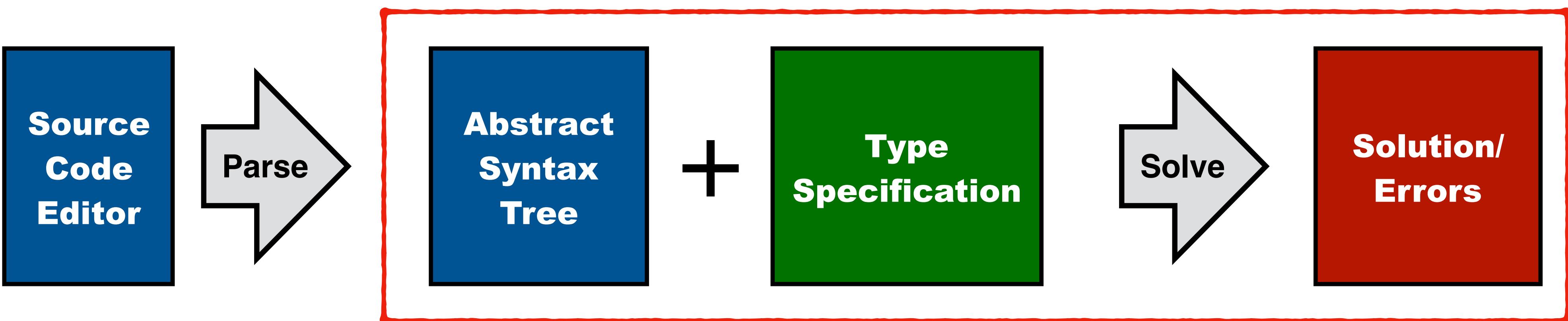
**CS4200 Compiler Construction**

**Hendrik van Antwerpen**

**TU Delft**

**October 2019**

# This lecture



- Type checking with type specifications
- Semantics of a type specification
- Type checking algorithms
- Constraint solving for type specifications
- Term equality and unification

# Reading Material

The following papers add background, conceptual exposition, and examples to the material from the slides. Some notation and technical details have been changed; check the documentation.

This paper describes the next generation of the approach.

Addresses (previously) open issues in expressiveness of scope graphs for type systems:

- Structural types
- Generic types

Addresses open issue with staging of information in type systems.

Introduces Statix DSL for definition of type systems.

OOPSLA 2018

<https://doi.org/10.1145/3276484>

## Scopes as Types

HENDRIK VAN ANTWERPEN, Delft University of Technology, Netherlands

CASPER BACH POULSEN, Delft University of Technology, Netherlands

ARJEN ROUVOET, Delft University of Technology, Netherlands

EELCO VISSER, Delft University of Technology, Netherlands

Scope graphs are a promising generic framework to model the binding structures of programming languages, bridging formalization and implementation, supporting the definition of type checkers and the automation of type safety proofs. However, previous work on scope graphs has been limited to simple, nominal type systems. In this paper, we show that viewing *scopes as types* enables us to model the internal structure of types in a range of non-simple type systems (including structural records and generic classes) using the generic representation of scopes. Further, we show that relations between such types can be expressed in terms of generalized scope graph queries. We extend scope graphs with scoped relations and queries. We introduce Statix, a new domain-specific meta-language for the specification of static semantics, based on scope graphs and constraints. We evaluate the scopes as types approach and the Statix design in case studies of the simply-typed lambda calculus with records, System F, and Featherweight Generic Java.

CCS Concepts: • Software and its engineering → Semantics; Domain specific languages;

Additional Key Words and Phrases: static semantics, type system, type checker, name resolution, scope graphs, domain-specific language

### ACM Reference Format:

Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. 2018. Scopes as Types. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 114 (November 2018), 30 pages. <https://doi.org/10.1145/3276484>

## 1 INTRODUCTION

The goal of our work is to support high-level specification of type systems that can be used for multiple purposes, including reasoning (about type safety among other things) and the implementation of type checkers [Visser et al. 2014]. Traditional approaches to type system specification do not reflect the commonality underlying the name binding mechanisms for different languages. Furthermore, operationalizing name binding in a type checker requires carefully staging the traversals of the abstract syntax tree in order to collect information before it is needed. In this paper, we introduce an approach to the declarative specification of type systems that is close in abstraction to traditional type system specifications, but can be directly interpreted as type checking rules. The approach is based on scope graphs for name resolution, and constraints to separate traversal order from solving order.

Authors' addresses: Hendrik van Antwerpen, Delft University of Technology, Delft, Netherlands, H.vanAntwerpen@tudelft.nl; Casper Bach Poulsen, Delft University of Technology, Delft, Netherlands, C.B.Poulsen@tudelft.nl; Arjen Rouvoet, Delft University of Technology, Delft, Netherlands, A.J.Rouvoet@tudelft.nl; Eelco Visser, Delft University of Technology, Delft, Netherlands, E.Visser@tudelft.nl.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2018 Copyright held by the owner/author(s).

2475-1421/2018/11-ART114

<https://doi.org/10.1145/3276484>

Good introduction to unification, which is the basis of many type inference approaches, constraint languages, and logic programming languages. Read sections 1, and 2.

## CHAPTER 8

# Unification theory

Franz Baader

Wayne Snyder

SECOND READERS: Paliath Narendran, Manfred Schmidt-Schauss, and Klaus Schulz.

### Contents

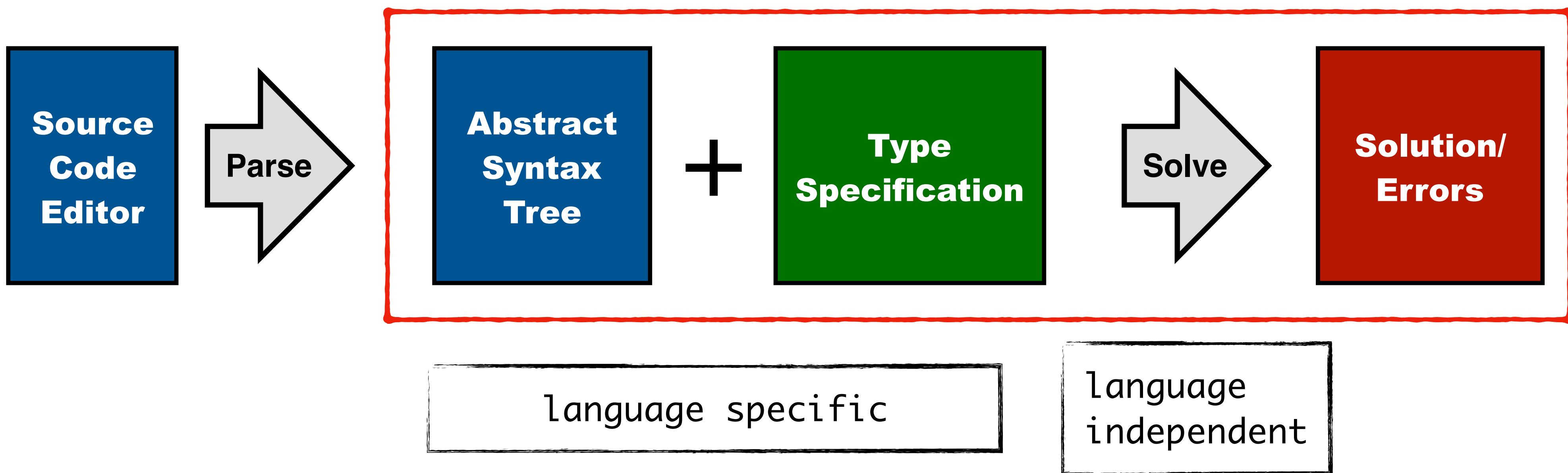
1	Introduction . . . . .	441
1.1	What is unification? . . . . .	441
1.2	History and applications . . . . .	442
1.3	Approach . . . . .	444
2	Syntactic unification . . . . .	444
2.1	Definitions . . . . .	444
2.2	Unification of terms . . . . .	446
2.3	Unification of term <i>dags</i> . . . . .	453
3	Equational unification . . . . .	463
3.1	Basic notions . . . . .	463
3.2	New issues . . . . .	467
3.3	Reformulations . . . . .	469
3.4	Survey of results for specific theories . . . . .	476
4	Syntactic methods for <i>E</i> -unification . . . . .	482
4.1	<i>E</i> -unification in arbitrary theories . . . . .	482
4.2	Restrictions on <i>E</i> -unification in arbitrary theories . . . . .	489
4.3	Narrowing . . . . .	489
4.4	Strategies and refinements of basic narrowing . . . . .	493
5	Semantic approaches to <i>E</i> -unification . . . . .	497
5.1	Unification modulo <i>ACU</i> , <i>ACUI</i> , and <i>AG</i> : an example . . . . .	498
5.2	The class of commutative/monoidal theories . . . . .	502
5.3	The corresponding semiring . . . . .	504
5.4	Results on unification in commutative theories . . . . .	505
6	Combination of unification algorithms . . . . .	507
6.1	A general combination method . . . . .	508
6.2	Proving correctness of the combination method . . . . .	511
7	Further topics . . . . .	513
	Bibliography . . . . .	515
	Index . . . . .	524

Baader et al. “Chapter 8 – Unification Theory.” In *Handbook of Automated Reasoning*, 445–533. Amsterdam: North-Holland, 2001.

<https://www.cs.bu.edu/~snyder/publications/UnifChapter.pdf>

HANDBOOK OF AUTOMATED REASONING  
Edited by Alan Robinson and Andrei Voronkov  
© Elsevier Science Publishers B.V., 2001

# Type Checking with Specifications



# Typing Rules

# Typing Rules

What are typing rules?

# Typing Rules

## What are typing rules?

- Predicates that specify constraints (= rule premises) on their arguments (= the program)

# Typing Rules

## What are typing rules?

- Predicates that specify constraints (= rule premises) on their arguments (= the program)
- Syntax-directed, match on program constructs (at least in Statix)

# Typing Rules

## What are typing rules?

- Predicates that specify constraints (= rule premises) on their arguments (= the program)
- Syntax-directed, match on program constructs (at least in Statix)
- *Specification* of what it means to be well-typed!

# Typing Rules

## What are typing rules?

- Predicates that specify constraints (= rule premises) on their arguments (= the program)
- Syntax-directed, match on program constructs (at least in Statix)
- *Specification* of what it means to be well-typed!

## What are the premises?

# Typing Rules

## What are typing rules?

- Predicates that specify constraints (= rule premises) on their arguments (= the program)
- Syntax-directed, match on program constructs (at least in Statix)
- *Specification* of what it means to be well-typed!

## What are the premises?

- Logical assertions that should hold for well-typed programs

# Typing Rules

## What are typing rules?

- Predicates that specify constraints (= rule premises) on their arguments (= the program)
- Syntax-directed, match on program constructs (at least in Statix)
- *Specification* of what it means to be well-typed!

## What are the premises?

- Logical assertions that should hold for well-typed programs
- Specification language determines what assertions can be made

# Typing Rules

## What are typing rules?

- Predicates that specify constraints (= rule premises) on their arguments (= the program)
- Syntax-directed, match on program constructs (at least in Statix)
- *Specification* of what it means to be well-typed!

## What are the premises?

- Logical assertions that should hold for well-typed programs
- Specification language determines what assertions can be made
  - ▶ Type equality and inequality, name resolution, ...

# Typing Rules

## What are typing rules?

- Predicates that specify constraints (= rule premises) on their arguments (= the program)
- Syntax-directed, match on program constructs (at least in Statix)
- *Specification* of what it means to be well-typed!

## What are the premises?

- Logical assertions that should hold for well-typed programs
- Specification language determines what assertions can be made
  - ▶ Type equality and inequality, name resolution, ...
- Determines the expressiveness of the specification!

# Typing Rules

## What are typing rules?

- Predicates that specify constraints (= rule premises) on their arguments (= the program)
- Syntax-directed, match on program constructs (at least in Statix)
- *Specification* of what it means to be well-typed!

## What are the premises?

- Logical assertions that should hold for well-typed programs
- Specification language determines what assertions can be made
  - ▶ Type equality and inequality, name resolution, ...
- Determines the expressiveness of the specification!

## Solving

# Typing Rules

## What are typing rules?

- Predicates that specify constraints (= rule premises) on their arguments (= the program)
- Syntax-directed, match on program constructs (at least in Statix)
- *Specification* of what it means to be well-typed!

## What are the premises?

- Logical assertions that should hold for well-typed programs
- Specification language determines what assertions can be made
  - ▶ Type equality and inequality, name resolution, ...
- Determines the expressiveness of the specification!

## Solving

- Given an initial predicate that must hold, ...

# Typing Rules

## What are typing rules?

- Predicates that specify constraints (= rule premises) on their arguments (= the program)
- Syntax-directed, match on program constructs (at least in Statix)
- *Specification* of what it means to be well-typed!

## What are the premises?

- Logical assertions that should hold for well-typed programs
- Specification language determines what assertions can be made
  - ▶ Type equality and inequality, name resolution, ...
- Determines the expressiveness of the specification!

## Solving

- Given an initial predicate that must hold, ...
- find an assignment for all logical variables, such that the predicate is satisfied

# Typing Checking

# Typing Checking

Challenges for type checker implementations?

# Typing Checking

## Challenges for type checker implementations?

- Collecting (non-lexical) binding information before use

# Typing Checking

## Challenges for type checker implementations?

- Collecting (non-lexical) binding information before use
- Dealing with unknown (type) values

# Typing Checking

## Challenges for type checker implementations?

- Collecting (non-lexical) binding information before use
- Dealing with unknown (type) values

## Separation of *what* from *how*

# Typing Checking

## Challenges for type checker implementations?

- Collecting (non-lexical) binding information before use
- Dealing with unknown (type) values

## Separation of *what* from *how*

- Typing rules says *what* is a well-typed program

# Typing Checking

## Challenges for type checker implementations?

- Collecting (non-lexical) binding information before use
- Dealing with unknown (type) values

## Separation of *what* from *how*

- Typing rules says *what* is a well-typed program
- Solver says *how* to determine that a program is well-typed

# Typing Checking

## Challenges for type checker implementations?

- Collecting (non-lexical) binding information before use
- Dealing with unknown (type) values

## Separation of *what* from *how*

- Typing rules says *what* is a well-typed program
- Solver says *how* to determine that a program is well-typed

## Separation of computation from program structure

# Typing Checking

## Challenges for type checker implementations?

- Collecting (non-lexical) binding information before use
- Dealing with unknown (type) values

## Separation of *what* from *how*

- Typing rules says *what* is a well-typed program
- Solver says *how* to determine that a program is well-typed

## Separation of computation from program structure

- Typing rules follow the structure of the program

# Typing Checking

## Challenges for type checker implementations?

- Collecting (non-lexical) binding information before use
- Dealing with unknown (type) values

## Separation of *what* from *how*

- Typing rules says *what* is a well-typed program
- Solver says *how* to determine that a program is well-typed

## Separation of computation from program structure

- Typing rules follow the structure of the program
- Solver is flexible in order of resolution

# Typing Checking

## Challenges for type checker implementations?

- Collecting (non-lexical) binding information before use
- Dealing with unknown (type) values

## Separation of *what* from *how*

- Typing rules says *what* is a well-typed program
- Solver says *how* to determine that a program is well-typed

## Separation of computation from program structure

- Typing rules follow the structure of the program
- Solver is flexible in order of resolution

## Approach: reusable solver for the specification language

# Typing Checking

## Challenges for type checker implementations?

- Collecting (non-lexical) binding information before use
- Dealing with unknown (type) values

## Separation of *what* from *how*

- Typing rules says *what* is a well-typed program
- Solver says *how* to determine that a program is well-typed

## Separation of computation from program structure

- Typing rules follow the structure of the program
- Solver is flexible in order of resolution

## Approach: reusable solver for the specification language

- Support logical variables for unknowns and *infer* their values

# Typing Checking

## Challenges for type checker implementations?

- Collecting (non-lexical) binding information before use
- Dealing with unknown (type) values

## Separation of *what* from *how*

- Typing rules says *what* is a well-typed program
- Solver says *how* to determine that a program is well-typed

## Separation of computation from program structure

- Typing rules follow the structure of the program
- Solver is flexible in order of resolution

## Approach: reusable solver for the specification language

- Support logical variables for unknowns and *infer* their values
- Automatically determine correct resolution order

# **Constraint Semantics**

# What gives constraints meaning?

**What is the meaning of constraints?**

# What gives constraints meaning?

What is the meaning of constraints?

```
ty == FUN(ty1,ty2)
Var{x} in s |-> d
ty1 == INT()
```

# What gives constraints meaning?

## What is the meaning of constraints?

- What is a valid solution?

```
ty == FUN(ty1,ty2)
Var{x} in s |-> d
ty1 == INT()
```

# What gives constraints meaning?

## What is the meaning of constraints?

- What is a valid solution?
- Or: in which models are the constraints satisfied?

```
ty == FUN(ty1,ty2)
Var{x} in s |-> d
ty1 == INT()
```

# What gives constraints meaning?

## What is the meaning of constraints?

- What is a valid solution?
- Or: in which models are the constraints satisfied?
- Can we describe this independent of an algorithm to find a solution?

```
ty == FUN(ty1,ty2)
Var{x} in s |-> d
ty1 == INT()
```

# What gives constraints meaning?

## What is the meaning of constraints?

- What is a valid solution?
- Or: in which models are the constraints satisfied?
- Can we describe this independent of an algorithm to find a solution?

```
ty == FUN(ty1,ty2)
Var{x} in s |-> d
ty1 == INT()
```

## When are constraints satisfied?

# What gives constraints meaning?

## What is the meaning of constraints?

- What is a valid solution?
- Or: in which models are the constraints satisfied?
- Can we describe this independent of an algorithm to find a solution?

```
ty == FUN(ty1,ty2)
Var{x} in s |-> d
ty1 == INT()
```

## When are constraints satisfied?

- Formally described by the declarative semantics

# What gives constraints meaning?

## What is the meaning of constraints?

- What is a valid solution?
- Or: in which models are the constraints satisfied?
- Can we describe this independent of an algorithm to find a solution?

```
ty == FUN(ty1,ty2)
Var{x} in s |-> d
ty1 == INT()
```

## When are constraints satisfied?

- Formally described by the declarative semantics
- Written as  $G, \phi \models C$

# What gives constraints meaning?

## What is the meaning of constraints?

- What is a valid solution?
- Or: in which models are the constraints satisfied?
- Can we describe this independent of an algorithm to find a solution?

```
ty == FUN(ty1,ty2)
Var{x} in s |-> d
ty1 == INT()
```

## When are constraints satisfied?

- Formally described by the declarative semantics
- Written as  $G, \phi \models C$
- Satisfied in a model

# What gives constraints meaning?

## What is the meaning of constraints?

- What is a valid solution?
- Or: in which models are the constraints satisfied?
- Can we describe this independent of an algorithm to find a solution?

```
ty == FUN(ty1,ty2)
Var{x} in s |-> d
ty1 == INT()
```

## When are constraints satisfied?

- Formally described by the declarative semantics
- Written as  $G, \phi \models C$
- Satisfied in a model
  - ▶ Substitution  $\phi$  (read: *phi*)

# What gives constraints meaning?

## What is the meaning of constraints?

- What is a valid solution?
- Or: in which models are the constraints satisfied?
- Can we describe this independent of an algorithm to find a solution?

```
ty == FUN(ty1,ty2)
Var{x} in s |-> d
ty1 == INT()
```

## When are constraints satisfied?

- Formally described by the declarative semantics
- Written as  $G, \phi \models C$
- Satisfied in a model
  - ▶ Substitution  $\phi$  (read: *phi*)
  - ▶ Scope graph  $G$

# What gives constraints meaning?

## What is the meaning of constraints?

- What is a valid solution?
- Or: in which models are the constraints satisfied?
- Can we describe this independent of an algorithm to find a solution?

```
ty == FUN(ty1,ty2)
Var{x} in s |-> d
ty1 == INT()
```

## When are constraints satisfied?

- Formally described by the declarative semantics
- Written as  $G, \phi \models C$
- Satisfied in a model
  - ▶ Substitution  $\phi$  (read: *phi*)
  - ▶ Scope graph  $G$
- Describes for every type of constraint when it is satisfied

# Semantics of (a Subset of) Statix Constraints

# Semantics of (a Subset of) Statix Constraints

## Syntax

```
C = t == t          // equality
| r in s |-> d    // name resolution
| C ∧ C           // conjunction
```

# Semantics of (a Subset of) Statix Constraints

## Syntax

```
C = t == t          // equality  
| r in s |-> d    // name resolution  
| C ∧ C          // conjunction
```

## Declarative semantics

# Semantics of (a Subset of) Statix Constraints

## Syntax

```
C = t == t          // equality  
| r in s |-> d    // name resolution  
| C ∧ C          // conjunction
```

## Declarative semantics

$$G, \phi \models t == u \quad \text{if } \phi(t) = \phi(u)$$

# Semantics of (a Subset of) Statix Constraints

## Syntax

```
C = t == t          // equality  
| r in s |-> d    // name resolution  
| C ∧ C          // conjunction
```

## Declarative semantics

$G, \phi \models t == u$	if $\phi(t) = \phi(u)$
$G, \phi \models r \text{ in } s \  -> d$	if $\phi(r) = \text{Var}\{x\}$ and $\phi(d) = \text{Var}\{x\}$ and $\phi(s) = \#i$ and $\text{Var}\{x\}$ resolves to $\text{Var}\{x\}$ from $\#i$ in $G$

# Semantics of (a Subset of) Statix Constraints

## Syntax

```
C = t == t          // equality  
| r in s |-> d    // name resolution  
| C ∧ C          // conjunction
```

## Declarative semantics

$G, \phi \models t == u$	if $\phi(t) = \phi(u)$
$G, \phi \models r \text{ in } s \  -> d$	if $\phi(r) = \text{Var}\{x\}$ and $\phi(d) = \text{Var}\{x\}$ and $\phi(s) = \#i$ and $\text{Var}\{x\}$ resolves to $\text{Var}\{x\}$ from $\#i$ in $G$
$G, \phi \models C_1 \wedge C_2$	if $G, \phi \models C_1$ and $G, \phi \models C_2$

# Using the Semantics

## Program

```
let
  function f1(i2 : int) : int =
    i3 + 1
in
  f4(14)
end
```

## Program constraints

```
ty1 == INT()
INT() == INT()
Var{"i"} in #s1 |-> d1
ty2 == INT()
Var{"f"} in #s0 |-> d2
ty3 == FUN(ty4,ty5)
ty4 == INT()
...
```

# Using the Semantics

## Program

```
let
  function f1(i2 : int) : int =
    i3 + 1
in
  f4(14)
end
```

## Program constraints

```
ty1 == INT()
INT() == INT()
Var{"i"} in #s1 |-> d1
ty2 == INT()
Var{"f"} in #s0 |-> d2
ty3 == FUN(ty4,ty5)
ty4 == INT()
...
```

## Constraint semantics

$G, \phi \models t == u$   
*if*  $\phi(t) = \phi(u)$

$G, \phi \models r \text{ in } s |-> d$   
*if*  $\phi(r) = \text{Var}\{x\}$   
*and*  $\phi(d) = \text{Var}\{x\}$   
*and*  $\phi(s) = \#i$   
*and*  $\text{Var}\{x\}$  resolves to  $\text{Var}\{x\}$  from  $\#i$  in  $G$

$G, \phi \models C_1 \wedge C_2$   
*if*  $G, \phi \models C_1$   
*and*  $G, \phi \models C_2$

# Using the Semantics

## Program

```
let
  function f1(i2 : int) : int =
    i3 + 1
in
  f4(14)
end
```

## Program constraints

```
ty1 == INT()
INT() == INT()
Var{"i"} in #s1 |-> d1
ty2 == INT()
Var{"f"} in #s0 |-> d2
ty3 == FUN(ty4,ty5)
ty4 == INT()
...
...
```

## Unifier $\phi$ (model)

```
 $\phi = \{ \begin{array}{l} ty1 \rightarrow INT(), \\ ty2 \rightarrow INT(), \\ ty3 \rightarrow FUN(INT(),ty5), \\ ty4 \rightarrow INT(), \\ d1 \rightarrow Var{"i"}, \\ d2 \rightarrow Var{"f"} \end{array} \}$ 
```

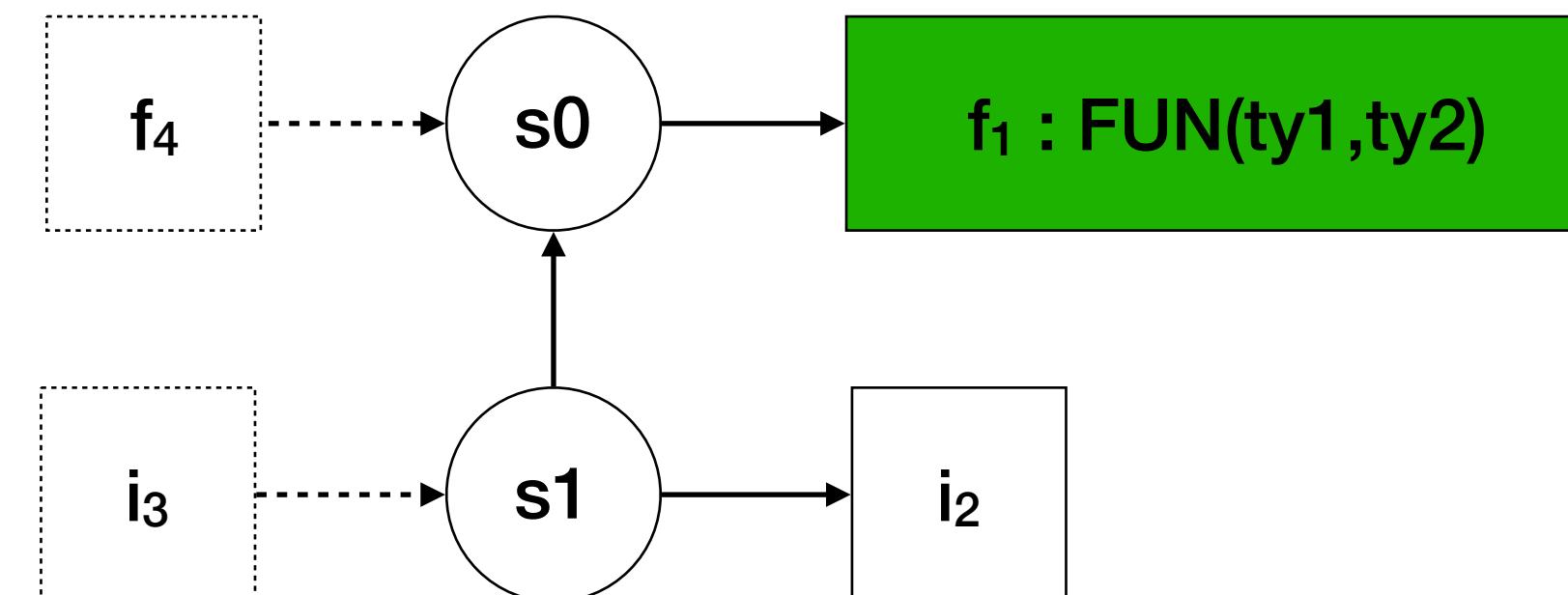
## Constraint semantics

$G, \phi \models t == u$   
*if*  $\phi(t) = \phi(u)$

$G, \phi \models r \text{ in } s |-> d$   
*if*  $\phi(r) = \text{Var}\{x\}$   
*and*  $\phi(d) = \text{Var}\{x\}$   
*and*  $\phi(s) = \#i$   
*and*  $\text{Var}\{x\}$  resolves to  $\text{Var}\{x\}$  from  $\#i$  in  $G$

$G, \phi \models C_1 \wedge C_2$   
*if*  $G, \phi \models C_1$   
*and*  $G, \phi \models C_2$

## Scope graph $G$ (model)



# Different Kinds of Variables

## Program

```

let
  function f1(i2 : int) : int =
    i3 + 1
in
  f4(14)
end
  
```

## Program constraints

```

ty1 == INT()
INT() == INT()
Var{"i"} in #s1 |-> d1
ty2 == INT()
Var{"f"} in #s0 |-> d2
ty3 == FUN(ty4,ty5)
ty4 == INT()
...
  
```

## Unifier $\phi$ (model)

```

 $\phi = \{ \begin{array}{l} ty1 \rightarrow INT(), \\ ty2 \rightarrow INT(), \\ ty3 \rightarrow FUN(INT(),ty5), \\ ty4 \rightarrow INT(), \\ d1 \rightarrow Var\{"i"\}, \\ d2 \rightarrow Var\{"f"\} \end{array} \}$ 
  
```

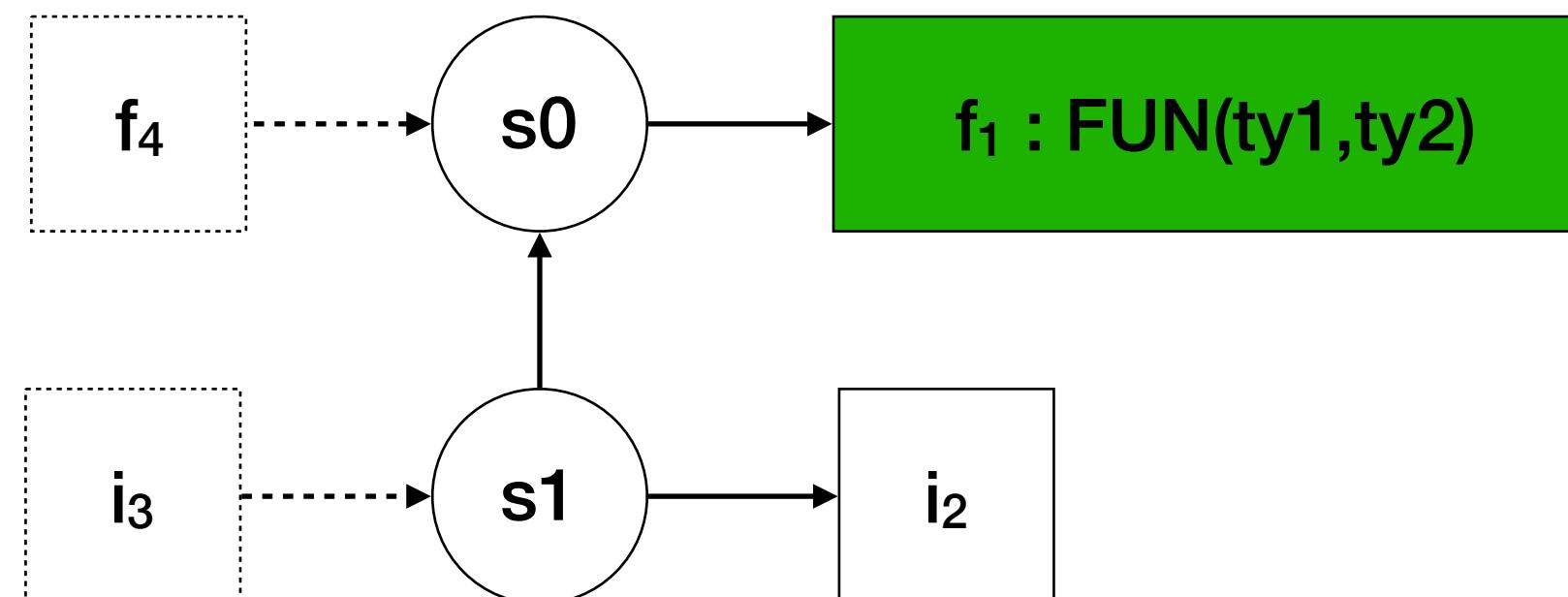
## Constraint semantics

$G, \phi \models t == u$   
*if*  $\phi(t) = \phi(u)$

$G, \phi \models r \text{ in } s |-> d$   
*if*  $\phi(r) = Var\{x\}$   
*and*  $\phi(d) = Var\{x\}$   
*and*  $\phi(s) = \#i$   
*and*  $Var\{x\}$  resolves to  $Var\{x\}$  from  $\#i$  in  $G$

$G, \phi \models C_1 \wedge C_2$   
*if*  $G, \phi \models C_1$   
*and*  $G, \phi \models C_2$

## Scope graph $G$ (model)



# Different Kinds of Variables

Program

```
let
  function f1(i2 : int) : int =
    i3 + 1
in
  f4(14)
end
```

Program constraints

```
ty1 == INT()
INT() == INT()
Var{"i"} in #s1 |-> d1
ty2 == INT()
Var{"f"} in #s0 |-> d2
ty3 == FUN(ty4, ty5)
ty4 == INT()
...
...
```

Unifier  $\phi$  (model)

```
 $\phi = \{ \begin{array}{l} ty1 \rightarrow INT(), \\ ty2 \rightarrow INT(), \\ ty3 \rightarrow FUN(INT(), ty5), \\ ty4 \rightarrow INT(), \\ d1 \rightarrow Var{"i"}, \\ d2 \rightarrow Var{"f"} \end{array} \}$ 
```

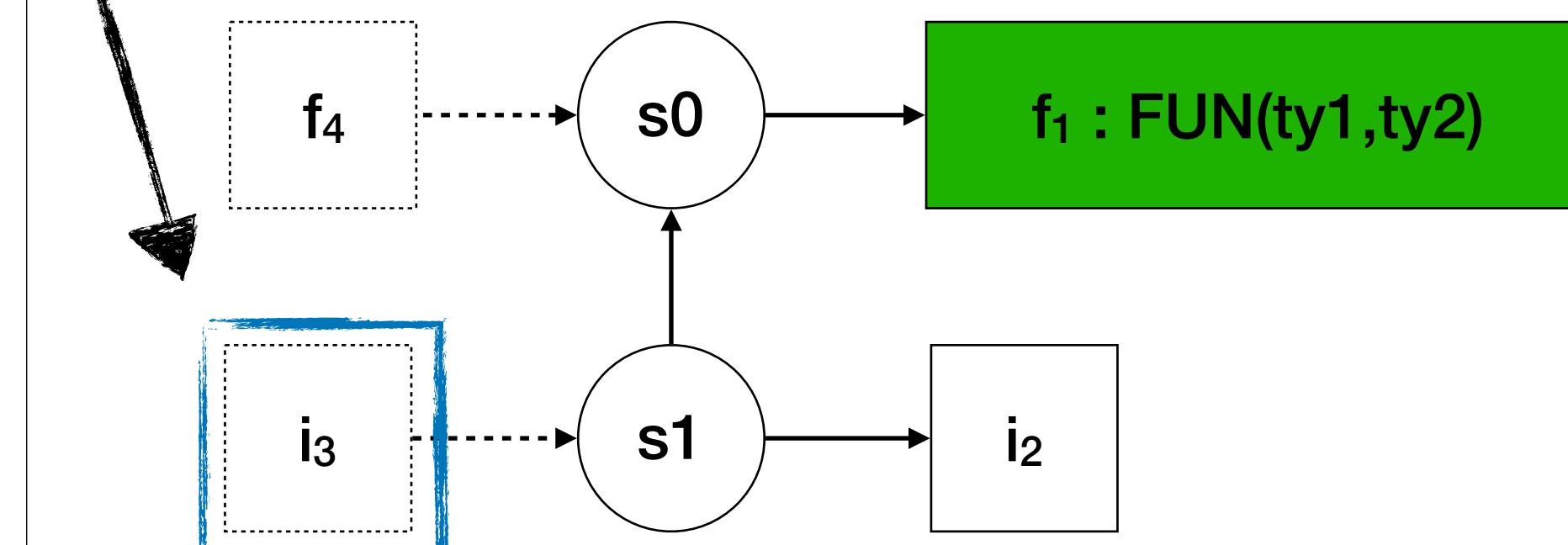
Constraint semantics

$G, \phi \models t == u$   
 $\text{if } \phi(t) = \phi(u)$

$G, \phi \models r \text{ in } s |-> d$   
 $\text{if } \phi(r) = \text{Var}\{x\}$   
 $\text{and } \phi(d) = \text{Var}\{x\}$   
 $\text{and } \phi(s) = \#i$   
 $\text{and } \text{Var}\{x\} \text{ resolves to } \text{Var}\{x\} \text{ from } \#i \text{ in } G$

$G, \phi \models C_1 \wedge C_2$   
 $\text{if } G, \phi \models C_1$   
 $\text{and } G, \phi \models C_2$

Object language  
variables



# Different Kinds of Variables

Program

```
let
  function f1(i2 : int) : int =
    i3 + 1
in
  f4(14)
end
```

Program constraints

```
ty1 == INT()
INT() == INT()
Var{"i"} in #s1 |-> d1
ty2 == INT()
Var{"f"} in #s0 |-> d2
ty3 == FUN(ty4, ty5)
ty4 == INT()
...
```

Unifier  $\phi$  (model)

```
 $\phi = \{ \begin{array}{l} ty1 \rightarrow INT(), \\ ty2 \rightarrow INT(), \\ ty3 \rightarrow FUN(INT(), ty5), \\ ty4 \rightarrow INT(), \\ d1 \rightarrow Var{"i"}, \\ d2 \rightarrow Var{"f"} \end{array} \}$ 
```

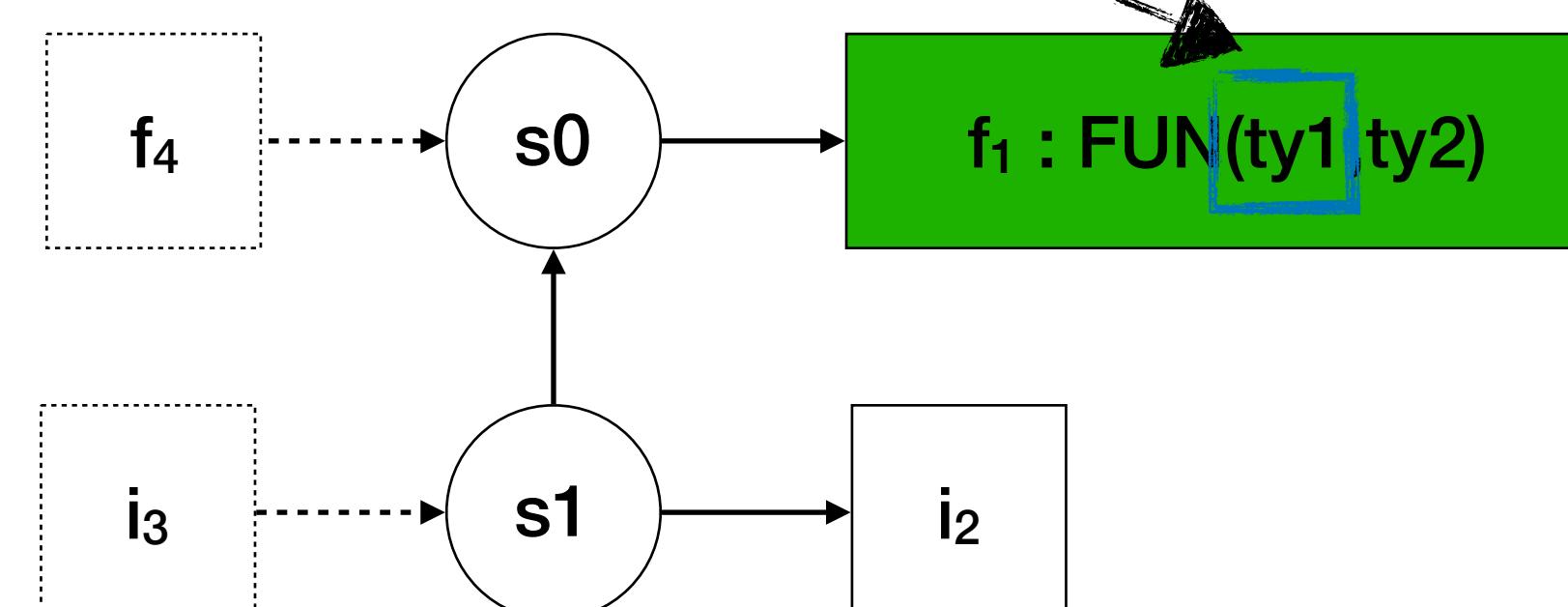
Constraint semantics

$G, \phi \models t == u$   
 $\text{if } \phi(t) = \phi(u)$

$G, \phi \models r \text{ in } s |-> d$   
 $\text{if } \phi(r) = \text{Var}\{x\}$   
 $\text{and } \phi(d) = \text{Var}\{x\}$   
 $\text{and } \phi(s) = \#i$   
 $\text{and } \text{Var}\{x\} \text{ resolves to } \text{Var}\{x\} \text{ from } \#i \text{ in } G$

$G, \phi \models C_1 \wedge C_2$   
 $\text{if } G, \phi \models C_1$   
 $\text{and } G, \phi \models C_2$

Constraint / logic variables



# Different Kinds of Variables

## Program

```
let
  function f1(i2 : int) : int =
    i3 + 1
in
  f4(14)
end
```

## Program constraints

$ty_1 == INT()$   
 $INT() == INT()$   
 $\text{Var}\{"i"\} \text{ in } \#s_1 \mapsto d_1$   
 $ty_2 == INT()$   
 $\text{Var}\{"f"\} \text{ in } \#s_0 \mapsto d_2$   
 $ty_3 == \text{FUN}(ty_4, ty_5)$   
 $ty_4 == INT()$   
...

## Unifier $\phi$ (model)

$$\phi = \{ \begin{array}{l} ty_1 \rightarrow INT(), \\ ty_2 \rightarrow INT(), \\ ty_3 \rightarrow \text{FUN}(INT(), ty_5), \\ ty_4 \rightarrow INT(), \\ d_1 \rightarrow \text{Var}\{"i"\}, \\ d_2 \rightarrow \text{Var}\{"f"\} \end{array} \}$$

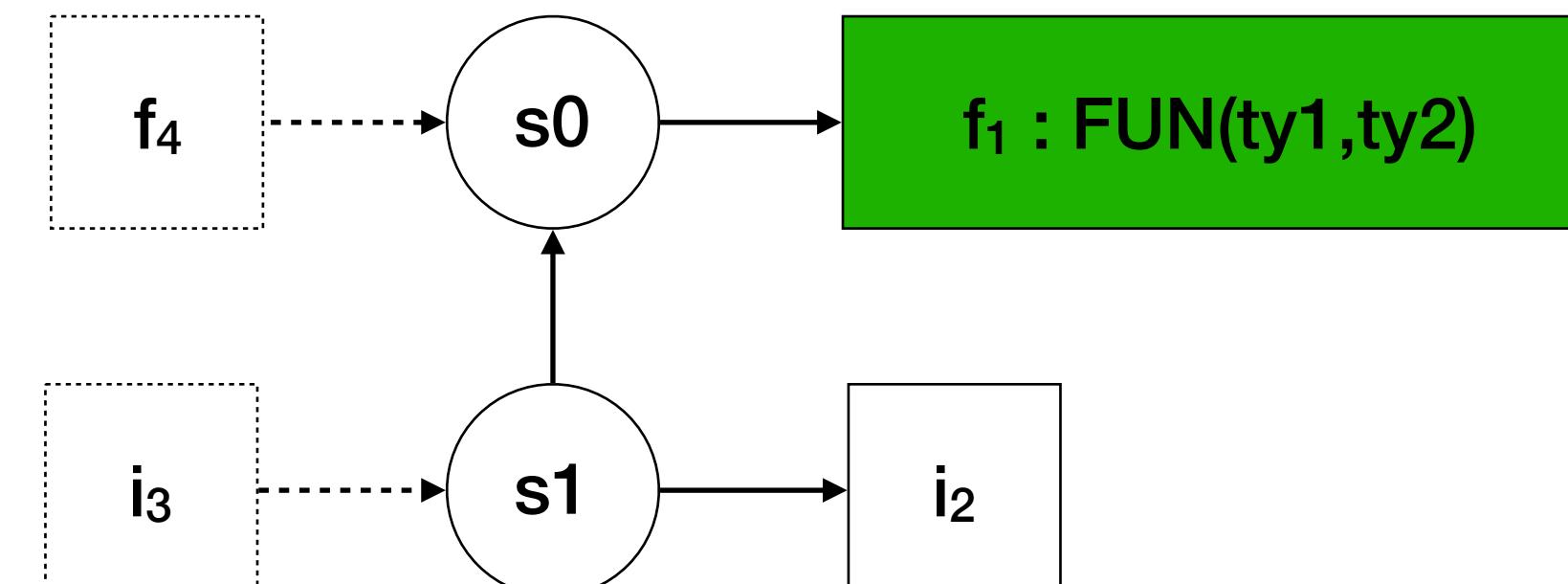
## Constraint semantics

$G, \phi \models t == u$   
if  $\phi(t) = \phi(u)$

$G, \phi \models r \text{ in } s \mapsto d$   
if  $\phi(r) = \text{Var}\{x\}$   
and  $\phi(d) = \text{Var}\{x\}$   
and  $\phi(s) = \#i$   
and  $\text{Var}\{x\}$  resolves to  $\text{Var}\{x\}$  from  $\#i$  in  $G$

$G, \phi \models C_1 \wedge C_2$   
if  $G, \phi \models C_1$   
and  $G, \phi \models C_2$

## Semantics meta-variables



# Using the Semantics

## Program

```

let
  function f1(i2 : int) : int =
    i3 + 1
in
  f4(14)
end

```

## Program constraints

```

ty1 == INT()
INT() == INT()
Var{"i"} in #s1 |-> d1
ty2 == INT()
Var{"f"} in #s0 |-> d2
ty3 == FUN(ty4,ty5)
ty4 == INT()
...

```

## Unifier $\phi$ (model)

```

 $\phi = \{ \begin{array}{l} ty1 \rightarrow INT(), \\ ty2 \rightarrow INT(), \\ ty3 \rightarrow FUN(INT(),ty5), \\ ty4 \rightarrow INT(), \\ d1 \rightarrow Var{"i"}, \\ d2 \rightarrow Var{"f"} \end{array} \}$ 

```

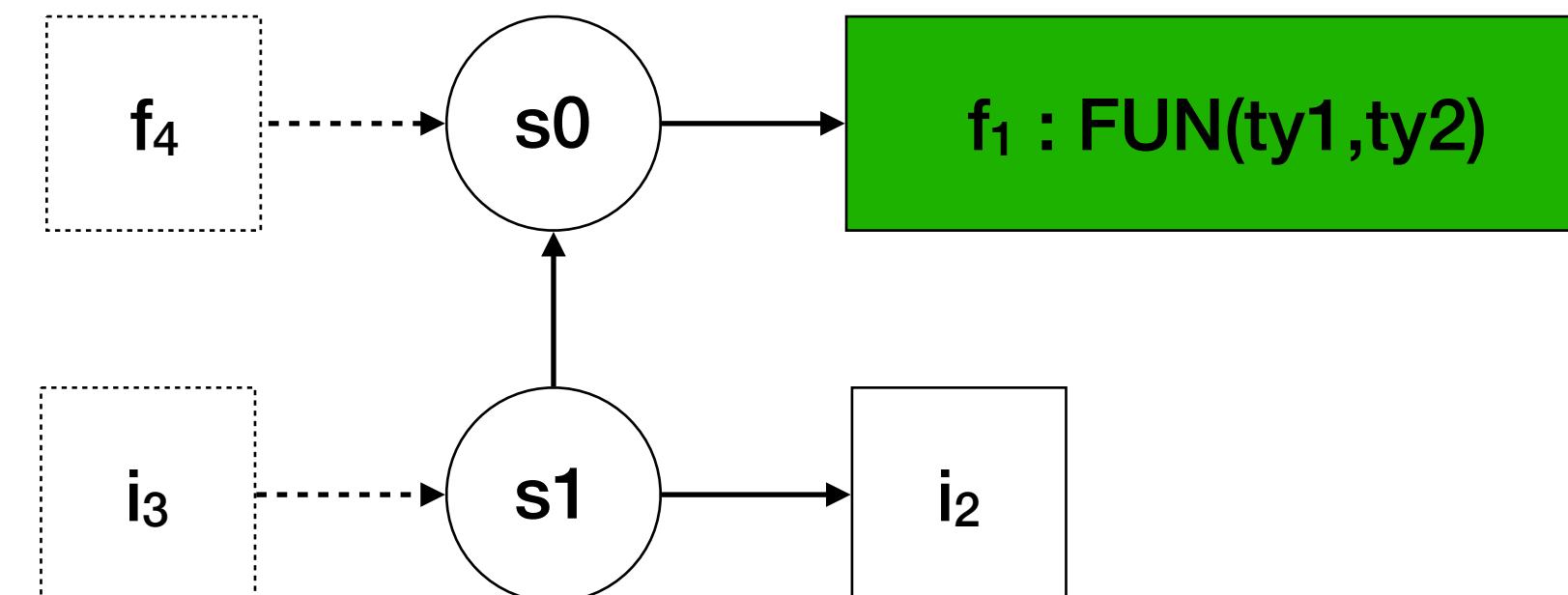
## Constraint semantics

$G, \phi \models t == u$   
*if*  $\phi(t) = \phi(u)$

$G, \phi \models r \text{ in } s |-> d$   
*if*  $\phi(r) = Var\{x\}$   
*and*  $\phi(d) = Var\{x\}$   
*and*  $\phi(s) = \#i$   
*and*  $Var\{x\}$  resolves to  $Var\{x\}$  from  $\#i$  in  $G$

$G, \phi \models C_1 \wedge C_2$   
*if*  $G, \phi \models C_1$   
*and*  $G, \phi \models C_2$

## Scope graph $G$ (model)



# Type Checking

# How to check types?

**What should a type checker do?**

# How to check types?

## What should a type checker do?

- Check that a program is well-typed!

# How to check types?

## What should a type checker do?

- Check that a program is well-typed!
- Resolve names, and check or compute types

# How to check types?

## What should a type checker do?

- Check that a program is well-typed!
- Resolve names, and check or compute types
- Report useful error messages

# How to check types?

## What should a type checker do?

- Check that a program is well-typed!
- Resolve names, and check or compute types
- Report useful error messages
- Provide a representation of name and type information

# How to check types?

## What should a type checker do?

- Check that a program is well-typed!
- Resolve names, and check or compute types
- Report useful error messages
- Provide a representation of name and type information
  - ▶ Type annotated AST

# How to check types?

## What should a type checker do?

- Check that a program is well-typed!
- Resolve names, and check or compute types
- Report useful error messages
- Provide a representation of name and type information
  - ▶ Type annotated AST

## This information is used for

# How to check types?

## What should a type checker do?

- Check that a program is well-typed!
- Resolve names, and check or compute types
- Report useful error messages
- Provide a representation of name and type information
  - ▶ Type annotated AST

## This information is used for

- Next compiler steps (optimization, code generation, ...)

# How to check types?

## What should a type checker do?

- Check that a program is well-typed!
- Resolve names, and check or compute types
- Report useful error messages
- Provide a representation of name and type information
  - ▶ Type annotated AST

## This information is used for

- Next compiler steps (optimization, code generation, ...)
- IDE (error reporting, code completion, refactoring, ...)

# How to check types?

## What should a type checker do?

- Check that a program is well-typed!
- Resolve names, and check or compute types
- Report useful error messages
- Provide a representation of name and type information
  - ▶ Type annotated AST

## This information is used for

- Next compiler steps (optimization, code generation, ...)
- IDE (error reporting, code completion, refactoring, ...)
- Other tools (API documentation, ...)

# How to check types?

## What should a type checker do?

- Check that a program is well-typed!
- Resolve names, and check or compute types
- Report useful error messages
- Provide a representation of name and type information
  - ▶ Type annotated AST

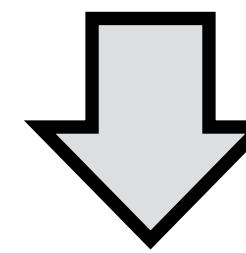
## This information is used for

- Next compiler steps (optimization, code generation, ...)
- IDE (error reporting, code completion, refactoring, ...)
- Other tools (API documentation, ...)

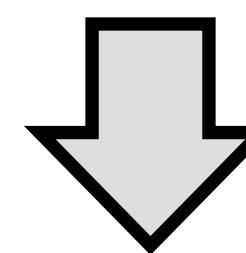
## How are type checkers implemented?

# Computing Type of Expression (recap)

```
function (a : int) = a + 1
```



```
Fun("a", INT(),  
    Plus(Var("a"), Int(1)))
```



```
FUN(INT(), INT())
```

```
typeOfExp(s, Int(_)) = INT().
```

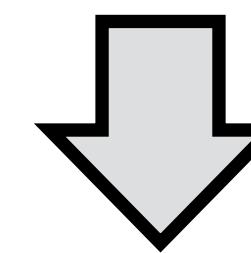
```
typeOfExp(s, Plus(e1, e2)) = INT() :-  
    typeOfExp(s, e1) == INT(),  
    typeOfExp(s, e2) == INT().
```

```
typeOfExp(s, Fun(x, te, e)) = FUN(S, T) :- {s_fun}  
    typeOfTypeExp(s, te) == S,  
    new s_fun, s_fun -P-> s,  
    s_fun -> Var{x} with typeOfDecl S,  
    typeOfExp(s_fun, e) == T.
```

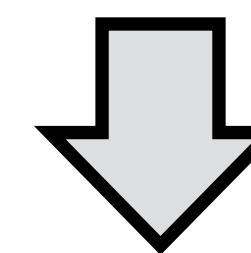
```
typeOfExp(s, Var(x)) = T :-  
    typeOfDecl of Var{x} in s |-> [(_, _, T)].
```

# Computing Type of Expression (recap)

```
function (a : int) = a + 1
```



```
Fun("a", INT(),  
    Plus(Var("a"), Int(1)))
```



```
FUN(INT(), INT())
```

```
typeOfExp(s, Int(_)) = INT().
```

```
typeOfExp(s, Plus(e1, e2)) = INT() :-  
    typeOfExp(s, e1) == INT(),  
    typeOfExp(s, e2) == INT().
```

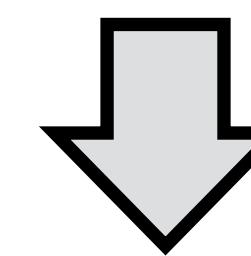
```
typeOfExp(s, Fun(x, te, e)) = FUN(S, T) :- {s_fun}  
    typeOfTypeExp(s, te) == S,  
    new s_fun, s_fun -P-> s,  
    s_fun -> Var{x} with typeOfDecl S,  
    typeOfExp(s_fun, e) == T.
```

```
typeOfExp(s, Var(x)) = T :-  
    typeOfDecl of Var{x} in s |-> [(_, _, T)].
```

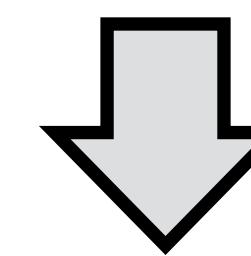
- Can be executed top down, in premise order

# Computing Type of Expression (recap)

```
function (a : int) = a + 1
```



```
Fun("a", INT(),  
    Plus(Var("a"), Int(1)))
```



```
FUN(INT(), INT())
```

```
typeOfExp(s, Int(_)) = INT().
```

```
typeOfExp(s, Plus(e1, e2)) = INT() :-  
    typeOfExp(s, e1) == INT(),  
    typeOfExp(s, e2) == INT().
```

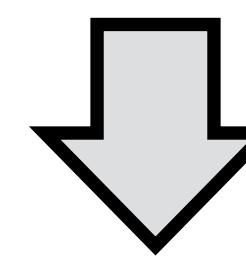
```
typeOfExp(s, Fun(x, te, e)) = FUN(S, T) :- {s_fun}  
    typeOfTypeExp(s, te) == S,  
    new s_fun, s_fun -P-> s,  
    s_fun -> Var{x} with typeOfDecl S,  
    typeOfExp(s_fun, e) == T.
```

```
typeOfExp(s, Var(x)) = T :-  
    typeOfDecl of Var{x} in s |-> [(_, _, T)].
```

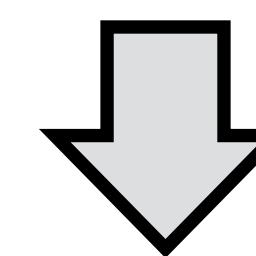
- Can be executed top down, in premise order
- Could be written almost like this in a functional language

# Inferring the Type of a Parameter

```
function (a : int) = a + 1
```



```
Fun("a", INT(),  
    Plus(Var("a"), Int(1)))
```



```
FUN(INT(), INT())
```

```
typeOfExp(s, Int(_)) = INT().
```

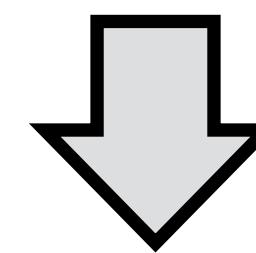
```
typeOfExp(s, Plus(e1, e2)) = INT() :-  
    typeOfExp(s, e1) == INT(),  
    typeOfExp(s, e2) == INT().
```

```
typeOfExp(s, Fun(x, te, e)) = FUN(S, T) :- {s_fun}  
    typeOfTypeExp(s, te) == S,  
    new s_fun, s_fun -P-> s,  
    s_fun -> Var{x} with typeOfDecl S,  
    typeOfExp(s_fun, e) == T.
```

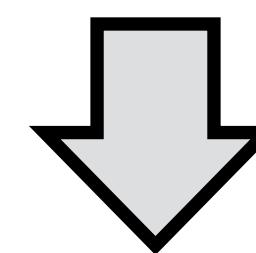
```
typeOfExp(s, Var(x)) = T :-  
    typeOfDecl of Var{x} in s |-> [(_, _, T)].
```

# Inferring the Type of a Parameter

```
function (a $\underline{\hspace{1cm}}$ ) = a + 1
```



```
Fun("a", INT(),  
    Plus(Var("a"), Int(1)))
```



```
FUN(INT(), INT())
```

```
typeOfExp(s, Int(_)) = INT().
```

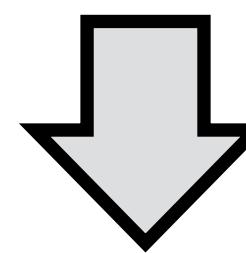
```
typeOfExp(s, Plus(e1, e2)) = INT() :-  
    typeOfExp(s, e1) == INT(),  
    typeOfExp(s, e2) == INT().
```

```
typeOfExp(s, Fun(x, te, e)) = FUN(S, T) :- {s_fun}  
    typeOfTypeExp(s, te) == S,  
    new s_fun, s_fun -P-> s,  
    s_fun -> Var{x} with typeOfDecl S,  
    typeOfExp(s_fun, e) == T.
```

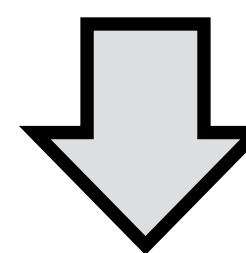
```
typeOfExp(s, Var(x)) = T :-  
    typeOfDecl of Var{x} in s |-> [(_, _, T)].
```

# Inferring the Type of a Parameter

```
function (a[ ]) = a + 1
```



```
Fun("a", [ ]  
      Plus(Var("a")), Int(1)))
```



```
FUN(INT(), INT())
```

```
typeOfExp(s, Int(_)) = INT().
```

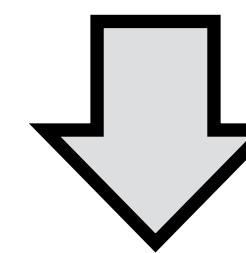
```
typeOfExp(s, Plus(e1, e2)) = INT() :-  
    typeOfExp(s, e1) == INT(),  
    typeOfExp(s, e2) == INT().
```

```
typeOfExp(s, Fun(x, te, e)) = FUN(S, T) :- {s_fun}  
    typeOfTypeExp(s, te) == S,  
    new s_fun, s_fun -P-> s,  
    s_fun -> Var{x} with typeOfDecl S,  
    typeOfExp(s_fun, e) == T.
```

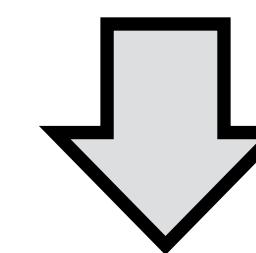
```
typeOfExp(s, Var(x)) = T :-  
    typeOfDecl of Var{x} in s |-> [(_, _, T)].
```

# Inferring the Type of a Parameter

```
function (a $\underline{\quad}$ ) = a + 1
```



```
Fun("a",  $\underline{\quad}$ ,  
    Plus(Var("a")), Int(1)))
```



```
FUN(INT(), INT())
```

```
typeOfExp(s, Int(_)) = INT().
```

```
typeOfExp(s, Plus(e1, e2)) = INT() :-  
    typeOfExp(s, e1) == INT(),  
    typeOfExp(s, e2) == INT().
```

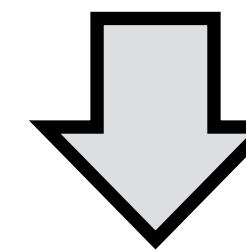
```
typeOfExp(s, Fun(x, te, e)) = FUN(S, T) :- {s_fun}  
    typeOfTypeExp(s, te) == S,  
    new s_fun, s_fun -P-> s,  
    s_fun -> Var{x} with typeOfDecl S,  
    typeOfExp(s_fun, e) == T.
```

```
typeOfExp(s, Var(x)) = T :-  
    typeOfDecl of Var{x} in s I-> [(_, _, T)].
```

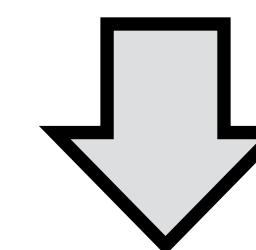
- What are the consequences for our typing rules?

# Inferring the Type of a Parameter

```
function (a[ ]) = a + 1
```



```
Fun("a", [ ]  
      Plus(Var("a")), Int(1)))
```



```
FUN(INT(), INT())
```

```
typeOfExp(s, Int(_)) = INT().
```

```
typeOfExp(s, Plus(e1, e2)) = INT() :-  
    typeOfExp(s, e1) == INT(),  
    typeOfExp(s, e2) == INT().
```

```
typeOfExp(s, Fun(x, [ ] e)) = FUN(S, T) :- {s_fun}
```

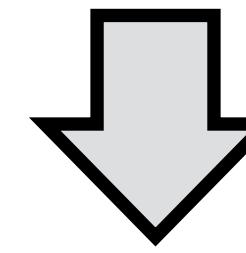
```
[ ]  
new s_fun, s_fun -P-> s,  
s_fun -> Var{x} with typeOfDecl S,  
typeOfExp(s_fun, e) == T.
```

```
typeOfExp(s, Var(x)) = T :-  
    typeOfDecl of Var{x} in s |-> [(_, _, T)].
```

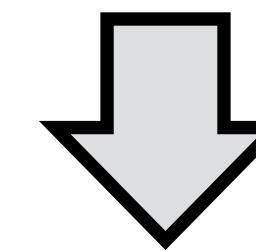
- What are the consequences for our typing rules?

# Inferring the Type of a Parameter

```
function (a[ ]) = a + 1
```



```
Fun("a", [ ]  
      Plus(Var("a")), Int(1)))
```



```
FUN(INT(), INT())
```

Unknown S!

```
typeOfExp(s, Int(_)) = INT().
```

```
typeOfExp(s, Plus(e1, e2)) = INT() :-  
    typeOfExp(s, e1) == INT(),  
    typeOfExp(s, e2) == INT().
```

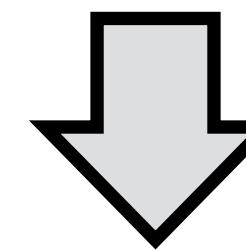
```
typeOfExp(s, Fun(x, [ ] e)) = FUN(S, T) :- {s_fun}  
    new s_fun, s_fun -P-> s,  
    s_fun -> Var{x} with typeOfDecl[S],  
    typeOfExp(s_fun, e) == T.
```

```
typeOfExp(s, Var(x)) = T :-  
    typeOfDecl of Var{x} in s I-> [(_, _, T)].
```

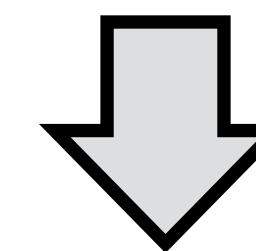
- What are the consequences for our typing rules?

# Inferring the Type of a Parameter

```
function (a[ ]) = a + 1
```



```
Fun("a", [ ]  
      Plus(Var("a")), Int(1)))
```



```
FUN(INT(), INT())
```

Unknown S!

```
typeOfExp(s, Int(_)) = INT().
```

```
typeOfExp(s, Plus(e1, e2)) = INT() :-  
    typeOfExp(s, e1) == INT(),  
    typeOfExp(s, e2) == INT().
```

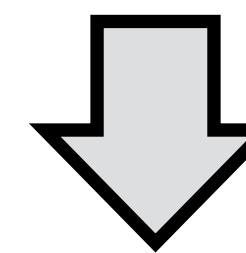
```
typeOfExp(s, Fun(x, [ ] e)) = FUN(S, T) :- {s_fun}  
    new s_fun, s_fun -P-> s,  
    s_fun -> Var{x} with typeOfDecl[S],  
    typeOfExp(s_fun, e) == T.
```

```
typeOfExp(s, Var(x)) = T :-  
    typeOfDecl of Var{x} in s I-> [(_, _, T)].
```

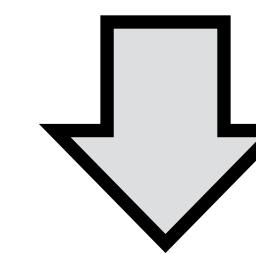
- What are the consequences for our typing rules?
- Types are not known from the start, but learned gradually

# Inferring the Type of a Parameter

```
function (a[ ]) = a + 1
```



```
Fun("a", [ ]  
      Plus(Var("a")), Int(1)))
```



```
FUN(INT(), INT())
```

Unknown S!

```
typeOfExp(s, Int(_)) = INT().
```

```
typeOfExp(s, Plus(e1, e2)) = INT() :-  
    typeOfExp(s, e1) == INT(),  
    typeOfExp(s, e2) == INT().
```

```
typeOfExp(s, Fun(x, [ ] e)) = FUN(S, T) :- {s_fun}  
    new s_fun, s_fun -P-> s,  
    s_fun -> Var{x} with typeOfDecl[S],  
    typeOfExp(s_fun, e) == T.
```

```
typeOfExp(s, Var(x)) = T :-  
    typeOfDecl of Var{x} in s I-> [(_, _, T)].
```

- What are the consequences for our typing rules?
- Types are not known from the start, but learned gradually
- A simple top-down traversal is insufficient

# Checking classes

```
class A {  
    B m() {  
        return new C();  
    }  
}  
  
class B {  
    int i;  
}  
  
class C extends B {  
    int m(A a) {  
        return a.m().i;  
    }  
}
```

# Checking classes

```
class A {  
    B m() {  
        return new C();  
    }  
}  
  
class B {  
    int i;  
}  
  
class C extends B {  
    int m(A a) {  
        return a.m().i;  
    }  
}
```

## How can we type check this program?

- Is there a possible single traversal strategy here?
- Why are the type annotations not enough?
- What strategy could be used?

# Checking classes

```
class A {  
    B m() {  
        return new C();  
    }  
}  
  
class B {  
    int i;  
}  
  
class C extends B {  
    int m(A a) {  
        return a.m().i;  
    }  
}
```

## How can we type check this program?

- Is there a possible single traversal strategy here?
- Why are the type annotations not enough?
- What strategy could be used?

## Two-pass approach

- The first pass builds a class table
- The second pass checks expressions using the class table

# Checking classes

```
class A {  
    B m() {  
        return new C();  
    }  
}  
  
class B {  
    int i;  
}  
  
class C extends B {  
    int m(A a) {  
        return a.m().i;  
    }  
}
```

## How can we type check this program?

- Is there a possible single traversal strategy here?
- Why are the type annotations not enough?
- What strategy could be used?

## Two-pass approach

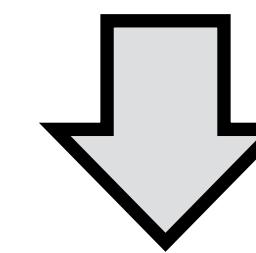
- The first pass builds a class table
- The second pass checks expressions using the class table

## Question

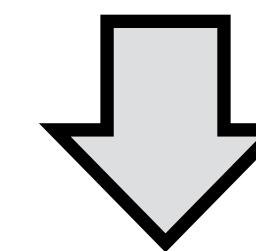
- Does this still work if we introduce nested classes?

# Variables and Constraints

```
function (a █) = a + 1
```



```
Fun("a", █  
    Plus(Var("a")), Int(1)))
```



```
FUN(?S, INT())
```

```
typeOfExp(s, Int(_)) = INT().
```

```
typeOfExp(s, Plus(e1, e2)) = INT() :-  
    typeOfExp(s, e1) == INT(),  
    typeOfExp(s, e2) == INT().
```

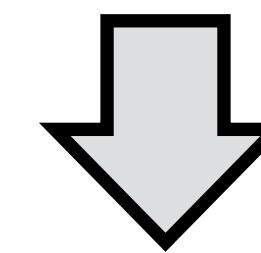
```
typeOfExp(s, Fun(x, █ e)) = FUN(S, T) :- {s_fun}
```

```
new s_fun, s_fun -P-> s,  
s_fun -> Var{x} with typeOfDecl S,  
typeOfExp(s_fun, e) == T.
```

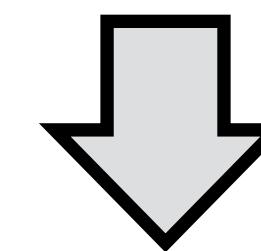
```
typeOfExp(s, Var(x)) = T :-  
    typeOfDecl of Var{x} in s |-> [(_, _, T)].
```

# Variables and Constraints

```
function (a █) = a + 1
```



```
Fun("a", █  
    Plus(Var("a")), Int(1)))
```



```
FUN(?S, INT())
```

```
typeOfExp(s, Int(_)) = INT().
```

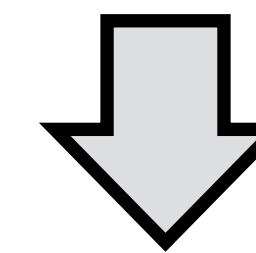
```
typeOfExp(s, Plus(e1, e2)) = INT() :-  
    typeOfExp(s, e1) == INT(),  
    typeOfExp(s, e2) == INT().
```

```
typeOfExp(s, Fun(x, █ e)) = FUN(S, T) :- {s_fun}  
    new s_fun, s_fun -P-> s,  
    s_fun -> Var{x} with typeOfDecl S,  
    typeOfExp(s_fun, e) == T.
```

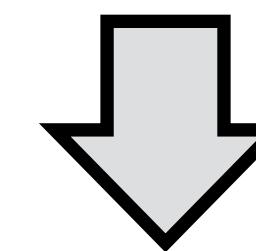
```
typeOfExp(s, Var(x)) = T :-  
    typeOfDecl of Var{x} in s |-> [(_, _, T)].
```

# Variables and Constraints

```
function (a █) = a + 1
```



```
Fun("a", █  
    Plus(Var("a")), Int(1)))
```



```
FUN(█, INT())
```

```
typeOfExp(s, Int(_)) = INT().
```

```
typeOfExp(s, Plus(e1, e2)) = INT() :-  
    typeOfExp(s, e1) == INT(),  
    typeOfExp(s, e2) == INT().
```

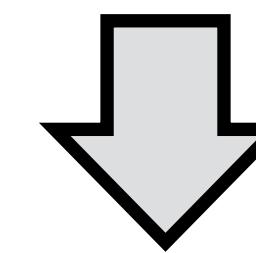
```
typeOfExp(s, Fun(x, █ e)) = FUN(█, T) :- {s_fun}
```

```
new s_fun, s_fun -P-> s,  
s_fun -> Var{x} with typeOfDecl S,  
typeOfExp(s_fun, e) == T.
```

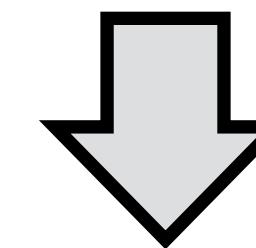
```
typeOfExp(s, Var(x)) = T :-  
    typeOfDecl of Var{x} in s |-> [(_, _, T)].
```

# Variables and Constraints

```
function (a █) = a + 1
```



```
Fun("a", █  
    Plus(Var("a")), Int(1)))
```



```
FUN(█, INT()) + ?S == INT()
```

```
typeOfExp(s, Int(_)) = INT().
```

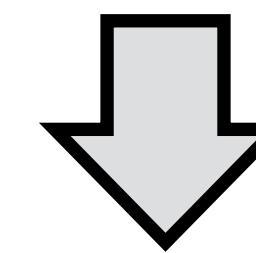
```
typeOfExp(s, Plus(e1, e2)) = INT() :-  
    typeOfExp(s, e1) == INT(),  
    typeOfExp(s, e2) == INT().
```

```
typeOfExp(s, Fun(x, █ e)) = FUN(█, T) :- {s_fun}  
    new s_fun, s_fun -P-> s,  
    s_fun -> Var{x} with typeOfDecl S,  
    typeOfExp(s_fun, e) == T.
```

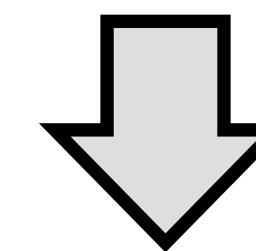
```
typeOfExp(s, Var(x)) = T :-  
    typeOfDecl of Var{x} in s I-> [(_, _, T)].
```

# Variables and Constraints

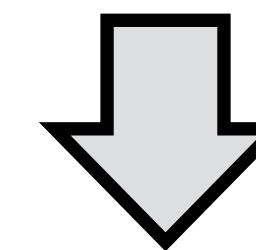
```
function (a[ ]) = a + 1
```



```
Fun("a", [ ],  
    Plus(Var("a")), Int(1)))
```



```
FUN(?S, INT()) + ?S == INT()
```



```
?S := INT()
```

```
typeOfExp(s, Int(_)) = INT().
```

```
typeOfExp(s, Plus(e1, e2)) = INT() :-  
    typeOfExp(s, e1) == INT(),  
    typeOfExp(s, e2) == INT().
```

```
typeOfExp(s, Fun(x, [ ] e)) = FUN(S, T) :- {s_fun}  
    new s_fun, s_fun -P-> s,  
    s_fun -> Var{x} with typeOfDecl S,  
    typeOfExp(s_fun, e) == T.
```

```
typeOfExp(s, Var(x)) = T :-  
    typeOfDecl of Var{x} in s I-> [(_, _, T)].
```

# How to check types?

What are challenges when implementing a type checker?

# How to check types?

## What are challenges when implementing a type checker?

- Collecting necessary binding information before using it

# How to check types?

## What are challenges when implementing a type checker?

- Collecting necessary binding information before using it
- Gradually learning type information

# How to check types?

## What are challenges when implementing a type checker?

- Collecting necessary binding information before using it
- Gradually learning type information

## What are the consequences of these challenges?

# How to check types?

## What are challenges when implementing a type checker?

- Collecting necessary binding information before using it
- Gradually learning type information

## What are the consequences of these challenges?

- The order of computation needs to be more flexible than the AST traversal

# How to check types?

## What are challenges when implementing a type checker?

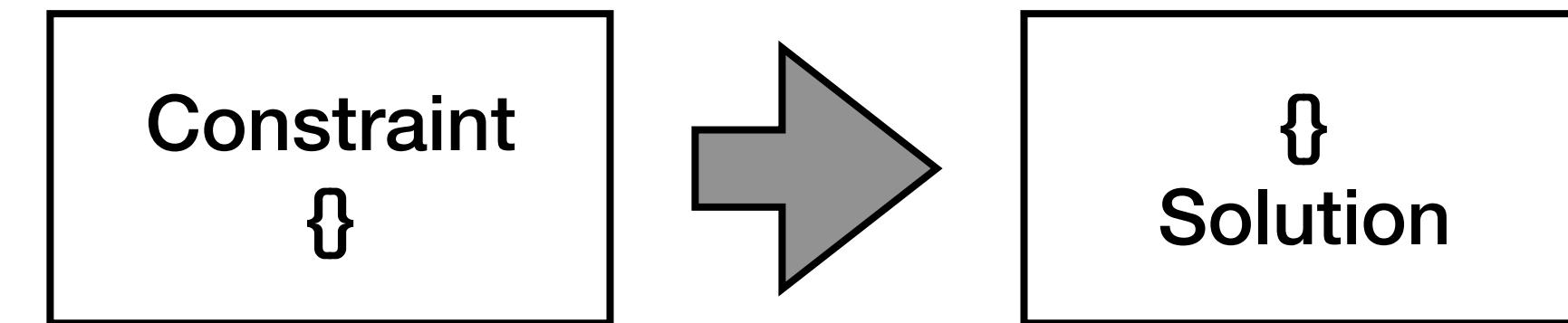
- Collecting necessary binding information before using it
- Gradually learning type information

## What are the consequences of these challenges?

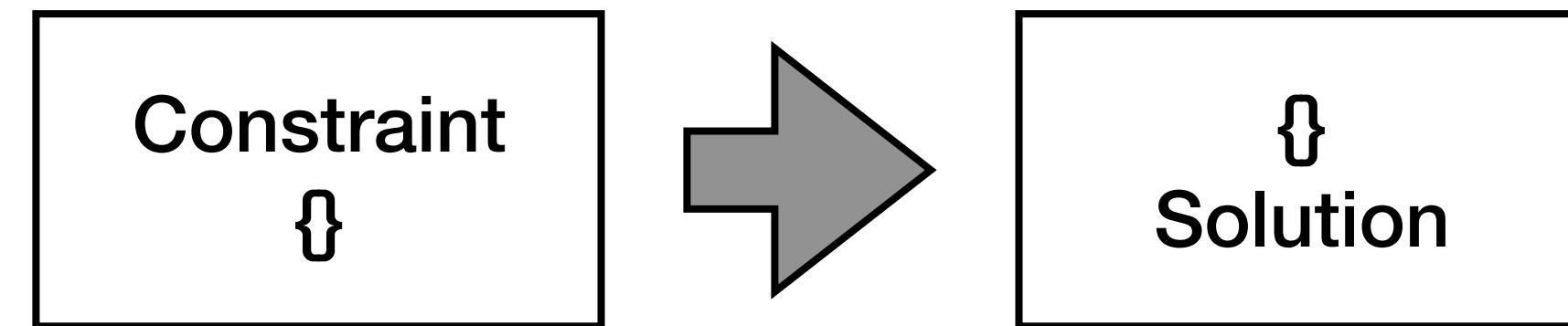
- The order of computation needs to be more flexible than the AST traversal
- Support explicit logical variables during solving

# Solving Constraints

# Solving by Rewriting

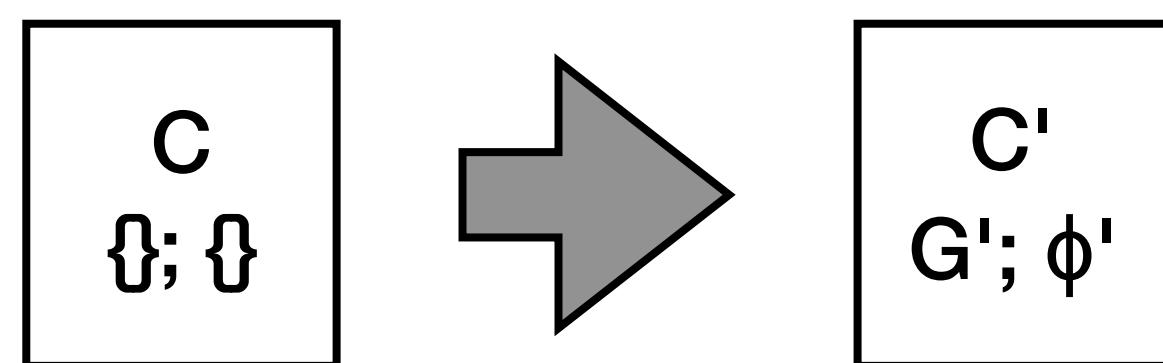
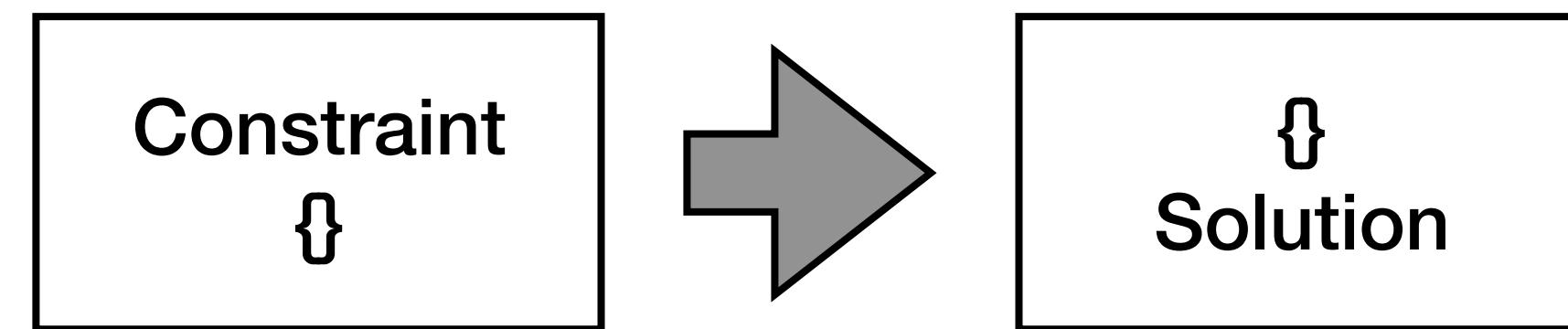


# Solving by Rewriting

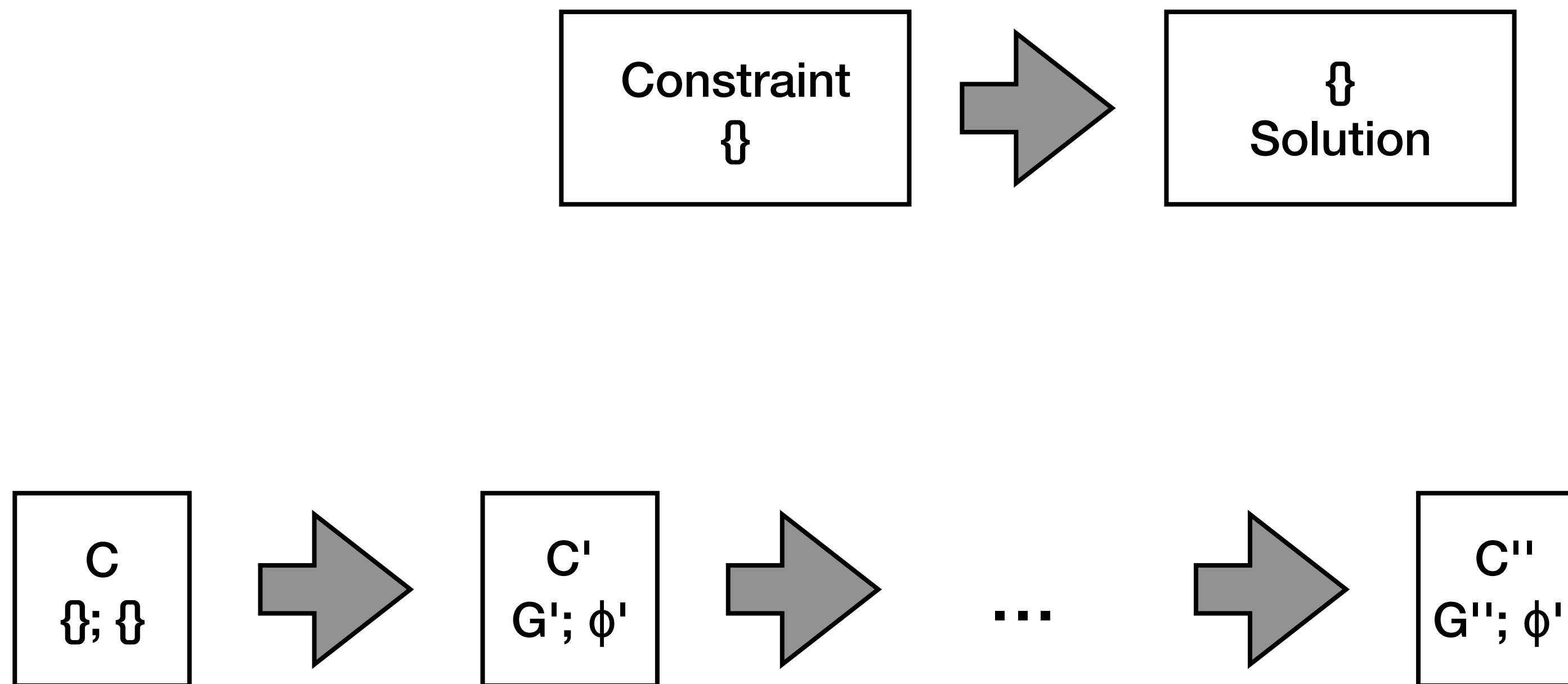


C  
{}; {}

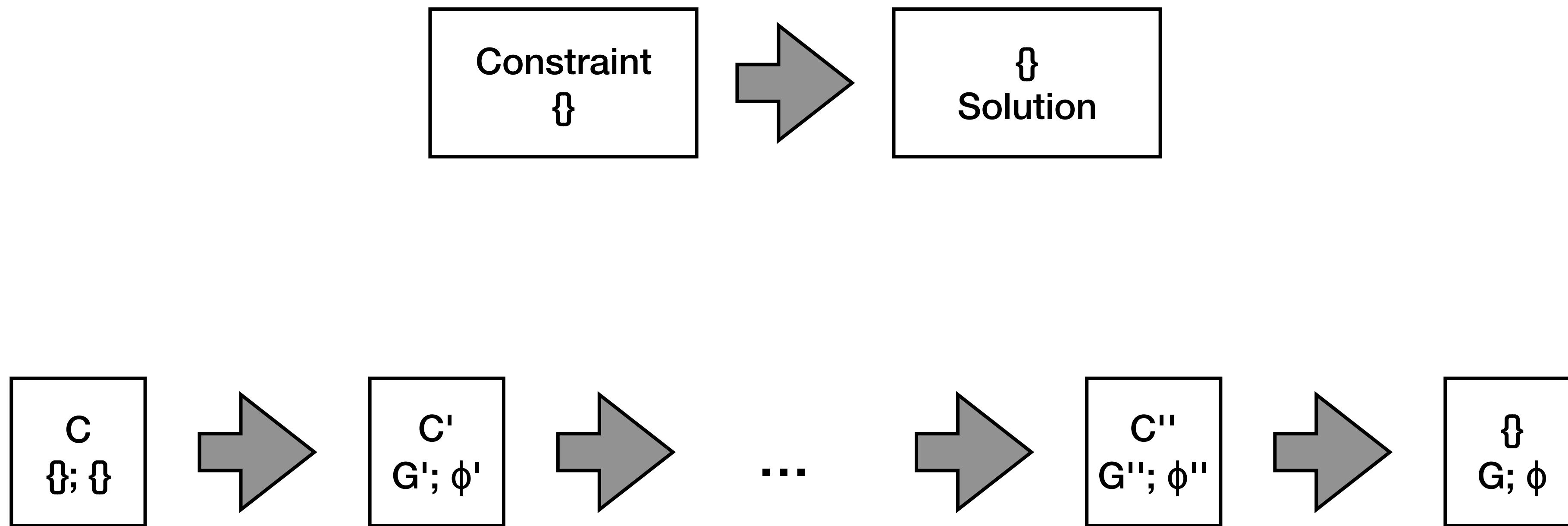
# Solving by Rewriting



# Solving by Rewriting



# Solving by Rewriting



# Solving by Rewriting

$$\boxed{<\mathcal{C}; \ G, \ \phi> \longrightarrow <\mathcal{C}; \ G, \ \phi>}$$

# Solving by Rewriting

$$\boxed{< C; \ G, \ \phi > \longrightarrow < C; \ G, \ \phi >}$$

$< t == u, \ C; \ G, \ \phi > \longrightarrow < C; \ G, \ \phi' >$  where  $\text{unify}(\phi, t, u) = \phi'$

# Solving by Rewriting

Non-deterministic  
constraint selection

$$\langle C; G, \phi \rangle \rightarrow \langle C; G, \phi \rangle$$

$$\langle t == u, C; G, \phi \rangle \rightarrow \langle C; G, \phi' \rangle \text{ where } \text{unify}(\phi, t, u) = \phi'$$

# Solving by Rewriting

$$\boxed{< C; \ G, \ \phi > \longrightarrow < C; \ G, \ \phi >}$$

$< t == u, \ C; \ G, \ \phi > \longrightarrow < C; \ G, \ \phi' >$  where  $\text{unify}(\phi, t, u) = \phi'$

$< s1 \ -L\rightarrow \ s2, \ C; \ G, \ \phi > \longrightarrow < C; \ G', \ \phi >$  where  $\phi(s1) = \#i, \ \phi(s2) = \#j,$   
 $G + \{\#i \ -L\rightarrow \ \#j\} = G'$

# Solving by Rewriting

$$\boxed{< C; \ G, \ \phi > \longrightarrow < C; \ G, \ \phi >}$$

$< t == u, \ C; \ G, \ \phi > \longrightarrow < C; \ G, \ \phi' >$  where  $\text{unify}(\phi, t, u) = \phi'$

$< s1 \ -L\rightarrow \ s2, \ C; \ G, \ \phi > \longrightarrow < C; \ G', \ \phi >$  where  $\phi(s1) = \#i, \ \phi(s2) = \#j,$   
 $G + \{\#i \ -L\rightarrow \ \#j\} = G'$

$< r \ \text{in} \ s \ |-> \ t, \ C; \ G, \ \phi > \longrightarrow < t == d; \ G, \ \phi >$  where  $\phi(r) = Ns\{x\}, \ \phi(s) = \#i,$   
 $\text{resolve}(G, \ \#i, \ Ns\{x\}) = d$

# Solving by Rewriting

$$\boxed{< C; \ G, \ \phi > \longrightarrow < C; \ G, \ \phi >}$$

$< t == u, \ C; \ G, \ \phi > \longrightarrow < C; \ G, \ \phi' >$  where  $\text{unify}(\phi, t, u) = \phi'$

$< s1 \ -L\rightarrow \ s2, \ C; \ G, \ \phi > \longrightarrow < C; \ G', \ \phi >$  where  $\phi(s1) = \#i, \ \phi(s2) = \#j,$   
 $G + \{\#i \ -L\rightarrow \ \#j\} = G'$

$< r \ \text{in} \ s \ |-> \ t, \ C; \ G, \ \phi > \longrightarrow < t == d; \ G, \ \phi >$  where  $\phi(r) = Ns\{x\}, \ \phi(s) = \#i,$   
 $\text{resolve}(G, \ \#i, \ Ns\{x\}) = d$

Scope graph and  
name resolution  
algorithm don't have  
to know about logical  
variables

# Solving by Rewriting

$$\boxed{< \mathcal{C}; \ G, \ \phi > \longrightarrow < \mathcal{C}; \ G, \ \phi >}$$

$< t == u, \ \mathcal{C}; \ G, \ \phi > \longrightarrow < \mathcal{C}; \ G, \ \phi' >$  where  $\text{unify}(\phi, t, u) = \phi'$

$< s_1 -\mathcal{L}-> s_2, \ \mathcal{C}; \ G, \ \phi > \longrightarrow < \mathcal{C}; \ G', \ \phi >$  where  $\phi(s_1) = \#i, \ \phi(s_2) = \#j,$   
 $G + \{\#i -\mathcal{L}-> \#j\} = G'$

$< r \text{ in } s \ |-> t, \ \mathcal{C}; \ G, \ \phi > \longrightarrow < t == d; \ G, \ \phi >$  where  $\phi(r) = Ns\{x\}, \ \phi(s) = \#i,$   
 $\text{resolve}(G, \ \#i, \ Ns\{x\}) = d$

```
def solve( $\mathcal{C}$ ):  
    if  $< \mathcal{C}; \ \{\}, \ \{\} > \longrightarrow^* < \{\}; \ G, \ \phi >:$   
        return  $< G, \ \phi >$   
    else:  
        fail
```

# Solving by Rewriting

**Solver = rewrite system**

# Solving by Rewriting

## **Solver = rewrite system**

- Rewrite a constraints set + solution

# Solving by Rewriting

## **Solver = rewrite system**

- Rewrite a constraints set + solution
- Simplifying and eliminating constraints

# Solving by Rewriting

## Solver = rewrite system

- Rewrite a constraints set + solution
- Simplifying and eliminating constraints
  - ▶ Constraint selecting is non-deterministic

# Solving by Rewriting

## Solver = rewrite system

- Rewrite a constraints set + solution
- Simplifying and eliminating constraints
  - ▶ Constraint selecting is non-deterministic
  - ▶ Resolution order is controlled by side conditions on rewrite rules

# Solving by Rewriting

## Solver = rewrite system

- Rewrite a constraints set + solution
- Simplifying and eliminating constraints
  - ▶ Constraint selecting is non-deterministic
  - ▶ Resolution order is controlled by side conditions on rewrite rules
- Rely on (other) solvers and algorithms for base cases

# Solving by Rewriting

## Solver = rewrite system

- Rewrite a constraints set + solution
- Simplifying and eliminating constraints
  - ▶ Constraint selecting is non-deterministic
  - ▶ Resolution order is controlled by side conditions on rewrite rules
- Rely on (other) solvers and algorithms for base cases
  - ▶ Unification for term equality

# Solving by Rewriting

## Solver = rewrite system

- Rewrite a constraints set + solution
- Simplifying and eliminating constraints
  - ▶ Constraint selecting is non-deterministic
  - ▶ Resolution order is controlled by side conditions on rewrite rules
- Rely on (other) solvers and algorithms for base cases
  - ▶ Unification for term equality
  - ▶ Scope graph resolution

# Solving by Rewriting

## Solver = rewrite system

- Rewrite a constraints set + solution
- Simplifying and eliminating constraints
  - ▶ Constraint selecting is non-deterministic
  - ▶ Resolution order is controlled by side conditions on rewrite rules
- Rely on (other) solvers and algorithms for base cases
  - ▶ Unification for term equality
  - ▶ Scope graph resolution
- The solution is final if all constraints are eliminated

# Solving by Rewriting

## Solver = rewrite system

- Rewrite a constraints set + solution
- Simplifying and eliminating constraints
  - ▶ Constraint selecting is non-deterministic
  - ▶ Resolution order is controlled by side conditions on rewrite rules
- Rely on (other) solvers and algorithms for base cases
  - ▶ Unification for term equality
  - ▶ Scope graph resolution
- The solution is final if all constraints are eliminated

**Does the order matter for the outcome?**

# Solving by Rewriting

## **Solver = rewrite system**

- Rewrite a constraints set + solution
- Simplifying and eliminating constraints
  - ▶ Constraint selecting is non-deterministic
  - ▶ Resolution order is controlled by side conditions on rewrite rules
- Rely on (other) solvers and algorithms for base cases
  - ▶ Unification for term equality
  - ▶ Scope graph resolution
- The solution is final if all constraints are eliminated

## **Does the order matter for the outcome?**

- Confluence: the output is the same for any solving order

# Solving by Rewriting

## **Solver = rewrite system**

- Rewrite a constraints set + solution
- Simplifying and eliminating constraints
  - ▶ Constraint selecting is non-deterministic
  - ▶ Resolution order is controlled by side conditions on rewrite rules
- Rely on (other) solvers and algorithms for base cases
  - ▶ Unification for term equality
  - ▶ Scope graph resolution
- The solution is final if all constraints are eliminated

## **Does the order matter for the outcome?**

- Confluence: the output is the same for any solving order
- Partly true for Statix

# Solving by Rewriting

## Solver = rewrite system

- Rewrite a constraints set + solution
- Simplifying and eliminating constraints
  - ▶ Constraint selecting is non-deterministic
  - ▶ Resolution order is controlled by side conditions on rewrite rules
- Rely on (other) solvers and algorithms for base cases
  - ▶ Unification for term equality
  - ▶ Scope graph resolution
- The solution is final if all constraints are eliminated

## Does the order matter for the outcome?

- Confluence: the output is the same for any solving order
- Partly true for Statix
  - ▶ Up to variable and scope names

# Solving by Rewriting

## Solver = rewrite system

- Rewrite a constraints set + solution
- Simplifying and eliminating constraints
  - ▶ Constraint selecting is non-deterministic
  - ▶ Resolution order is controlled by side conditions on rewrite rules
- Rely on (other) solvers and algorithms for base cases
  - ▶ Unification for term equality
  - ▶ Scope graph resolution
- The solution is final if all constraints are eliminated

## Does the order matter for the outcome?

- Confluence: the output is the same for any solving order
- Partly true for Statix
  - ▶ Up to variable and scope names
  - ▶ Only if all constraints are reduced

# Semantics vs Algorithm

**What is the difference?**

# Semantics vs Algorithm

## What is the difference?

- Algorithm computes a solution (= model)

# Semantics vs Algorithm

## What is the difference?

- Algorithm computes a solution (= model)
- Semantics describes when a constraint is satisfied by a model

# Semantics vs Algorithm

## What is the difference?

- Algorithm computes a solution (= model)
- Semantics describes when a constraint is satisfied by a model

## How are these related?

# Semantics vs Algorithm

## What is the difference?

- Algorithm computes a solution (= model)
- Semantics describes when a constraint is satisfied by a model

## How are these related?

- Soundness

# Semantics vs Algorithm

## What is the difference?

- Algorithm computes a solution (= model)
- Semantics describes when a constraint is satisfied by a model

## How are these related?

- Soundness
  - If the solver returns  $\langle G, \phi \rangle$ , then  $G, \phi \models C$

# Semantics vs Algorithm

## What is the difference?

- Algorithm computes a solution (= model)
- Semantics describes when a constraint is satisfied by a model

## How are these related?

- Soundness
  - If the solver returns  $\langle G, \phi \rangle$ , then  $G, \phi \models C$
- Completeness:

# Semantics vs Algorithm

## What is the difference?

- Algorithm computes a solution (= model)
- Semantics describes when a constraint is satisfied by a model

## How are these related?

- Soundness
  - If the solver returns  $\langle G, \phi \rangle$ , then  $G, \phi \models C$
- Completeness:
  - If a  $G$  and  $\phi$  exists such that  $G, \phi \models C$ , then the solver returns it

# Semantics vs Algorithm

## What is the difference?

- Algorithm computes a solution (= model)
- Semantics describes when a constraint is satisfied by a model

## How are these related?

- Soundness
  - If the solver returns  $\langle G, \phi \rangle$ , then  $G, \phi \models C$
- Completeness:
  - If a  $G$  and  $\phi$  exists such that  $G, \phi \models C$ , then the solver returns it
  - If no such  $G$  or  $\phi$  exists, the solver fails

# Semantics vs Algorithm

## What is the difference?

- Algorithm computes a solution (= model)
- Semantics describes when a constraint is satisfied by a model

## How are these related?

- Soundness
  - If the solver returns  $\langle G, \phi \rangle$ , then  $G, \phi \models C$
- Completeness:
  - If a  $G$  and  $\phi$  exists such that  $G, \phi \models C$ , then the solver returns it
  - If no such  $G$  or  $\phi$  exists, the solver fails
- Principality

# Semantics vs Algorithm

## What is the difference?

- Algorithm computes a solution (= model)
- Semantics describes when a constraint is satisfied by a model

## How are these related?

- Soundness
  - If the solver returns  $\langle G, \phi \rangle$ , then  $G, \phi \models C$
- Completeness:
  - If a  $G$  and  $\phi$  exists such that  $G, \phi \models C$ , then the solver returns it
  - If no such  $G$  or  $\phi$  exists, the solver fails
- Principality
  - The solver finds the most general  $\phi$

# **Term Equality & Unification**

# Syntactic Terms

terms	t, u
functions	f, g, h

## Generic Terms

# Syntactic Terms

terms	t, u
functions	f, g, h

## Generic Terms

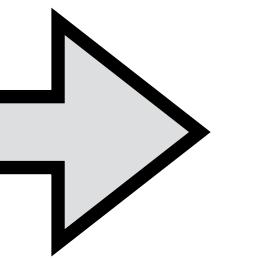
INT()
FUN(INT(), INT())

# Syntactic Terms

terms	t, u
functions	f, g, h

## Generic Terms

```
INT()
FUNC(INT(), INT())
```

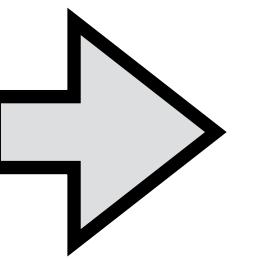


# Syntactic Terms

terms	t, u
functions	f, g, h

## Generic Terms

INT○  
FUN(INT○, INT○)

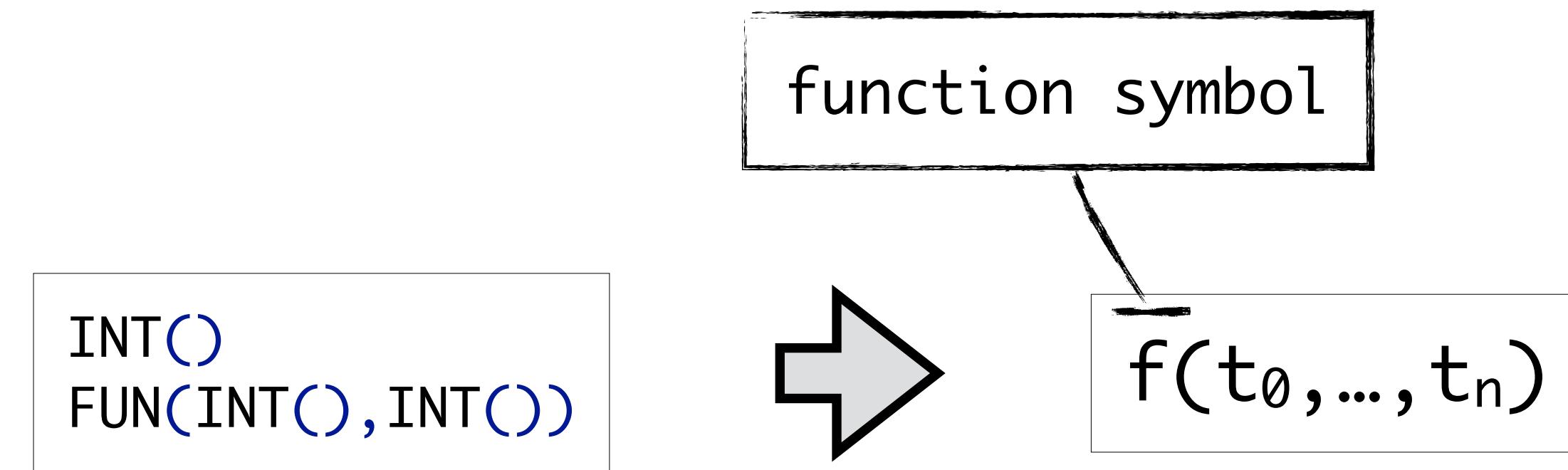


$f(t_0, \dots, t_n)$

# Syntactic Terms

terms	t, u
functions	f, g, h

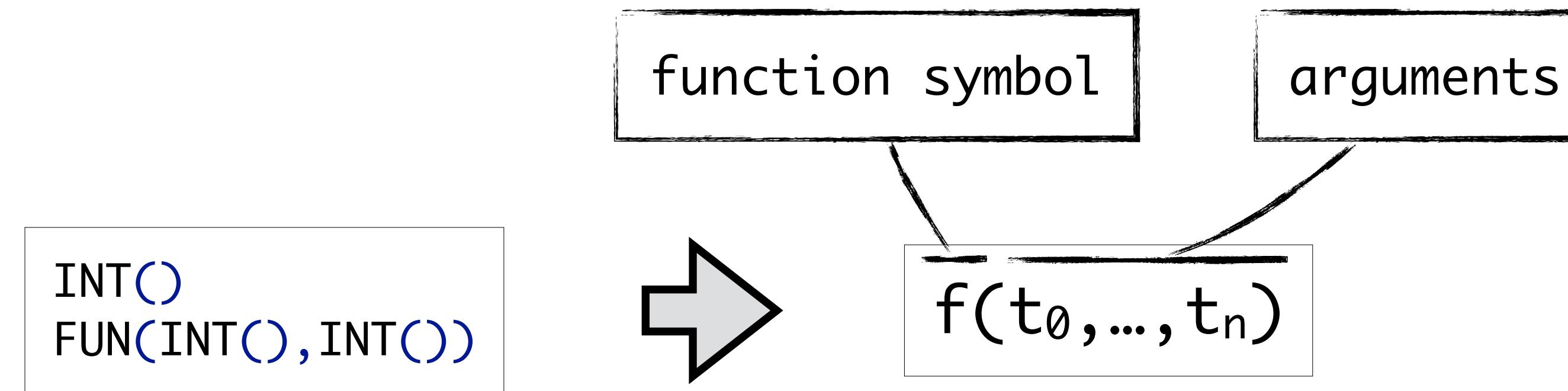
## Generic Terms



# Syntactic Terms

terms  $t, u$   
functions  $f, g, h$

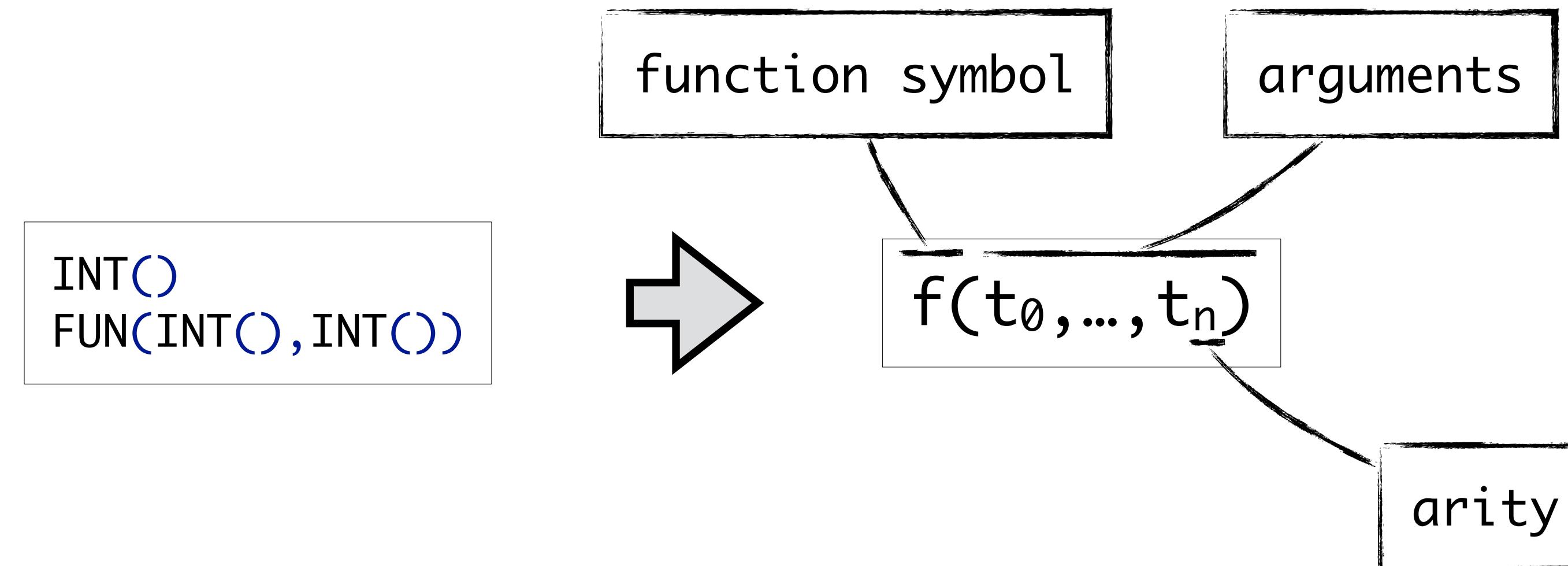
## Generic Terms



# Syntactic Terms

terms  $t, u$   
functions  $f, g, h$

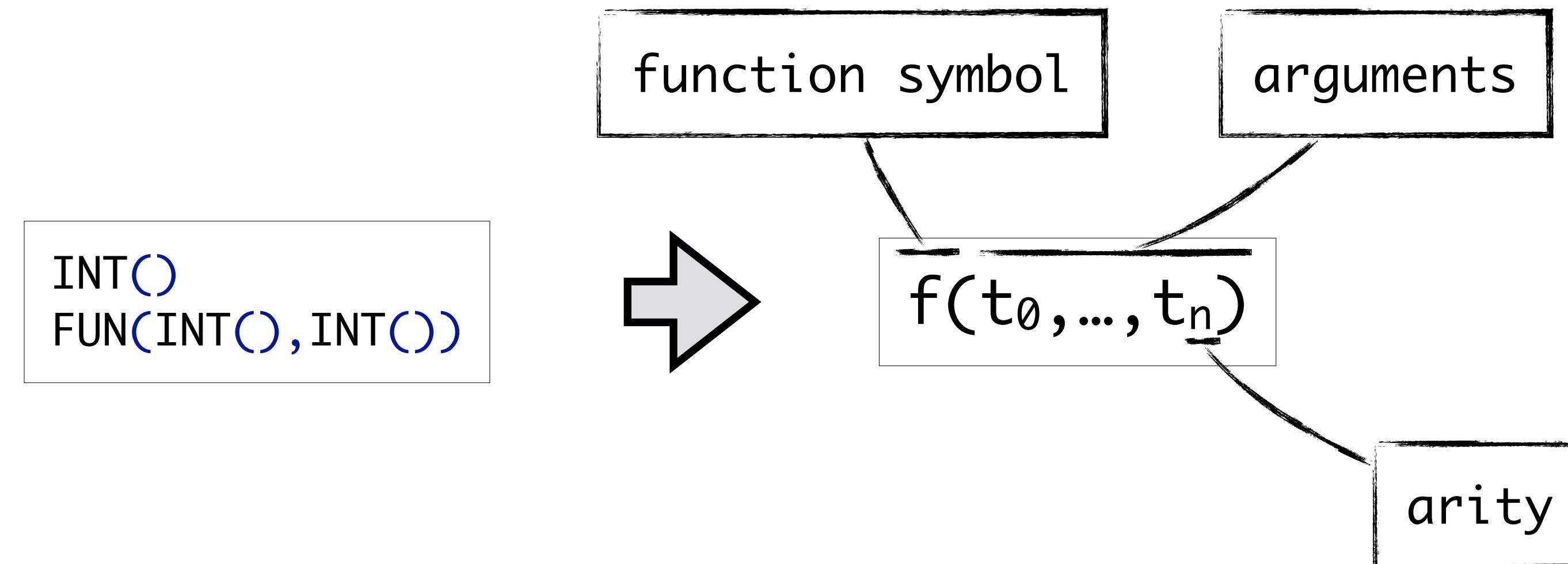
## Generic Terms



# Syntactic Terms

terms      t, u  
functions    f, g, h

## Generic Terms



## Syntactic Equality

$f(t_0, \dots, t_n) == g(u_0, \dots, u_m)$  if  
-  $f = g$ , and  $n = m$   
-  $t_i == u_i$  for every  $i$

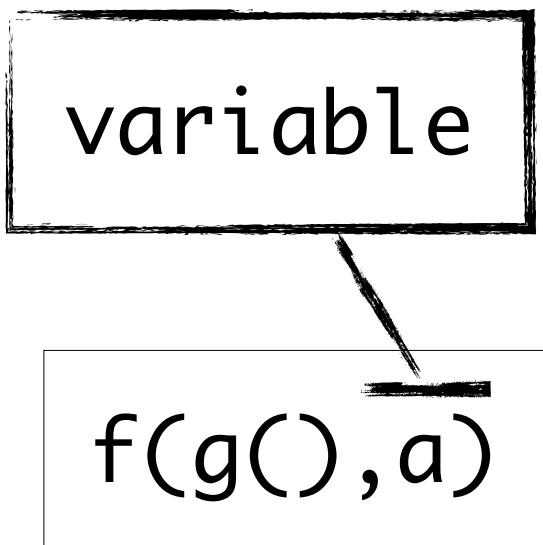
# Variables and Substitution

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

f(g(), a)

# Variables and Substitution

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$



# Variables and Substitution

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

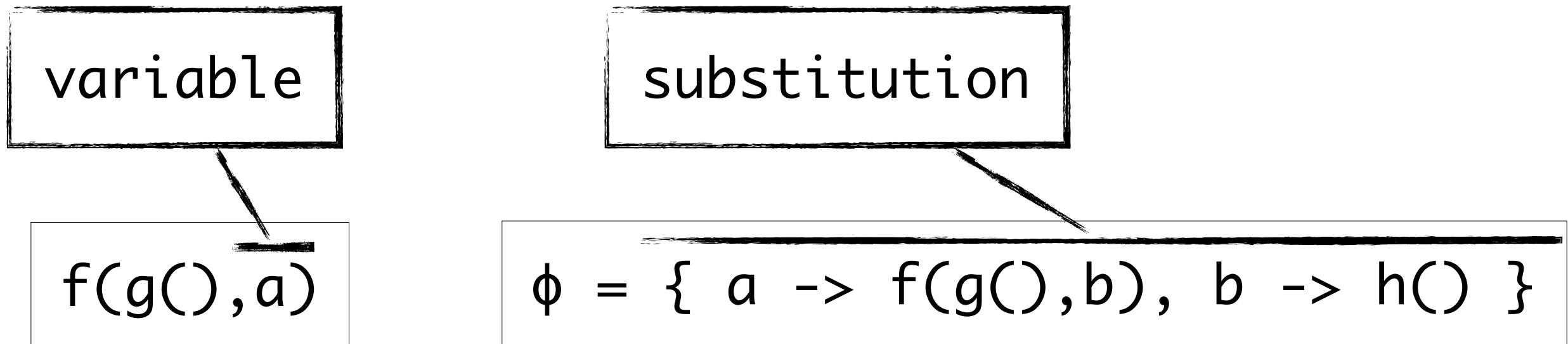
variable

$f(g(), \underline{a})$

$$\phi = \{ a \rightarrow f(g(), b), b \rightarrow h() \}$$

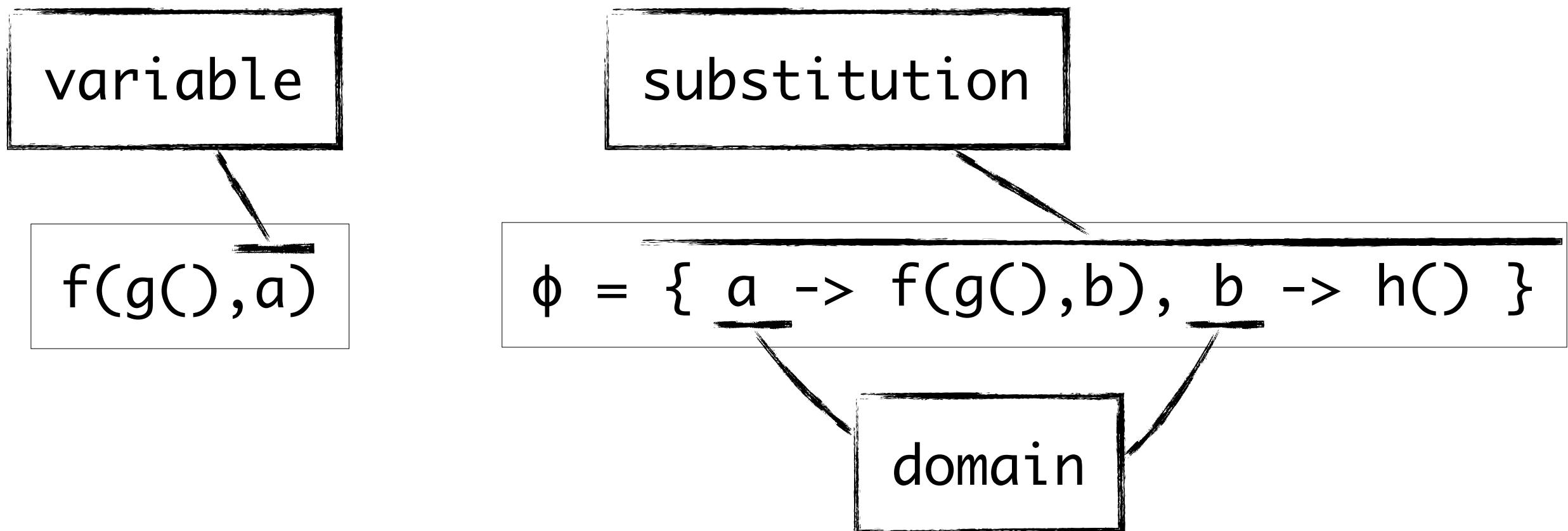
# Variables and Substitution

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$



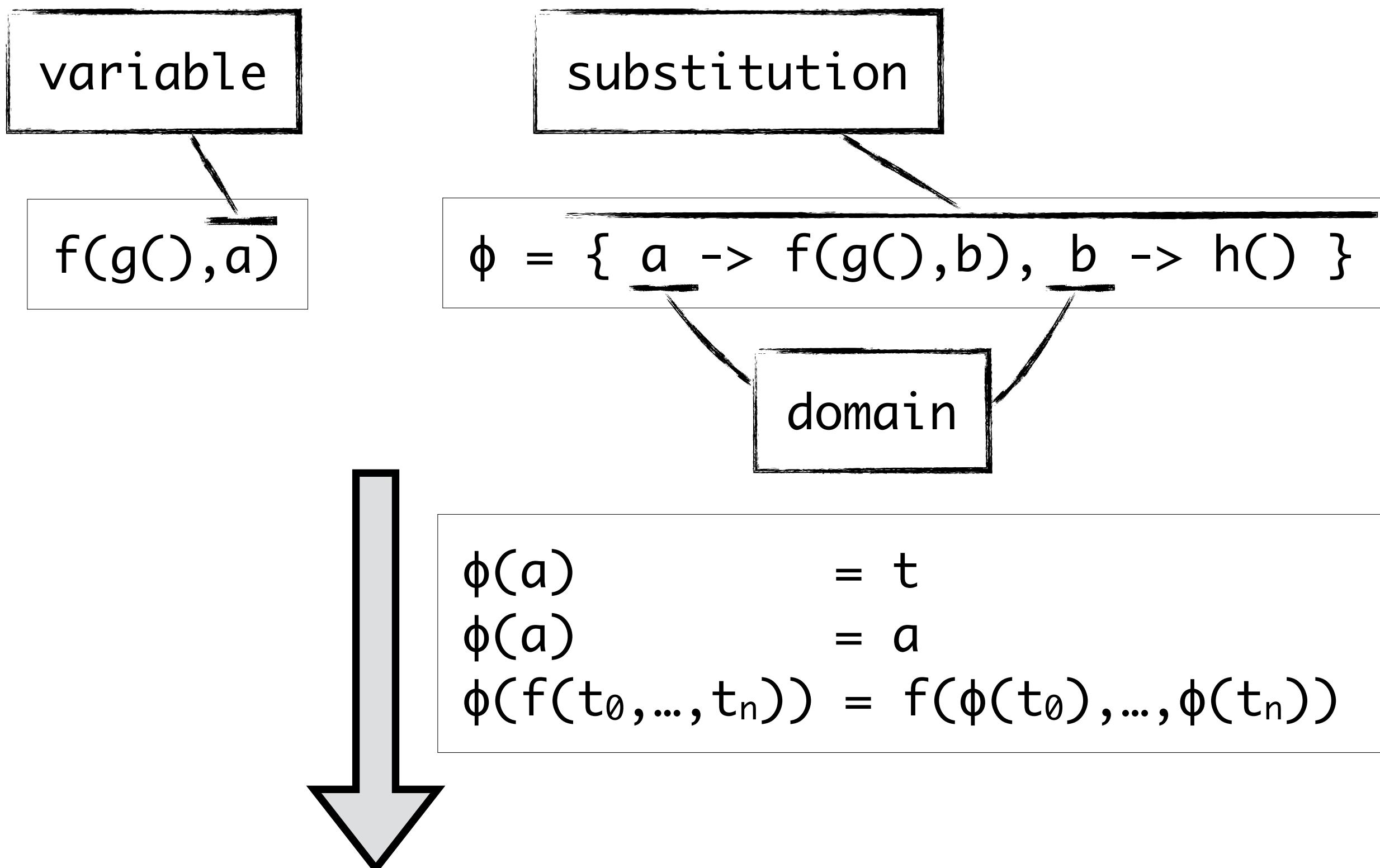
# Variables and Substitution

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$



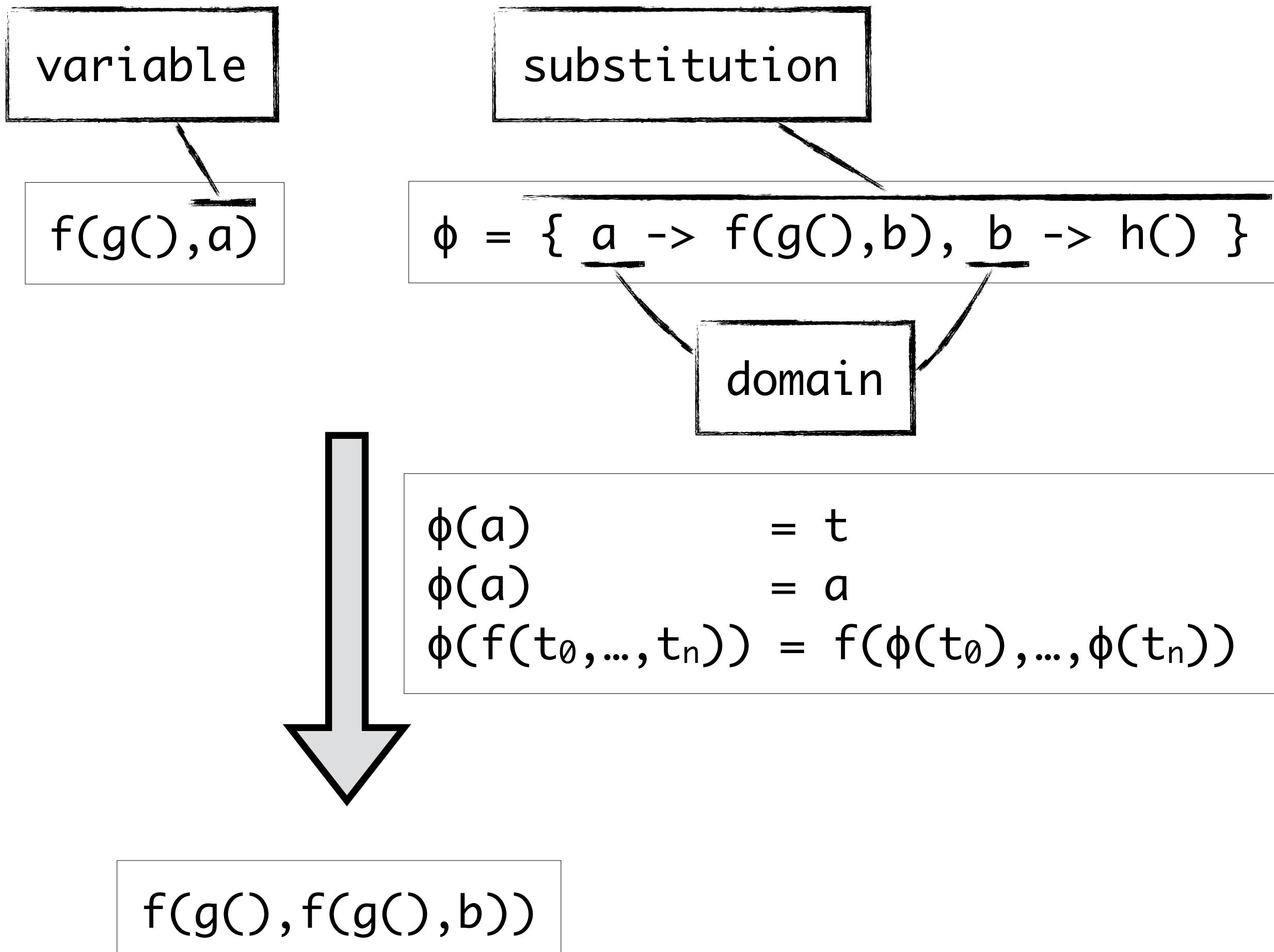
# Variables and Substitution

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$



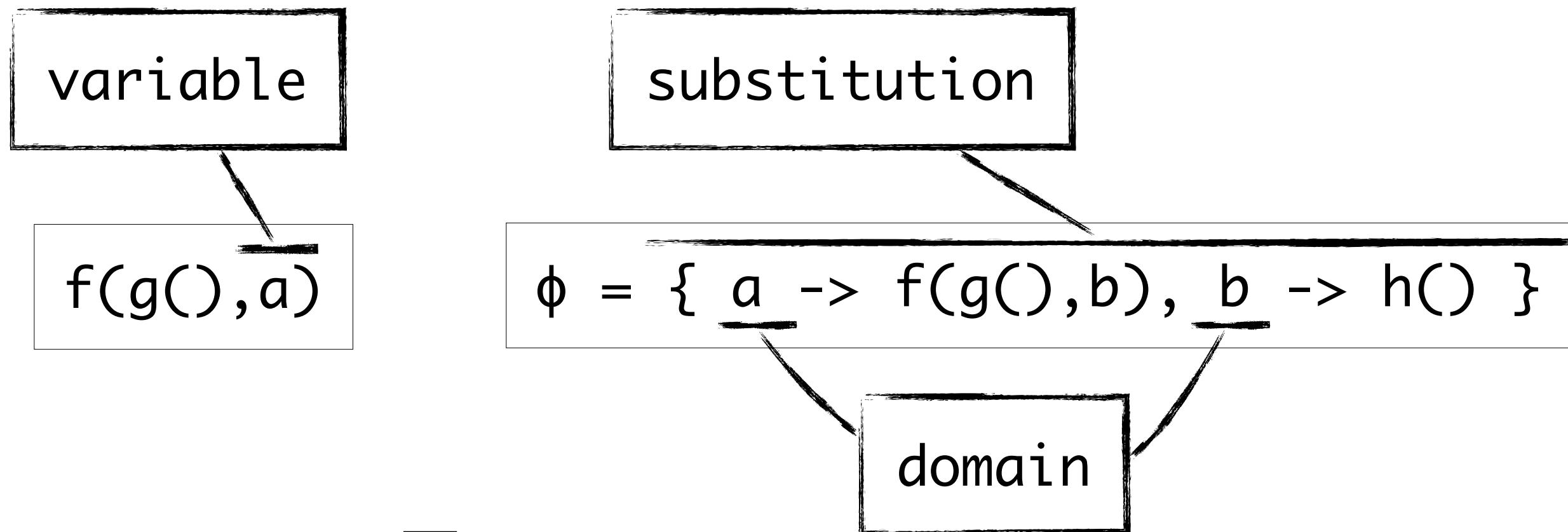
# Variables and Substitution

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

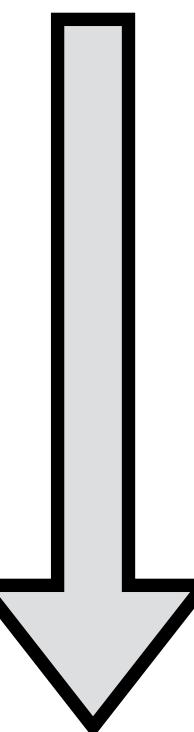


# Variables and Substitution

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$



$\phi(a) = t$  if  $\{ a \rightarrow t \}$  in  $\phi$   
 $\phi(a) = a$  otherwise  
 $\phi(f(t_0, \dots, t_n)) = f(\phi(t_0), \dots, \phi(t_n))$



$f(g(), f(g(), b))$

ground term: a term without variables

# Unifiers

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

# Unifiers

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

unifier: a substitution that makes terms equal

# Unifiers

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

unifier: a substitution that makes terms equal

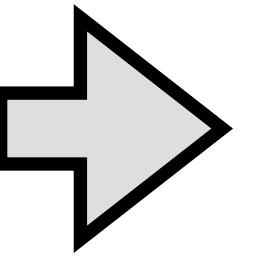
$f(a, g()) == f(h(), b)$

# Unifiers

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

unifier: a substitution that makes terms equal

$f(a, g()) == f(h(), b)$

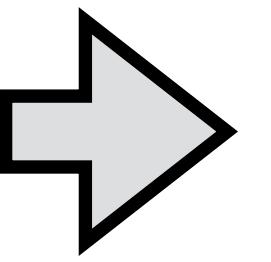


# Unifiers

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

unifier: a substitution that makes terms equal

$f(a, g()) == f(h(), b)$



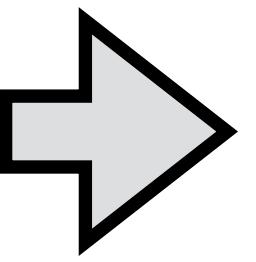
$a \rightarrow h()$   
 $b \rightarrow g()$

# Unifiers

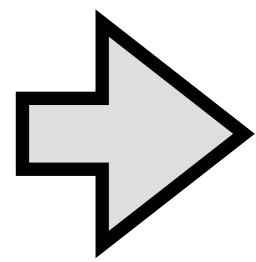
terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

unifier: a substitution that makes terms equal

$f(a, g()) == f(h(), b)$



$a \rightarrow h()$   
 $b \rightarrow g()$



# Unifiers

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

unifier: a substitution that makes terms equal

$$f(a, g()) == f(h(), b) \rightarrow a \rightarrow h() \\ b \rightarrow g() \rightarrow f(h(), g()) == f(h(), g())$$

# Unifiers

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

unifier: a substitution that makes terms equal

$$f(a, g()) == f(h(), b) \rightarrow a \rightarrow h() \\ b \rightarrow g() \rightarrow f(h(), g()) == f(h(), g())$$

$$g(a, f(b)) == g(f(h()), a)$$

# Unifiers

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

unifier: a substitution that makes terms equal

$$f(a, g()) == f(h(), b) \rightarrow a \rightarrow h() \\ b \rightarrow g()$$

$$g(a, f(b)) == g(f(h()), a)$$

# Unifiers

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

unifier: a substitution that makes terms equal

$$f(a, g()) == f(h(), b) \rightarrow a \rightarrow h() \\ b \rightarrow g()$$

$$g(a, f(b)) == g(f(h()), a) \rightarrow a \rightarrow f(h()) \\ b \rightarrow h()$$

# Unifiers

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

unifier: a substitution that makes terms equal

$$f(a, g()) == f(h(), b) \rightarrow a \rightarrow h() \quad b \rightarrow g() \rightarrow f(h(), g()) == f(h(), g())$$

$$g(a, f(b)) == g(f(h()), a) \rightarrow a \rightarrow f(h()) \quad b \rightarrow h() \rightarrow g(f(h()), a) == g(f(h()), a)$$

# Unifiers

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

unifier: a substitution that makes terms equal

$$f(a, g()) == f(h(), b) \rightarrow a \rightarrow h() \quad b \rightarrow g() \rightarrow f(h(), g()) == f(h(), g())$$

$$g(a, f(b)) == g(f(h()), a) \rightarrow a \rightarrow f(h()) \quad b \rightarrow h() \rightarrow g(f(h()), f(h())) == g(f(h()), f(h()))$$

# Unifiers

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

unifier: a substitution that makes terms equal

$$f(a, g()) == f(h(), b) \rightarrow a \rightarrow h() \quad b \rightarrow g() \rightarrow f(h(), g()) == f(h(), g())$$

$$g(a, f(b)) == g(f(h()), a) \rightarrow a \rightarrow f(h()) \quad b \rightarrow h() \rightarrow g(f(h()), f(h())) == g(f(h()), f(h()))$$

$$f(a, h()) == g(h(), b)$$

# Unifiers

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

unifier: a substitution that makes terms equal

$$f(a, g()) == f(h(), b) \rightarrow a \rightarrow h() \quad b \rightarrow g() \rightarrow f(h(), g()) == f(h(), g())$$

$$g(a, f(b)) == g(f(h()), a) \rightarrow a \rightarrow f(h()) \quad b \rightarrow h() \rightarrow g(f(h()), f(h())) == g(f(h()), f(h()))$$

$$f(a, h()) == g(h(), b) \rightarrow$$

# Unifiers

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

unifier: a substitution that makes terms equal

$$f(a, g()) == f(h(), b) \rightarrow a \rightarrow h() \quad b \rightarrow g() \rightarrow f(h(), g()) == f(h(), g())$$

$$g(a, f(b)) == g(f(h()), a) \rightarrow a \rightarrow f(h()) \quad b \rightarrow h() \rightarrow g(f(h()), f(h())) == g(f(h()), f(h()))$$

$$f(a, h()) == g(h(), b) \rightarrow \text{no unifier, } f \neq g$$

# Unifiers

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

unifier: a substitution that makes terms equal

$$f(a, g()) == f(h(), b) \rightarrow a \rightarrow h() \quad b \rightarrow g() \rightarrow f(h(), g()) == f(h(), g())$$

$$g(a, f(b)) == g(f(h()), a) \rightarrow a \rightarrow f(h()) \quad b \rightarrow h() \rightarrow g(f(h()), f(h())) == g(f(h()), f(h()))$$

$$f(a, h()) == g(h(), b) \rightarrow \text{no unifier, } f \neq g$$

$$f(b, b) == b$$

# Unifiers

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

unifier: a substitution that makes terms equal

$$f(a, g()) == f(h(), b) \rightarrow a \rightarrow h() \quad b \rightarrow g() \rightarrow f(h(), g()) == f(h(), g())$$

$$g(a, f(b)) == g(f(h()), a) \rightarrow a \rightarrow f(h()) \quad b \rightarrow h() \rightarrow g(f(h()), f(h())) == g(f(h()), f(h()))$$

$$f(a, h()) == g(h(), b) \rightarrow \text{no unifier, } f \neq g$$

$$f(b, b) == b \rightarrow$$

# Unifiers

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

unifier: a substitution that makes terms equal

$$f(a, g()) == f(h(), b) \rightarrow a \rightarrow h() \quad b \rightarrow g() \rightarrow f(h(), g()) == f(h(), g())$$

$$g(a, f(b)) == g(f(h()), a) \rightarrow a \rightarrow f(h()) \quad b \rightarrow h() \rightarrow g(f(h()), f(h())) == g(f(h()), f(h()))$$

$$f(a, h()) == g(h(), b) \rightarrow \text{no unifier, } f \neq g$$

$$f(b, b) == b \rightarrow b \rightarrow f(b, b)$$

# Unifiers

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

unifier: a substitution that makes terms equal

$$f(a, g()) == f(h(), b) \rightarrow a \rightarrow h() \quad b \rightarrow g() \rightarrow f(h(), g()) == f(h(), g())$$

$$g(a, f(b)) == g(f(h()), a) \rightarrow a \rightarrow f(h()) \quad b \rightarrow h() \rightarrow g(f(h()), f(h())) == g(f(h()), f(h()))$$

$$f(a, h()) == g(h(), b) \rightarrow \text{no unifier, } f \neq g$$

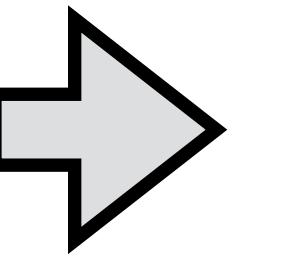
$$f(b, b) == b \rightarrow b \rightarrow f(b, b)$$

not idempotent

# Most General Unifiers

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

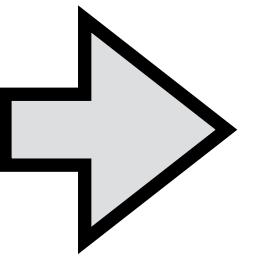
$f(a, b) == f(b, c)$



# Most General Unifiers

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

$f(a, b) == f(b, c)$

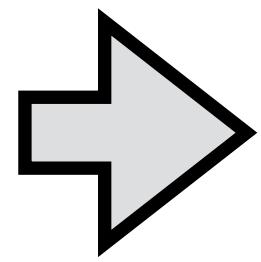


$a \rightarrow b$   
 $c \rightarrow b$

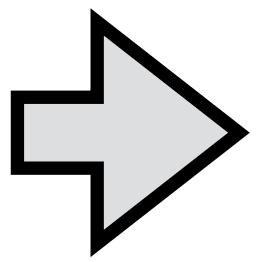
# Most General Unifiers

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

$f(a,b) == f(b,c)$



$a \rightarrow b$   
 $c \rightarrow b$

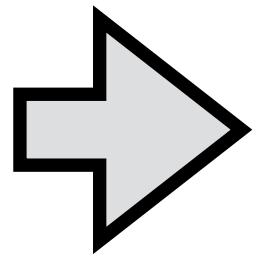


$f(b,b) == f(b,b)$

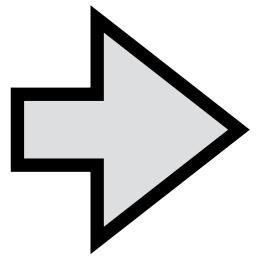
# Most General Unifiers

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

$f(a,b) == f(b,c)$



$a \rightarrow g()$   
 $b \rightarrow g()$   
 $c \rightarrow g()$

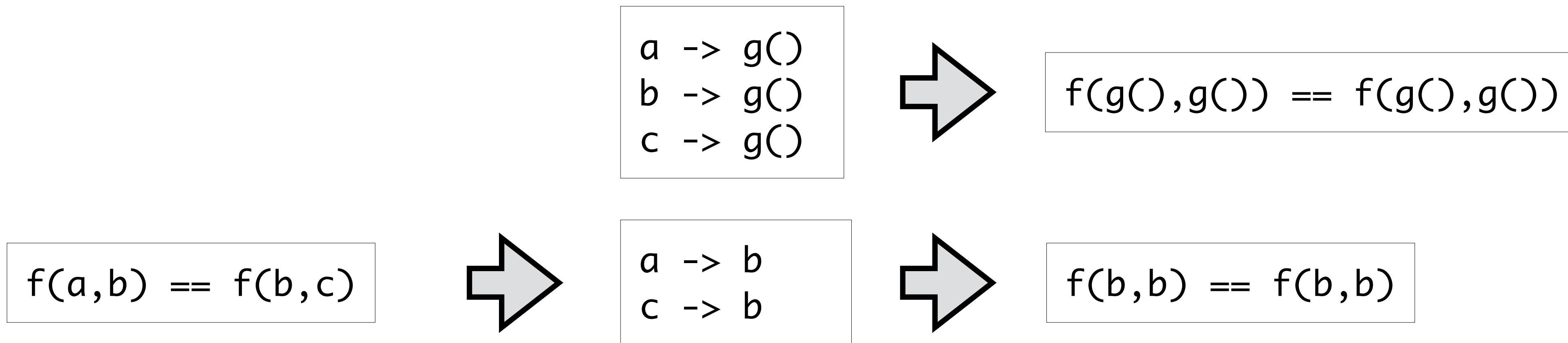


$a \rightarrow b$   
 $c \rightarrow b$

$f(b,b) == f(b,b)$

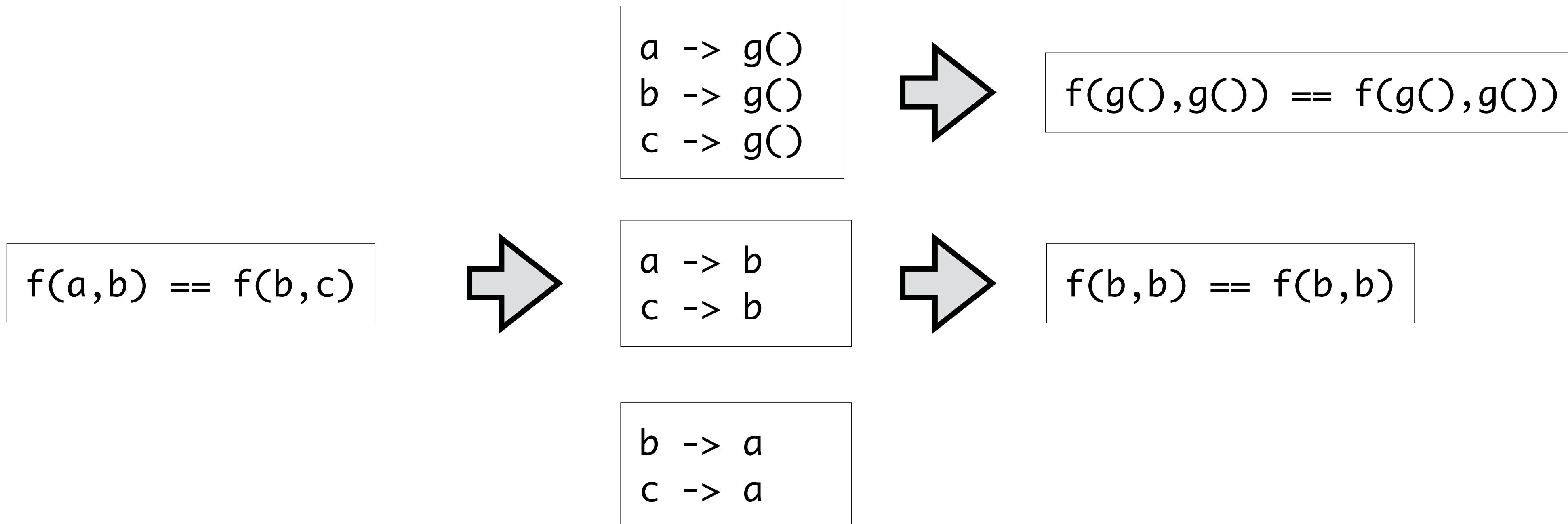
# Most General Unifiers

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$



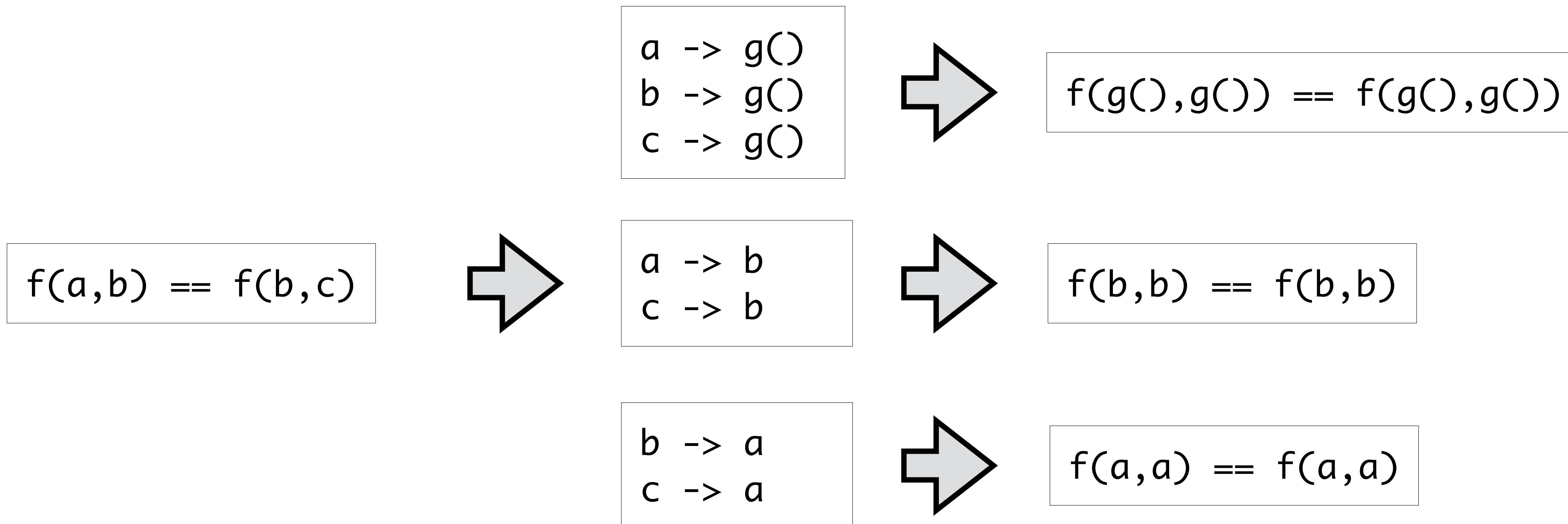
# Most General Unifiers

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$



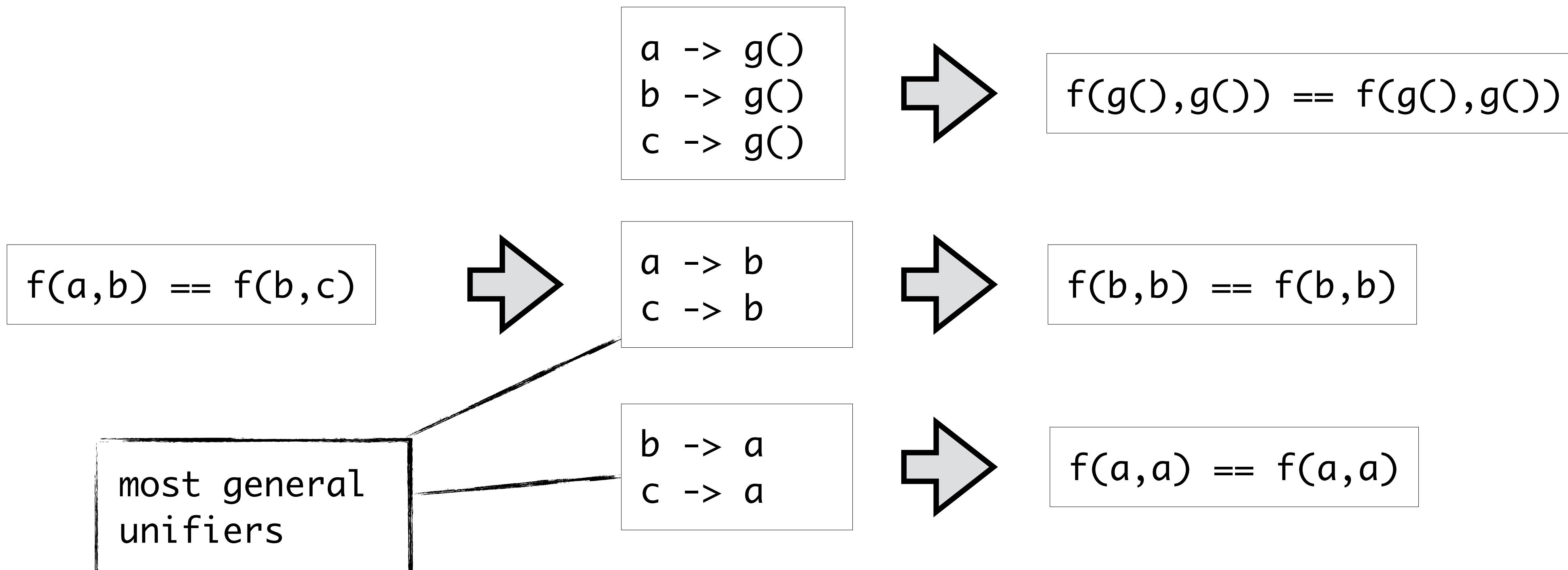
# Most General Unifiers

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$



# Most General Unifiers

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$



# Most General Unifiers

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

every unifier is an instance of a most general unifier

$$\begin{array}{l} a \rightarrow b \\ b \rightarrow b \\ c \rightarrow b \end{array}$$



$$\begin{array}{l} a \rightarrow g() \\ b \rightarrow g() \\ c \rightarrow g() \end{array}$$

# Most General Unifiers

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

every unifier is an instance of a most general unifier

(implicit) identity case

$$\begin{array}{l} a \rightarrow b \\ | \\ b \rightarrow b \\ | \\ c \rightarrow b \end{array}$$



$$\begin{array}{l} a \rightarrow g() \\ b \rightarrow g() \\ c \rightarrow g() \end{array}$$

# Most General Unifiers

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

every unifier is an instance of a most general unifier

$$\begin{array}{l} a \rightarrow b \\ b \rightarrow b \\ c \rightarrow b \end{array}$$

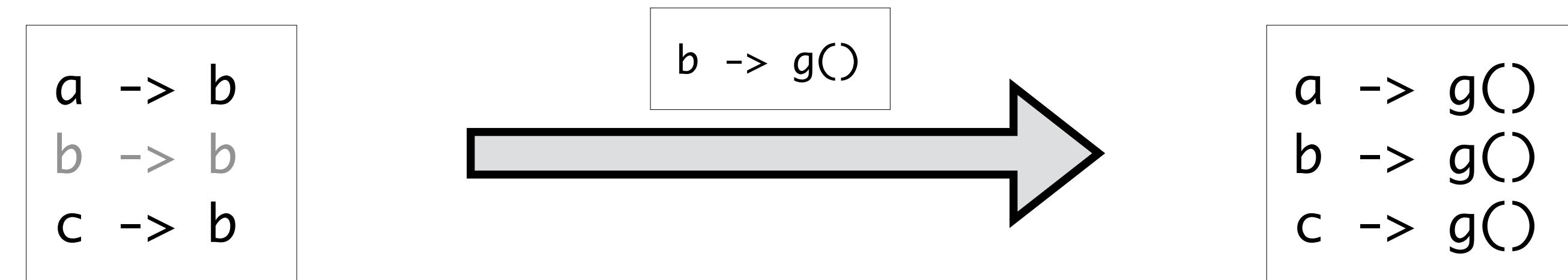


$$\begin{array}{l} a \rightarrow g() \\ b \rightarrow g() \\ c \rightarrow g() \end{array}$$

# Most General Unifiers

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

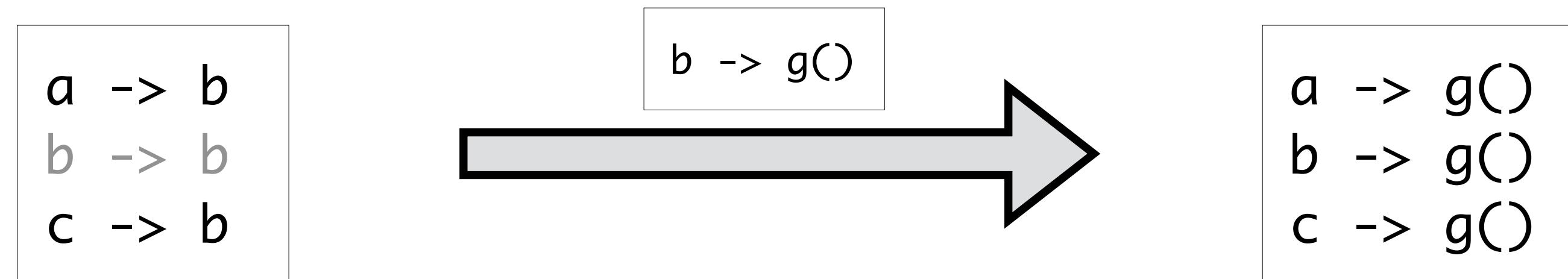
every unifier is an instance of a most general unifier



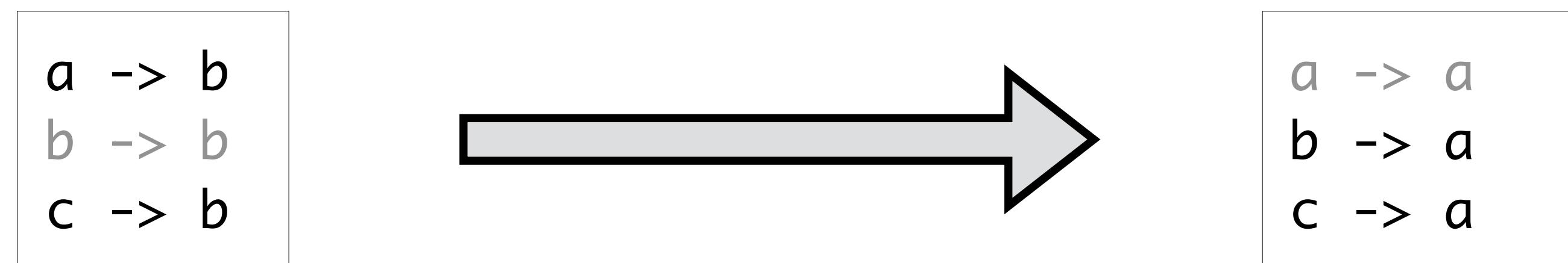
# Most General Unifiers

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

every unifier is an instance of a most general unifier



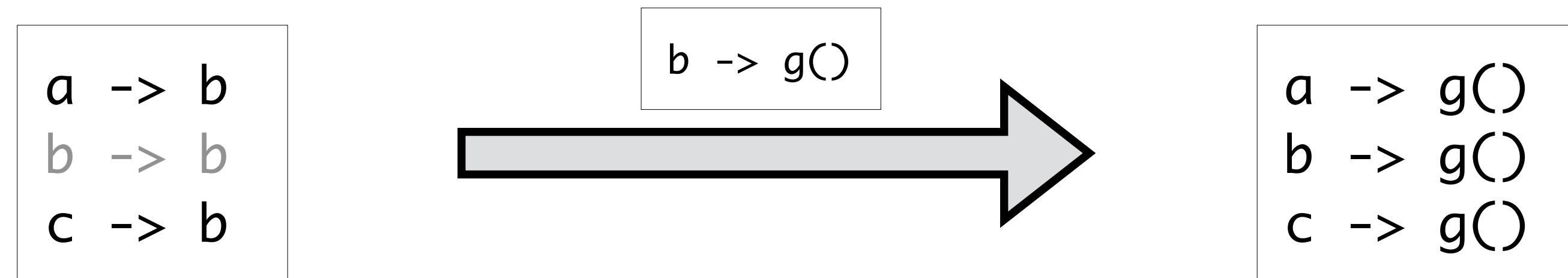
most general unifiers are related by renaming substitutions



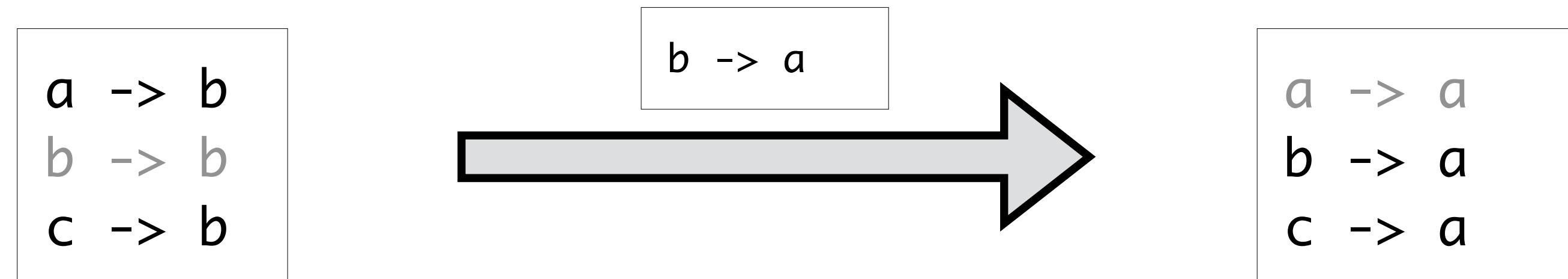
# Most General Unifiers

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

every unifier is an instance of a most general unifier



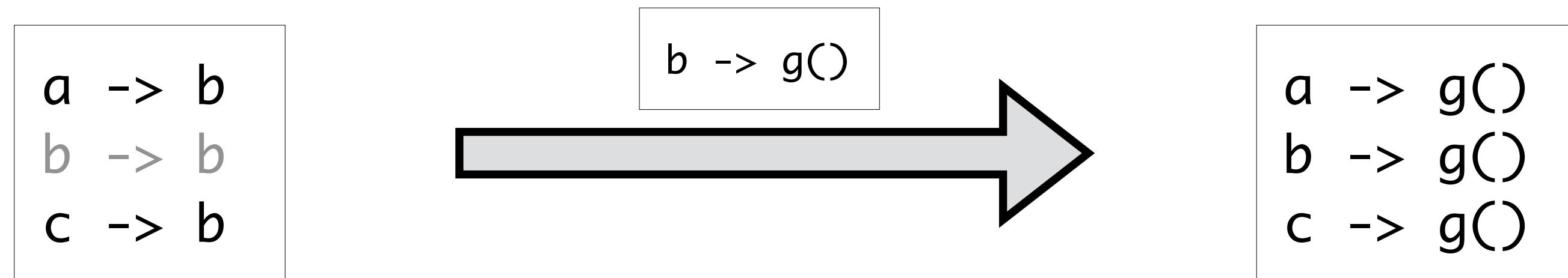
most general unifiers are related by renaming substitutions



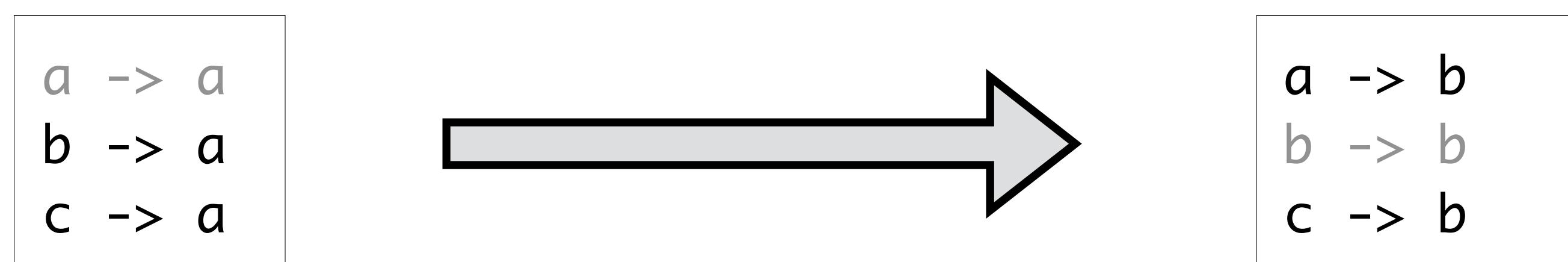
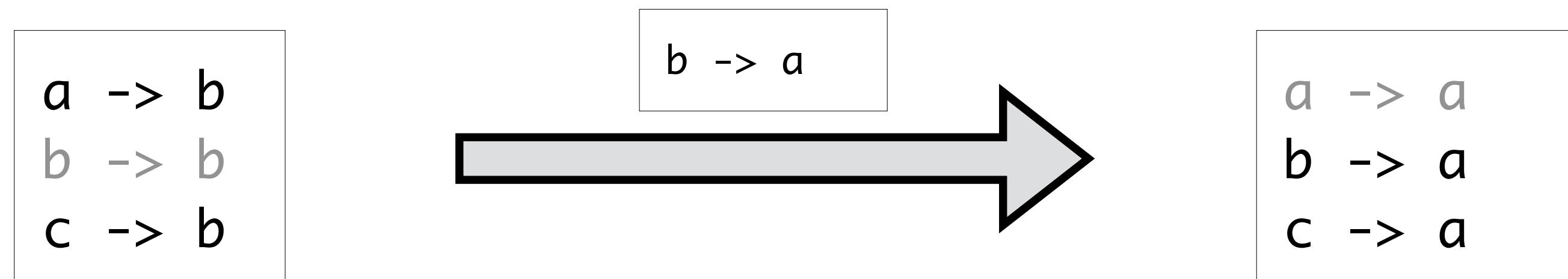
# Most General Unifiers

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

every unifier is an instance of a most general unifier



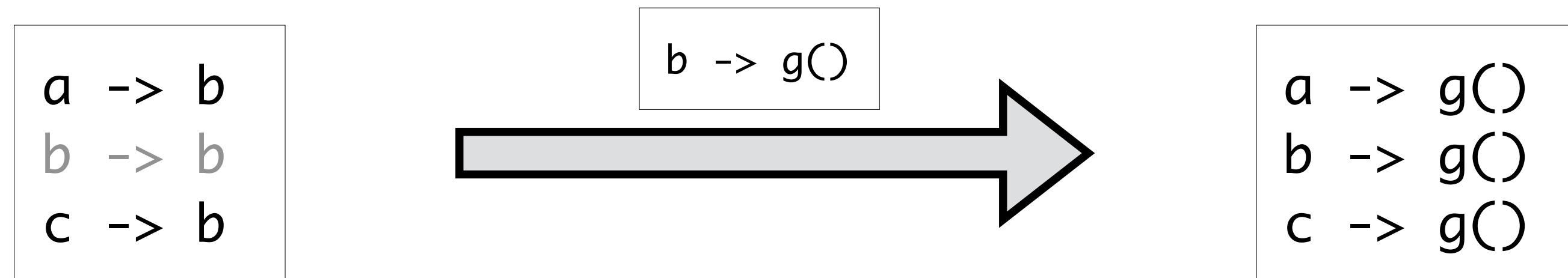
most general unifiers are related by renaming substitutions



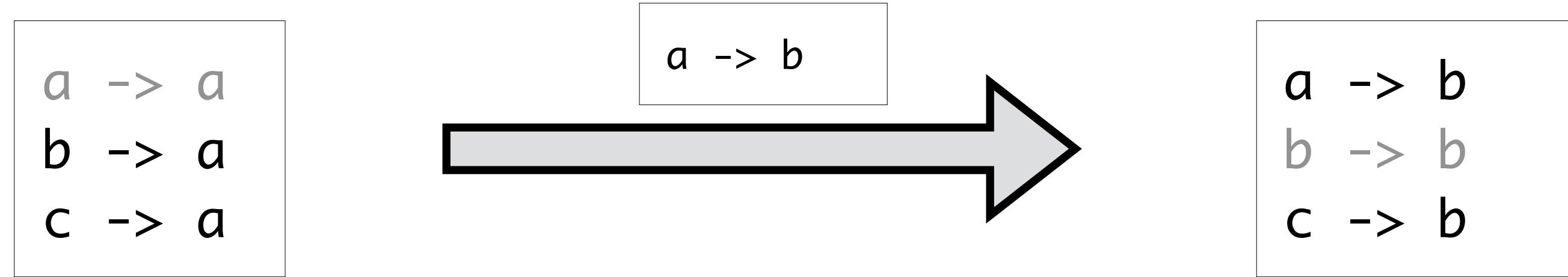
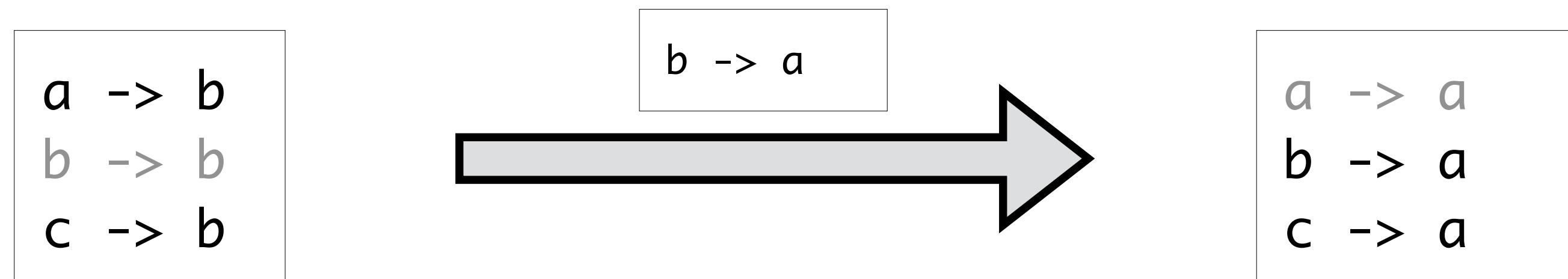
# Most General Unifiers

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

every unifier is an instance of a most general unifier



most general unifiers are related by renaming substitutions



# Unification

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

```
global  $\phi$ 
def unify(t, u):
    if t is a variable:
        t :=  $\phi(t)$ 
    if u is a variable:
        u :=  $\phi(u)$ 
    if t is a variable and t == u:
        pass
    else if t == f( $t_0, \dots, t_n$ ) and u == g( $u_0, \dots, u_m$ ):
        if f == g and n == m:
            for i := 1 to n:
                unify( $t_i, u_i$ )
        else:
            fail "different function symbols"
    else if t is not a variable:
        unify(u, t)
    else if t occurs in u:
        fail "recursive term"
    else:
         $\phi += \{ t \rightarrow u \}$ 
```

# Unification

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

```
global  $\phi$ 
def unify(t, u):
    if t is a variable:
        t :=  $\phi(t)$ 
    if u is a variable:
        u :=  $\phi(u)$ 
    if t is a variable and t == u:
        pass
    else if t == f( $t_0, \dots, t_n$ ) and u == g( $u_0, \dots, u_m$ ):
        if f == g and n == m:
            for i := 1 to n:
                unify( $t_i, u_i$ )
        else:
            fail "different function symbols"
    else if t is not a variable:
        unify(u, t)
    else if t occurs in u:
        fail "recursive term"
    else:
         $\phi += \{ t \rightarrow u \}$ 
```

$\boxed{t == a}$   
 $\boxed{\text{instantiate variable}}$

# Unification

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

```
global  $\phi$ 
def unify(t, u):
    if t is a variable:
        t :=  $\phi(t)$ 
    if u is a variable:
        u :=  $\phi(u)$ 
    if t is a variable and t == u:
        pass
    else if t == f( $t_0, \dots, t_n$ ) and u == g( $u_0, \dots, u_m$ ):
        if f == g and n == m:
            for i := 1 to n:
                unify( $t_i, u_i$ )
        else:
            fail "different function symbols"
    else if t is not a variable:
        unify(u, t)
    else if t occurs in u:
        fail "recursive term"
    else:
         $\phi += \{ t \rightarrow u \}$ 
```

$\boxed{\begin{array}{l} t == a \\ \text{instantiate variable} \\ u == b \\ \text{instantiate variable} \end{array}}$

# Unification

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

```
global  $\phi$ 
def unify(t, u):
    if t is a variable:
        t :=  $\phi(t)$ 
    if u is a variable:
        u :=  $\phi(u)$ 
    if t is a variable and t == u:
        pass
    else if t == f( $t_0, \dots, t_n$ ) and u == g( $u_0, \dots, u_m$ ):
        if f == g and n == m:
            for i := 1 to n:
                unify( $t_i, u_i$ )
        else:
            fail "different function symbols"
    else if t is not a variable:
        unify(u, t)
    else if t occurs in u:
        fail "recursive term"
    else:
         $\phi += \{ t \rightarrow u \}$ 
```

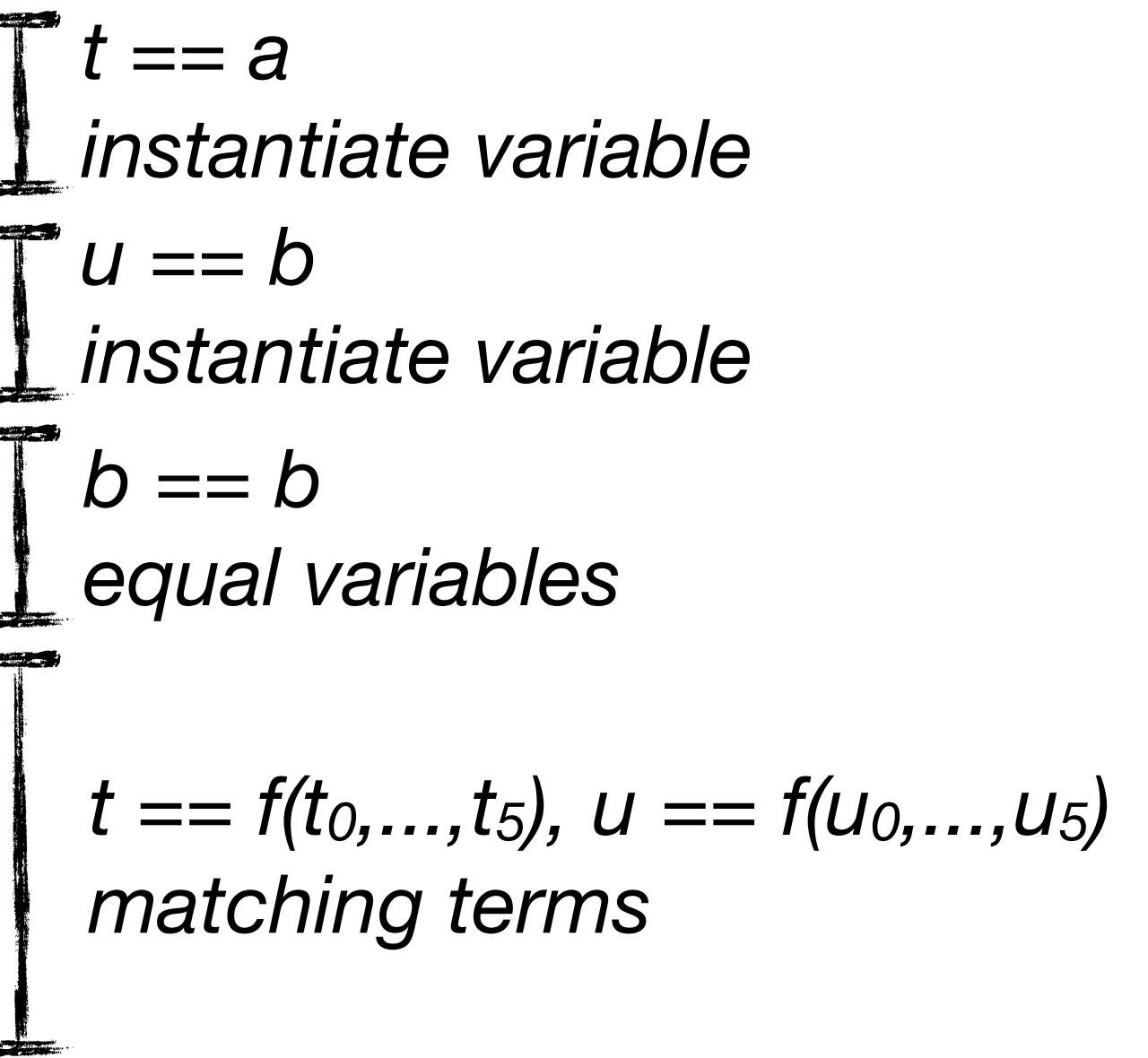
$\boxed{t == a}$   
 $\boxed{u == b}$   
 $\boxed{b == b}$

*instantiate variable*  
*instantiate variable*  
*equal variables*

# Unification

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

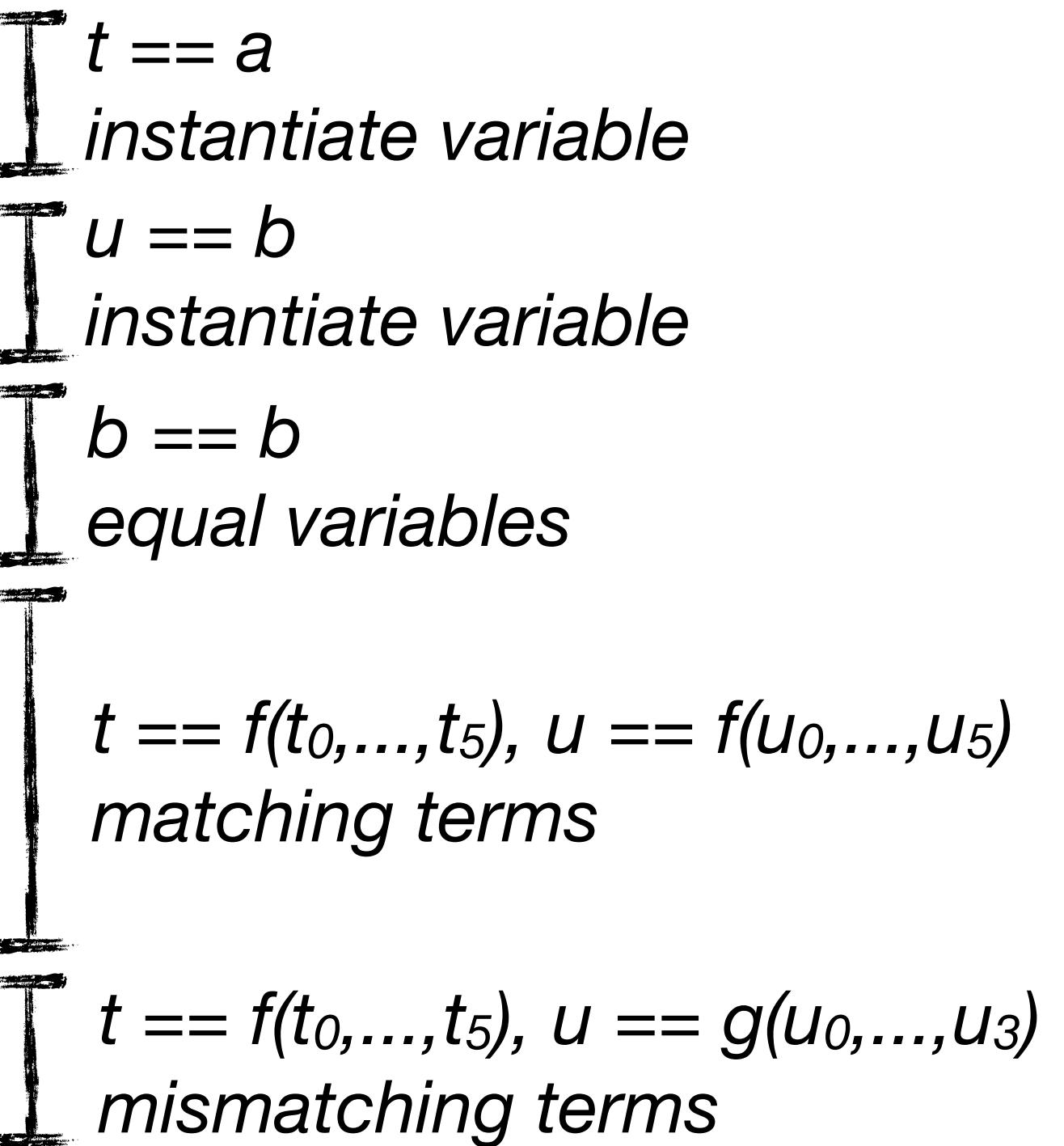
```
global  $\phi$ 
def unify(t, u):
    if t is a variable:
        t :=  $\phi(t)$ 
    if u is a variable:
        u :=  $\phi(u)$ 
    if t is a variable and t == u:
        pass
    else if t == f( $t_0, \dots, t_n$ ) and u == g( $u_0, \dots, u_m$ ):
        if f == g and n == m:
            for i := 1 to n:
                unify( $t_i, u_i$ )
        else:
            fail "different function symbols"
    else if t is not a variable:
        unify(u, t)
    else if t occurs in u:
        fail "recursive term"
    else:
         $\phi += \{ t \rightarrow u \}$ 
```



# Unification

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

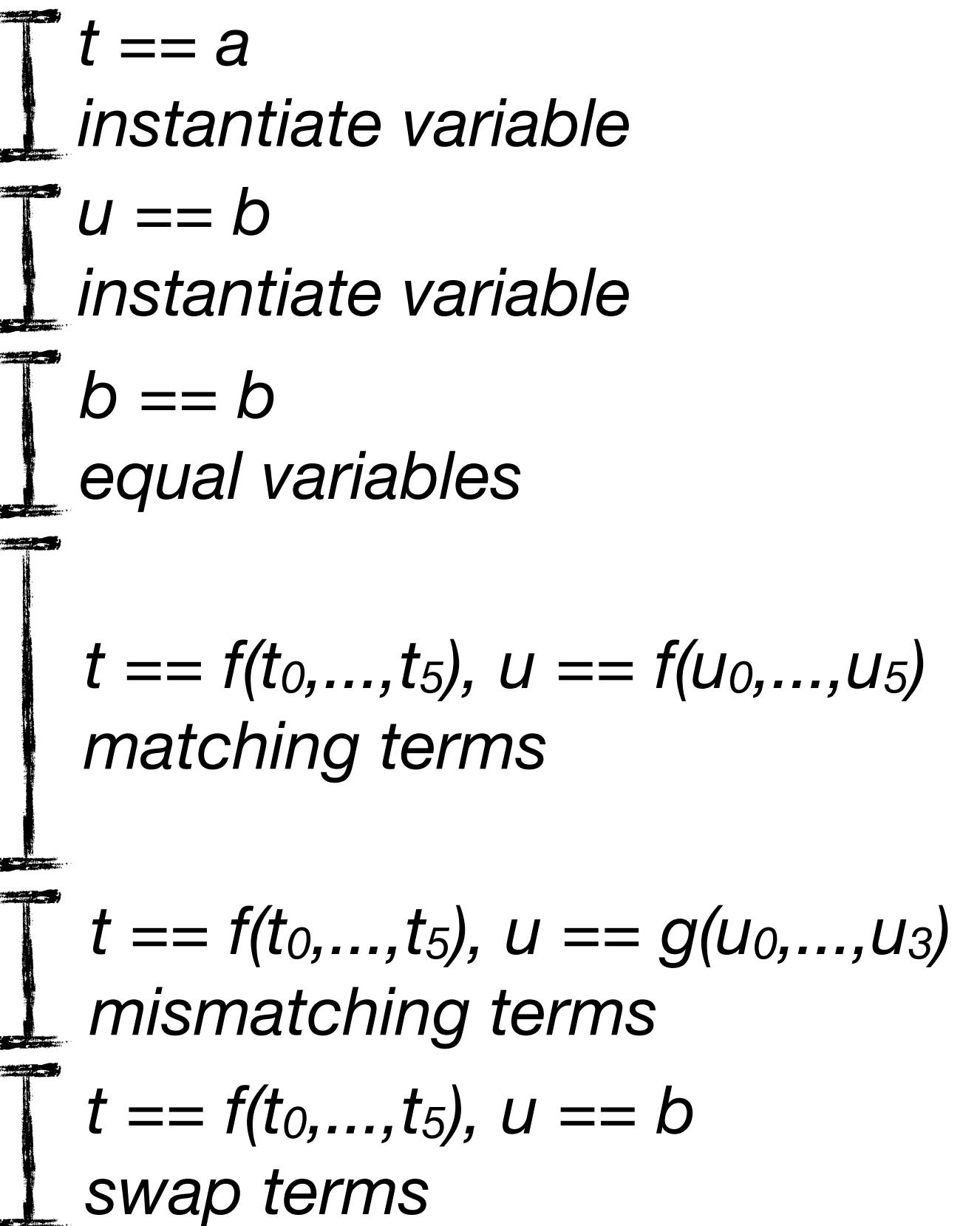
```
global  $\phi$ 
def unify(t, u):
    if t is a variable:
        t :=  $\phi(t)$ 
    if u is a variable:
        u :=  $\phi(u)$ 
    if t is a variable and t == u:
        pass
    else if t == f( $t_0, \dots, t_n$ ) and u == g( $u_0, \dots, u_m$ ):
        if f == g and n == m:
            for i := 1 to n:
                unify( $t_i, u_i$ )
        else:
            fail "different function symbols"
    else if t is not a variable:
        unify(u, t)
    else if t occurs in u:
        fail "recursive term"
    else:
         $\phi += \{ t \rightarrow u \}$ 
```



# Unification

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

```
global  $\phi$ 
def unify(t, u):
    if t is a variable:
        t :=  $\phi(t)$ 
    if u is a variable:
        u :=  $\phi(u)$ 
    if t is a variable and t == u:
        pass
    else if t == f( $t_0, \dots, t_n$ ) and u == g( $u_0, \dots, u_m$ ):
        if f == g and n == m:
            for i := 1 to n:
                unify( $t_i, u_i$ )
        else:
            fail "different function symbols"
    else if t is not a variable:
        unify(u, t)
    else if t occurs in u:
        fail "recursive term"
    else:
         $\phi += \{ t \rightarrow u \}$ 
```



# Unification

terms	$t, u$
functions	$f, g, h$
variables	$a, b, c$
substitution	$\phi$

```

global φ
def unify(t, u):
    if t is a variable:
        t := φ(t)
    if u is a variable:
        u := φ(u)
    if t is a variable and t == u:
        pass
    else if t == f(t0, ..., tn) and u == g(u0, ..., um):
        if f == g and n == m:
            for i := 1 to n:
                unify(ti, ui)
        else:
            fail "different function symbols"
    else if t is not a variable:
        unify(u, t)
    else if t occurs in u:
        fail "recursive term"
    else:
        φ += { t -> u }

```

- ─  $t == a$   
*instantiate variable*
- ─  $u == b$   
*instantiate variable*
- ─  $b == b$   
*equal variables*
- ─  $t == f(t_0, \dots, t_5), u == f(u_0, \dots, u_5)$   
*matching terms*
- ─  $t == f(t_0, \dots, t_5), u == g(u_0, \dots, u_3)$   
*mismatching terms*
- ─  $t == f(t_0, \dots, t_5), u == b$   
*swap terms*
- ─  $t == a, u == k(g(a, f))$   
*recursive terms*

# Unification

terms	$t, u$
functions	$f, g, h$
variables	$a, b, c$
substitution	$\phi$

```

global φ
def unify(t, u):
    if t is a variable:
        t := φ(t)
    if u is a variable:
        u := φ(u)
    if t is a variable and t == u:
        pass
    else if t == f(t0, ..., tn) and u == g(u0, ..., um):
        if f == g and n == m:
            for i := 1 to n:
                unify(ti, ui)
        else:
            fail "different function symbols"
    else if t is not a variable:
        unify(u, t)
    else if t occurs in u:
        fail "recursive term"
    else:
        φ += { t -> u }

```

- ─  $t == a$   
*instantiate variable*
- ─  $u == b$   
*instantiate variable*
- ─  $b == b$   
*equal variables*
- ─  $t == f(t_0, \dots, t_5), u == f(u_0, \dots, u_5)$   
*matching terms*
- ─  $t == f(t_0, \dots, t_5), u == g(u_0, \dots, u_3)$   
*mismatching terms*
- ─  $t == f(t_0, \dots, t_5), u == b$   
*swap terms*
- ─  $t == a, u == k(g(a, f()))$   
*recursive terms*
- ─  $t == a, u == k(u_0, \dots, u_5)$   
*extend unifier*

# Properties of Unification

# Properties of Unification

## Soundness

# Properties of Unification

## Soundness

- If the algorithm returns a unifier, it makes the terms equal

# Properties of Unification

## Soundness

- If the algorithm returns a unifier, it makes the terms equal

## Completeness

# Properties of Unification

## Soundness

- If the algorithm returns a unifier, it makes the terms equal

## Completeness

- If a unifier exists, the algorithm will return it

# Properties of Unification

## Soundness

- If the algorithm returns a unifier, it makes the terms equal

## Completeness

- If a unifier exists, the algorithm will return it

## Principality

# Properties of Unification

## Soundness

- If the algorithm returns a unifier, it makes the terms equal

## Completeness

- If a unifier exists, the algorithm will return it

## Principality

- If the algorithm returns a unifier, it is a most general unifier

# Properties of Unification

## Soundness

- If the algorithm returns a unifier, it makes the terms equal

## Completeness

- If a unifier exists, the algorithm will return it

## Principality

- If the algorithm returns a unifier, it is a most general unifier

## Termination

# Properties of Unification

## Soundness

- If the algorithm returns a unifier, it makes the terms equal

## Completeness

- If a unifier exists, the algorithm will return it

## Principality

- If the algorithm returns a unifier, it is a most general unifier

## Termination

- The algorithm always returns a unifier or fails

# Conclusion

# Summary

What is the meaning of constraints?

# Summary

## What is the meaning of constraints?

- Formally described by constraint semantics

# Summary

## What is the meaning of constraints?

- Formally described by constraint semantics
- Semantics classify solutions, but do not compute them

# Summary

## What is the meaning of constraints?

- Formally described by constraint semantics
- Semantics classify solutions, but do not compute them
- Semantics are expressed in terms of other theories

# Summary

## What is the meaning of constraints?

- Formally described by constraint semantics
- Semantics classify solutions, but do not compute them
- Semantics are expressed in terms of other theories
  - ▶ Syntactic equality

# Summary

## What is the meaning of constraints?

- Formally described by constraint semantics
- Semantics classify solutions, but do not compute them
- Semantics are expressed in terms of other theories
  - ▶ Syntactic equality
  - ▶ Scope graph resolution

# Summary

## What is the meaning of constraints?

- Formally described by constraint semantics
- Semantics classify solutions, but do not compute them
- Semantics are expressed in terms of other theories
  - ▶ Syntactic equality
  - ▶ Scope graph resolution

## What techniques can we use to implement solvers?

# Summary

## What is the meaning of constraints?

- Formally described by constraint semantics
- Semantics classify solutions, but do not compute them
- Semantics are expressed in terms of other theories
  - ▶ Syntactic equality
  - ▶ Scope graph resolution

## What techniques can we use to implement solvers?

- Constraint simplification

# Summary

## What is the meaning of constraints?

- Formally described by constraint semantics
- Semantics classify solutions, but do not compute them
- Semantics are expressed in terms of other theories
  - ▶ Syntactic equality
  - ▶ Scope graph resolution

## What techniques can we use to implement solvers?

- Constraint simplification
  - ▶ Simplification rules

# Summary

## What is the meaning of constraints?

- Formally described by constraint semantics
- Semantics classify solutions, but do not compute them
- Semantics are expressed in terms of other theories
  - ▶ Syntactic equality
  - ▶ Scope graph resolution

## What techniques can we use to implement solvers?

- Constraint simplification
  - ▶ Simplification rules
  - ▶ Depends on built-in procedures to unify or resolve names

# Summary

## What is the meaning of constraints?

- Formally described by constraint semantics
- Semantics classify solutions, but do not compute them
- Semantics are expressed in terms of other theories
  - ▶ Syntactic equality
  - ▶ Scope graph resolution

## What techniques can we use to implement solvers?

- Constraint simplification
  - ▶ Simplification rules
  - ▶ Depends on built-in procedures to unify or resolve names
- Unification

# Summary

## What is the meaning of constraints?

- Formally described by constraint semantics
- Semantics classify solutions, but do not compute them
- Semantics are expressed in terms of other theories
  - ▶ Syntactic equality
  - ▶ Scope graph resolution

## What techniques can we use to implement solvers?

- Constraint simplification
  - ▶ Simplification rules
  - ▶ Depends on built-in procedures to unify or resolve names
- Unification
  - ▶ Unifiers make terms with variables equal

# Summary

## What is the meaning of constraints?

- Formally described by constraint semantics
- Semantics classify solutions, but do not compute them
- Semantics are expressed in terms of other theories
  - ▶ Syntactic equality
  - ▶ Scope graph resolution

## What techniques can we use to implement solvers?

- Constraint simplification
  - ▶ Simplification rules
  - ▶ Depends on built-in procedures to unify or resolve names
- Unification
  - ▶ Unifiers make terms with variables equal
  - ▶ Unification computes most general unifiers

# Summary

## What is the meaning of constraints?

- Formally described by constraint semantics
- Semantics classify solutions, but do not compute them
- Semantics are expressed in terms of other theories
  - ▶ Syntactic equality
  - ▶ Scope graph resolution

## What techniques can we use to implement solvers?

- Constraint simplification
  - ▶ Simplification rules
  - ▶ Depends on built-in procedures to unify or resolve names
- Unification
  - ▶ Unifiers make terms with variables equal
  - ▶ Unification computes most general unifiers

## What is the relation between solver and semantics?

# Summary

## What is the meaning of constraints?

- Formally described by constraint semantics
- Semantics classify solutions, but do not compute them
- Semantics are expressed in terms of other theories
  - ▶ Syntactic equality
  - ▶ Scope graph resolution

## What techniques can we use to implement solvers?

- Constraint simplification
  - ▶ Simplification rules
  - ▶ Depends on built-in procedures to unify or resolve names
- Unification
  - ▶ Unifiers make terms with variables equal
  - ▶ Unification computes most general unifiers

## What is the relation between solver and semantics?

- Soundness: any solution satisfies the semantics

# Summary

## What is the meaning of constraints?

- Formally described by constraint semantics
- Semantics classify solutions, but do not compute them
- Semantics are expressed in terms of other theories
  - ▶ Syntactic equality
  - ▶ Scope graph resolution

## What techniques can we use to implement solvers?

- Constraint simplification
  - ▶ Simplification rules
  - ▶ Depends on built-in procedures to unify or resolve names
- Unification
  - ▶ Unifiers make terms with variables equal
  - ▶ Unification computes most general unifiers

## What is the relation between solver and semantics?

- Soundness: any solution satisfies the semantics
- Completeness: if a solution exists, the solver finds it

# Summary

## What is the meaning of constraints?

- Formally described by constraint semantics
- Semantics classify solutions, but do not compute them
- Semantics are expressed in terms of other theories
  - ▶ Syntactic equality
  - ▶ Scope graph resolution

## What techniques can we use to implement solvers?

- Constraint simplification
  - ▶ Simplification rules
  - ▶ Depends on built-in procedures to unify or resolve names
- Unification
  - ▶ Unifiers make terms with variables equal
  - ▶ Unification computes most general unifiers

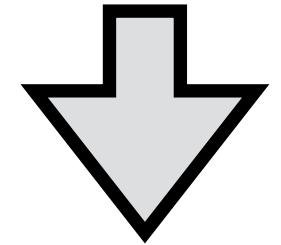
## What is the relation between solver and semantics?

- Soundness: any solution satisfies the semantics
- Completeness: if a solution exists, the solver finds it
- Principality: the solver computes most general solutions

# **Efficient Unification with Union-Find**

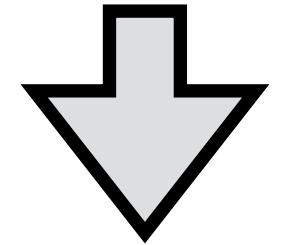
# Complexity of Unification

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

$$h(a_1, \dots, a_n), f(b_0, b_0), \dots, f(b_{n-1}, b_{n-1}), a_n == \\ h(f(a_0, a_0), \dots, f(a_{n-1}, a_{n-1}), b_1, \dots, b_{n-1}, b_n)$$


# Complexity of Unification

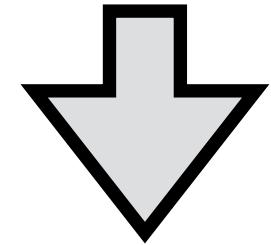
terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

$$h(a_1, \dots, a_n), f(b_0, b_0), \dots, f(b_{n-1}, b_{n-1}), a_n == \\ h(f(a_0, a_0), \dots, f(a_{n-1}, a_{n-1}), b_1, \dots, b_{n-1}, b_n)$$

$$a_1 \rightarrow f(a_0, a_0) \\ a_2 \rightarrow f(f(a_0, a_0), f(a_0, a_0)) \\ a_i \rightarrow \dots 2^{i+1}-1 \text{ subterms} \dots \\ b_1 \rightarrow f(a_0, a_0) \\ b_2 \rightarrow f(f(a_0, a_0), f(a_0, a_0)) \\ b_i \rightarrow \dots 2^{i+1}-1 \text{ subterms} \dots$$

# Complexity of Unification

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

## Space complexity

$$h(a_1, \dots, a_n, f(b_0, b_0), \dots, f(b_{n-1}, b_{n-1}), a_n) == \\ h(f(a_0, a_0), \dots, f(a_{n-1}, a_{n-1}), b_1, \dots, b_{n-1}, b_n)$$

$$a_1 \rightarrow f(a_0, a_0)$$
$$a_2 \rightarrow f(f(a_0, a_0), f(a_0, a_0))$$
$$a_i \rightarrow \dots 2^{i+1}-1 \text{ subterms} \dots$$
$$b_1 \rightarrow f(a_0, a_0)$$
$$b_2 \rightarrow f(f(a_0, a_0), f(a_0, a_0))$$
$$b_i \rightarrow \dots 2^{i+1}-1 \text{ subterms} \dots$$

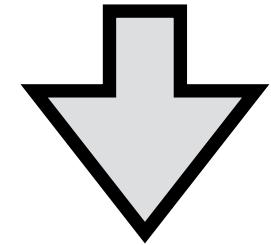
# Complexity of Unification

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

## Space complexity

- Exponential

$$\begin{array}{l} h(a_1, \dots, a_n, f(b_0, b_0), \dots, f(b_{n-1}, b_{n-1}), a_n) == \\ h(f(a_0, a_0), \dots, f(a_{n-1}, a_{n-1}), b_1, \dots, b_{n-1}, b_n) \end{array}$$



$a_1 \rightarrow f(a_0, a_0)$   
 $a_2 \rightarrow f(f(a_0, a_0), f(a_0, a_0))$   
 $a_i \rightarrow \dots 2^{i+1}-1 \text{ subterms} \dots$   
 $b_1 \rightarrow f(a_0, a_0)$   
 $b_2 \rightarrow f(f(a_0, a_0), f(a_0, a_0))$   
 $b_i \rightarrow \dots 2^{i+1}-1 \text{ subterms} \dots$

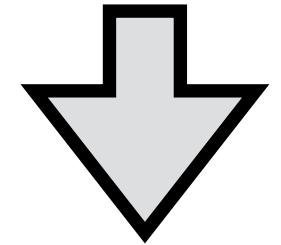
# Complexity of Unification

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

## Space complexity

- Exponential
- Representation of unifier

$$\begin{aligned} h(a_1, \dots, a_n, f(b_0, b_0), \dots, f(b_{n-1}, b_{n-1}), a_n) = \\ h(f(a_0, a_0), \dots, f(a_{n-1}, a_{n-1}), b_1, \dots, b_{n-1}, b_n) \end{aligned}$$



$a_1 \rightarrow f(a_0, a_0)$   
 $a_2 \rightarrow f(f(a_0, a_0), f(a_0, a_0))$   
 $a_i \rightarrow \dots 2^{i+1}-1 \text{ subterms} \dots$   
 $b_1 \rightarrow f(a_0, a_0)$   
 $b_2 \rightarrow f(f(a_0, a_0), f(a_0, a_0))$   
 $b_i \rightarrow \dots 2^{i+1}-1 \text{ subterms} \dots$

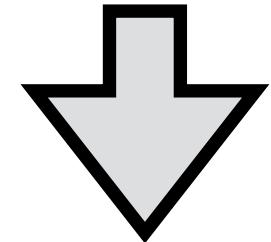
# Complexity of Unification

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

## Space complexity

- Exponential
- Representation of unifier

$$\begin{aligned} h(a_1, \dots, a_n, f(b_0, b_0), \dots, f(b_{n-1}, b_{n-1}), a_n) = \\ h(f(a_0, a_0), \dots, f(a_{n-1}, a_{n-1}), b_1, \dots, b_{n-1}, b_n) \end{aligned}$$



$a_1 \rightarrow f(a_0, a_0)$   
 $a_2 \rightarrow f(f(a_0, a_0), f(a_0, a_0))$   
 $a_i \rightarrow \dots 2^{i+1}-1 \text{ subterms} \dots$   
 $b_1 \rightarrow f(a_0, a_0)$   
 $b_2 \rightarrow f(f(a_0, a_0), f(a_0, a_0))$   
 $b_i \rightarrow \dots 2^{i+1}-1 \text{ subterms} \dots$

fully applied

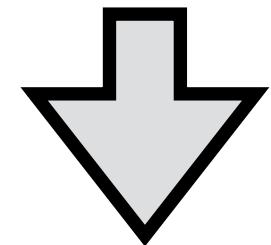
# Complexity of Unification

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

## Space complexity

- Exponential
- Representation of unifier

$$h(a_1, \dots, a_n, f(b_0, b_0), \dots, f(b_{n-1}, b_{n-1}), a_n) == \\ h(f(a_0, a_0), \dots, f(a_{n-1}, a_{n-1}), b_1, \dots, b_{n-1}, b_n)$$



$a_1 \rightarrow f(a_0, a_0)$   
 $a_2 \rightarrow f(f(a_0, a_0), f(a_0, a_0))$   
 $a_i \rightarrow \dots 2^{i+1}-1 \text{ subterms} \dots$   
 $b_1 \rightarrow f(a_0, a_0)$   
 $b_2 \rightarrow f(f(a_0, a_0), f(a_0, a_0))$   
 $b_i \rightarrow \dots 2^{i+1}-1 \text{ subterms} \dots$

fully applied

triangular

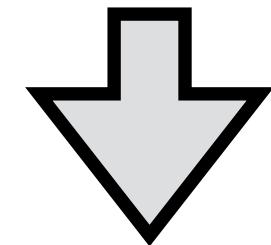
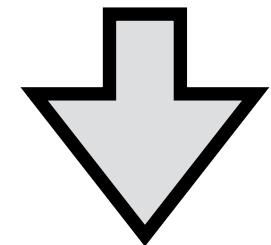
# Complexity of Unification

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

## Space complexity

- Exponential
- Representation of unifier

$$\begin{array}{l} h(a_1, \dots, a_n, f(b_0, b_0), \dots, f(b_{n-1}, b_{n-1}), a_n) == \\ h(f(a_0, a_0), \dots, f(a_{n-1}, a_{n-1}), b_1, \dots, b_{n-1}, b_n) \end{array}$$



$a_1 \rightarrow f(a_0, a_0)$   
 $a_2 \rightarrow f(f(a_0, a_0), f(a_0, a_0))$   
 $a_i \rightarrow \dots 2^{i+1}-1 \text{ subterms} \dots$   
 $b_1 \rightarrow f(a_0, a_0)$   
 $b_2 \rightarrow f(f(a_0, a_0), f(a_0, a_0))$   
 $b_i \rightarrow \dots 2^{i+1}-1 \text{ subterms} \dots$

$a_1 \rightarrow f(a_0, a_0)$   
 $a_2 \rightarrow f(a_1, a_1)$   
 $a_i \rightarrow \dots 3 \text{ subterms} \dots$   
 $b_1 \rightarrow f(a_0, a_0)$   
 $b_2 \rightarrow f(a_1, a_1)$   
 $b_i \rightarrow \dots 3 \text{ subterms} \dots$

fully applied

triangular

# Complexity of Unification

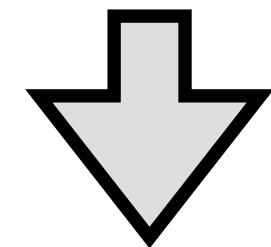
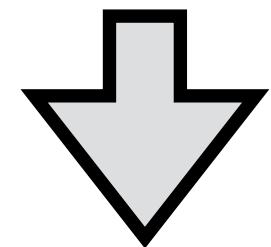
terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

## Space complexity

- Exponential
- Representation of unifier

$$\begin{array}{l} h(a_1, \dots, a_n, f(b_0, b_0), \dots, f(b_{n-1}, b_{n-1}), a_n) == \\ h(f(a_0, a_0), \dots, f(a_{n-1}, a_{n-1}), b_1, \dots, b_{n-1}, b_n) \end{array}$$

## Time complexity



$a_1 \rightarrow f(a_0, a_0)$   
 $a_2 \rightarrow f(f(a_0, a_0), f(a_0, a_0))$   
 $a_i \rightarrow \dots 2^{i+1}-1 \text{ subterms} \dots$   
 $b_1 \rightarrow f(a_0, a_0)$   
 $b_2 \rightarrow f(f(a_0, a_0), f(a_0, a_0))$   
 $b_i \rightarrow \dots 2^{i+1}-1 \text{ subterms} \dots$

$a_1 \rightarrow f(a_0, a_0)$   
 $a_2 \rightarrow f(a_1, a_1)$   
 $a_i \rightarrow \dots 3 \text{ subterms} \dots$   
 $b_1 \rightarrow f(a_0, a_0)$   
 $b_2 \rightarrow f(a_1, a_1)$   
 $b_i \rightarrow \dots 3 \text{ subterms} \dots$

fully applied

triangular

# Complexity of Unification

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

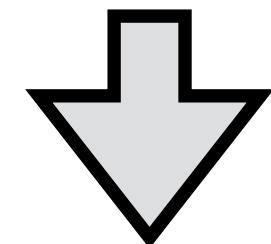
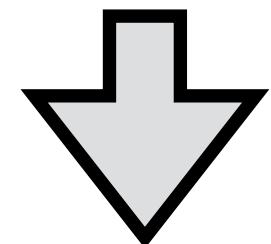
## Space complexity

- Exponential
- Representation of unifier

$$\begin{aligned} h(a_1, \dots, a_n, f(b_0, b_0), \dots, f(b_{n-1}, b_{n-1}), a_n) = \\ h(f(a_0, a_0), \dots, f(a_{n-1}, a_{n-1}), b_1, \dots, b_{n-1}, b_n) \end{aligned}$$

## Time complexity

- Exponential



$a_1 \rightarrow f(a_0, a_0)$   
 $a_2 \rightarrow f(f(a_0, a_0), f(a_0, a_0))$   
 $a_i \rightarrow \dots 2^{i+1}-1 \text{ subterms} \dots$   
 $b_1 \rightarrow f(a_0, a_0)$   
 $b_2 \rightarrow f(f(a_0, a_0), f(a_0, a_0))$   
 $b_i \rightarrow \dots 2^{i+1}-1 \text{ subterms} \dots$

$a_1 \rightarrow f(a_0, a_0)$   
 $a_2 \rightarrow f(a_1, a_1)$   
 $a_i \rightarrow \dots 3 \text{ subterms} \dots$   
 $b_1 \rightarrow f(a_0, a_0)$   
 $b_2 \rightarrow f(a_1, a_1)$   
 $b_i \rightarrow \dots 3 \text{ subterms} \dots$

fully applied

triangular

# Complexity of Unification

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

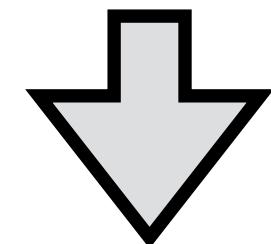
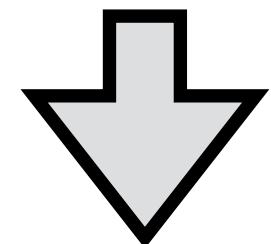
## Space complexity

- Exponential
- Representation of unifier

$$\begin{aligned} h(a_1, \dots, a_n, f(b_0, b_0), \dots, f(b_{n-1}, b_{n-1}), a_n) = \\ h(f(a_0, a_0), \dots, f(a_{n-1}, a_{n-1}), b_1, \dots, b_{n-1}, b_n) \end{aligned}$$

## Time complexity

- Exponential
- Recursive calls on terms



$a_1 \rightarrow f(a_0, a_0)$   
 $a_2 \rightarrow f(f(a_0, a_0), f(a_0, a_0))$   
 $a_i \rightarrow \dots 2^{i+1}-1 \text{ subterms} \dots$   
 $b_1 \rightarrow f(a_0, a_0)$   
 $b_2 \rightarrow f(f(a_0, a_0), f(a_0, a_0))$   
 $b_i \rightarrow \dots 2^{i+1}-1 \text{ subterms} \dots$

$a_1 \rightarrow f(a_0, a_0)$   
 $a_2 \rightarrow f(a_1, a_1)$   
 $a_i \rightarrow \dots 3 \text{ subterms} \dots$   
 $b_1 \rightarrow f(a_0, a_0)$   
 $b_2 \rightarrow f(a_1, a_1)$   
 $b_i \rightarrow \dots 3 \text{ subterms} \dots$

fully applied

triangular

# Complexity of Unification

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

## Space complexity

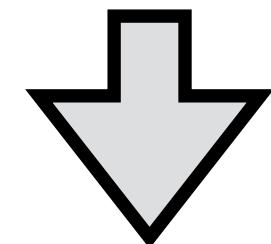
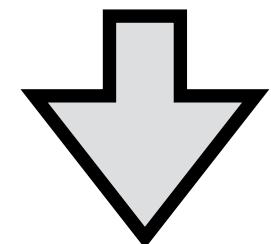
- Exponential
- Representation of unifier

$$\begin{aligned} h(a_1, \dots, a_n, f(b_0, b_0), \dots, f(b_{n-1}, b_{n-1}), a_n) = \\ h(f(a_0, a_0), \dots, f(a_{n-1}, a_{n-1}), b_1, \dots, b_{n-1}, b_n) \end{aligned}$$

## Time complexity

- Exponential
- Recursive calls on terms

## Solution



$a_1 \rightarrow f(a_0, a_0)$   
 $a_2 \rightarrow f(f(a_0, a_0), f(a_0, a_0))$   
 $a_i \rightarrow \dots 2^{i+1}-1 \text{ subterms} \dots$   
 $b_1 \rightarrow f(a_0, a_0)$   
 $b_2 \rightarrow f(f(a_0, a_0), f(a_0, a_0))$   
 $b_i \rightarrow \dots 2^{i+1}-1 \text{ subterms} \dots$

$a_1 \rightarrow f(a_0, a_0)$   
 $a_2 \rightarrow f(a_1, a_1)$   
 $a_i \rightarrow \dots 3 \text{ subterms} \dots$   
 $b_1 \rightarrow f(a_0, a_0)$   
 $b_2 \rightarrow f(a_1, a_1)$   
 $b_i \rightarrow \dots 3 \text{ subterms} \dots$

fully applied

triangular

# Complexity of Unification

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

## Space complexity

- Exponential
- Representation of unifier

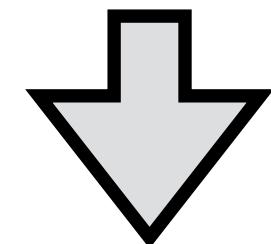
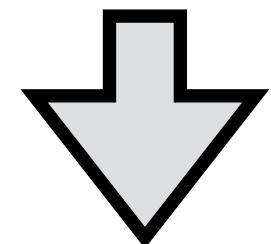
$$\begin{aligned} h(a_1, \dots, a_n, f(b_0, b_0), \dots, f(b_{n-1}, b_{n-1}), a_n) = \\ h(f(a_0, a_0), \dots, f(a_{n-1}, a_{n-1}), b_1, \dots, b_{n-1}, b_n) \end{aligned}$$

## Time complexity

- Exponential
- Recursive calls on terms

## Solution

- Union-Find algorithm



$a_1 \rightarrow f(a_0, a_0)$   
 $a_2 \rightarrow f(f(a_0, a_0), f(a_0, a_0))$   
 $a_i \rightarrow \dots 2^{i+1}-1 \text{ subterms} \dots$   
 $b_1 \rightarrow f(a_0, a_0)$   
 $b_2 \rightarrow f(f(a_0, a_0), f(a_0, a_0))$   
 $b_i \rightarrow \dots 2^{i+1}-1 \text{ subterms} \dots$

$a_1 \rightarrow f(a_0, a_0)$   
 $a_2 \rightarrow f(a_1, a_1)$   
 $a_i \rightarrow \dots 3 \text{ subterms} \dots$   
 $b_1 \rightarrow f(a_0, a_0)$   
 $b_2 \rightarrow f(a_1, a_1)$   
 $b_i \rightarrow \dots 3 \text{ subterms} \dots$

fully applied

triangular

# Complexity of Unification

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	$\phi$

## Space complexity

- Exponential
- Representation of unifier

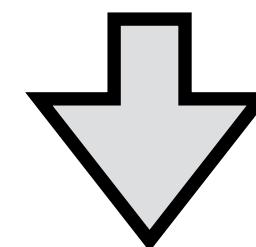
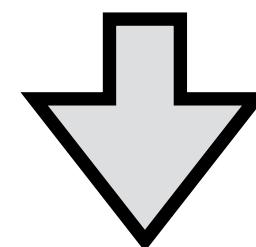
$$\begin{aligned} h(a_1, \dots, a_n, f(b_0, b_0), \dots, f(b_{n-1}, b_{n-1}), a_n) = \\ h(f(a_0, a_0), \dots, f(a_{n-1}, a_{n-1}), b_1, \dots, b_{n-1}, b_n) \end{aligned}$$

## Time complexity

- Exponential
- Recursive calls on terms

## Solution

- Union-Find algorithm
- Complexity growth can be considered constant



$a_1 \rightarrow f(a_0, a_0)$   
 $a_2 \rightarrow f(f(a_0, a_0), f(a_0, a_0))$   
 $a_i \rightarrow \dots 2^{i+1}-1 \text{ subterms} \dots$   
 $b_1 \rightarrow f(a_0, a_0)$   
 $b_2 \rightarrow f(f(a_0, a_0), f(a_0, a_0))$   
 $b_i \rightarrow \dots 2^{i+1}-1 \text{ subterms} \dots$

$a_1 \rightarrow f(a_0, a_0)$   
 $a_2 \rightarrow f(a_1, a_1)$   
 $a_i \rightarrow \dots 3 \text{ subterms} \dots$   
 $b_1 \rightarrow f(a_0, a_0)$   
 $b_2 \rightarrow f(a_1, a_1)$   
 $b_i \rightarrow \dots 3 \text{ subterms} \dots$

fully applied

triangular

# Set Representatives

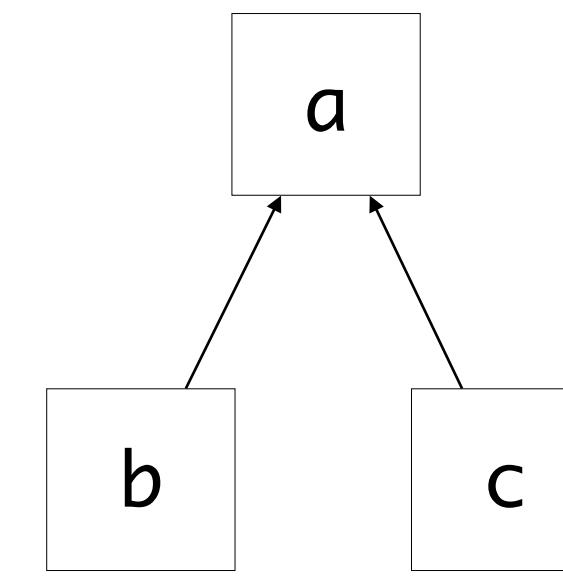


# Set Representatives

$a == b$   
 $c == a$

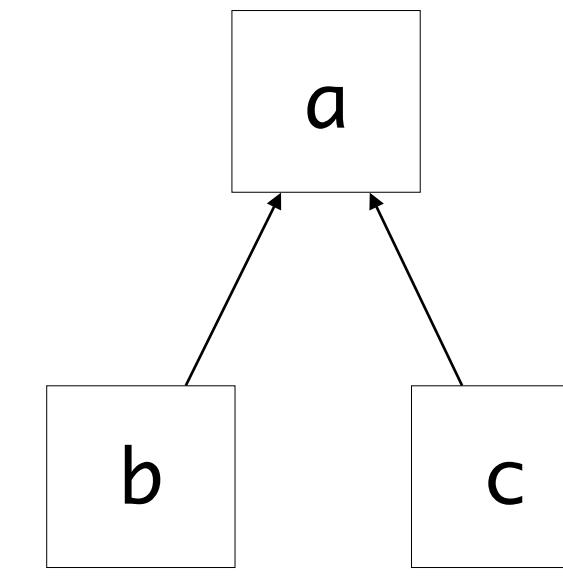
# Set Representatives

$a == b$   
 $c == a$



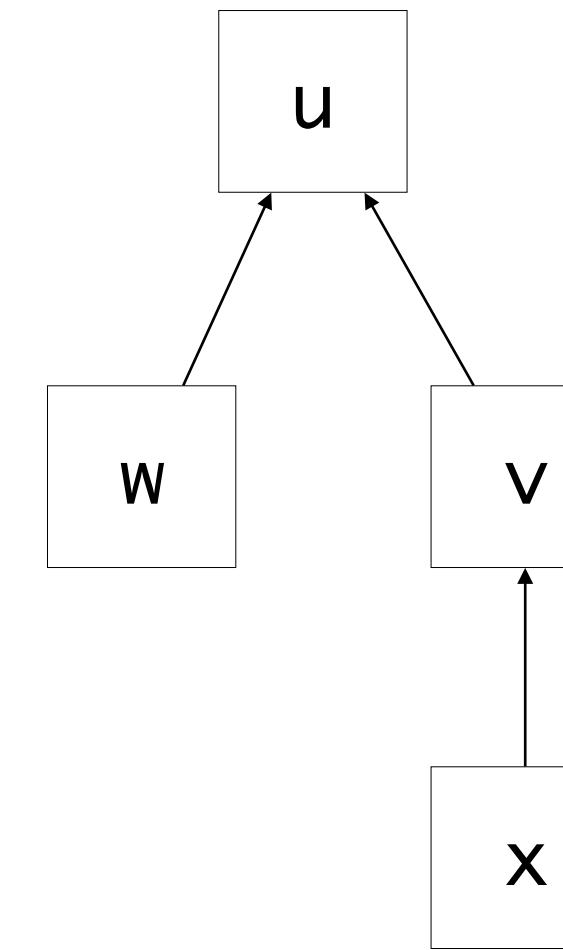
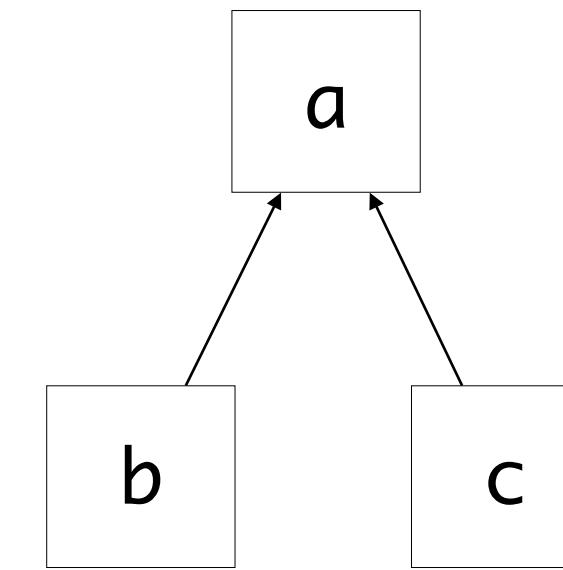
# Set Representatives

$a == b$   
 $c == a$   
 $u == w$   
 $v == u$   
 $x == v$

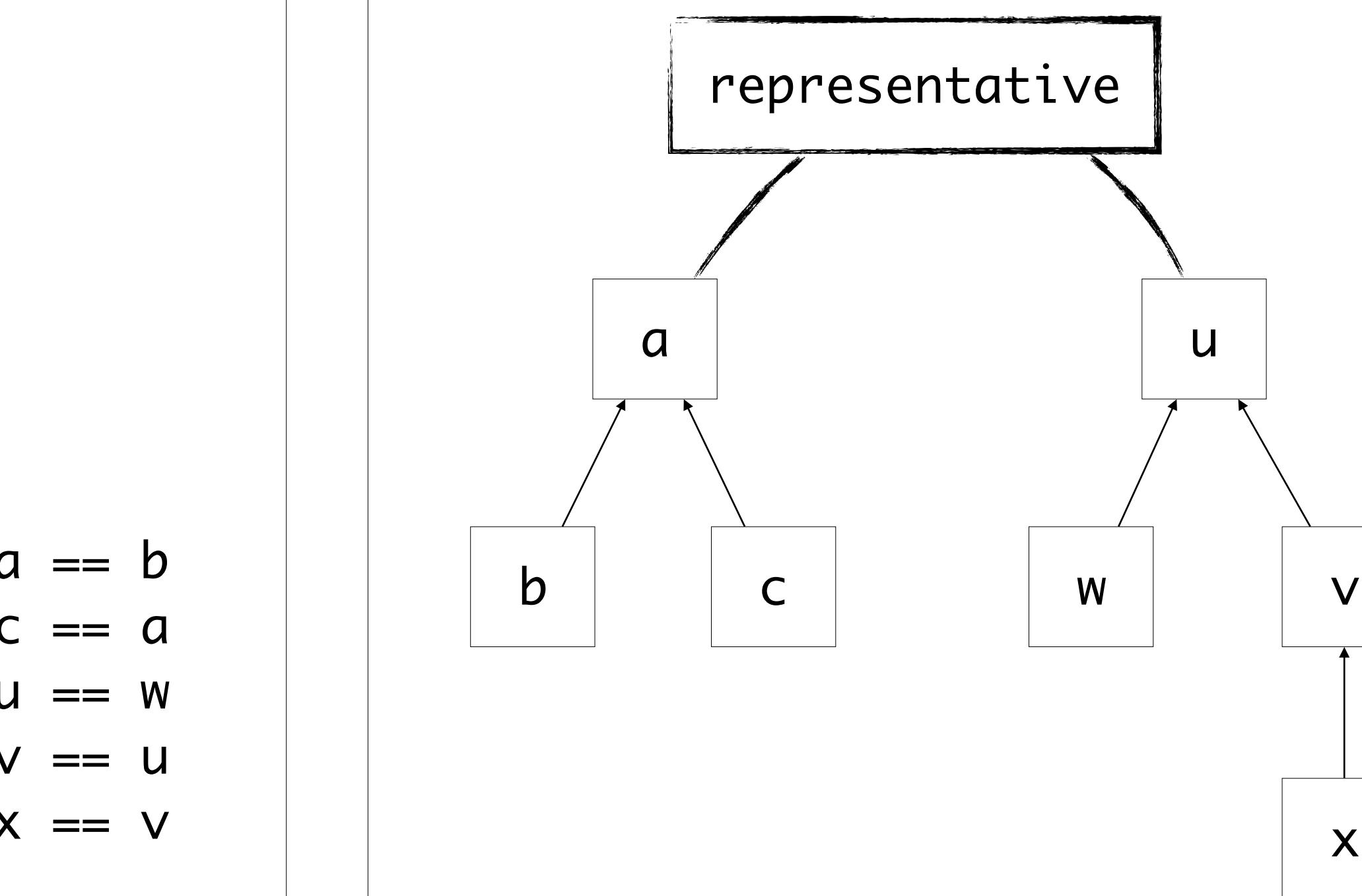


# Set Representatives

$a == b$   
 $c == a$   
 $u == w$   
 $v == u$   
 $x == v$



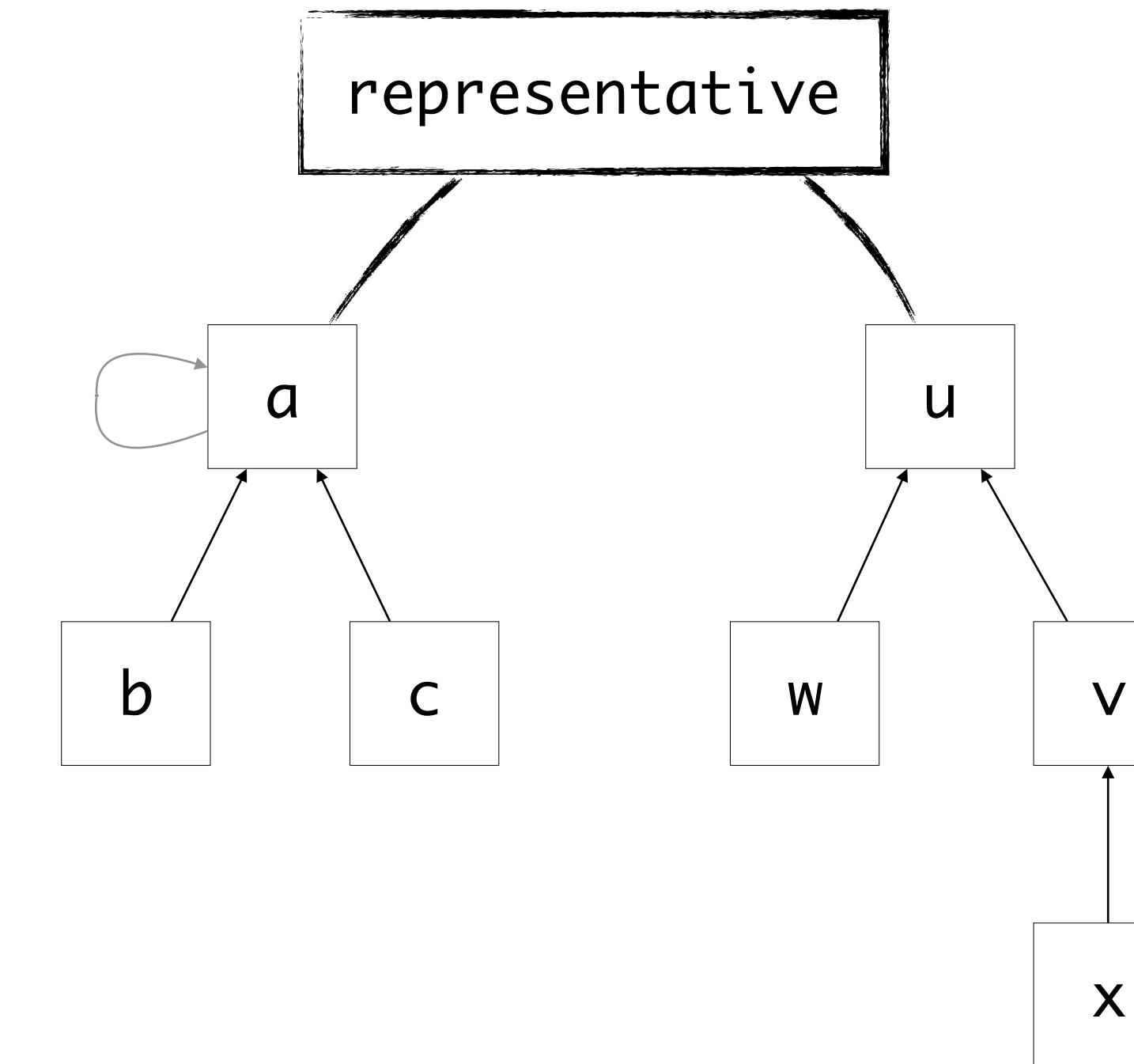
# Set Representatives



$a == b$   
 $c == a$   
 $u == w$   
 $v == u$   
 $x == v$

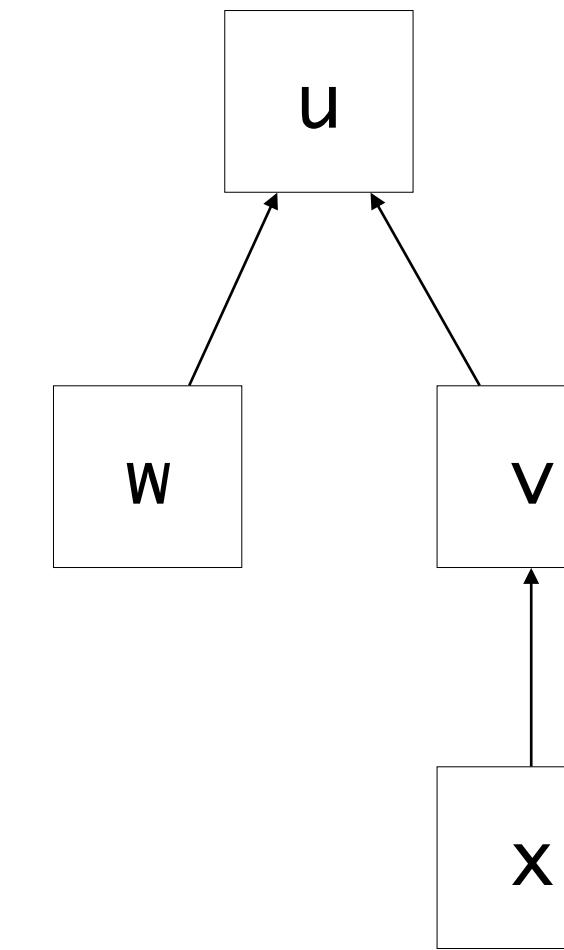
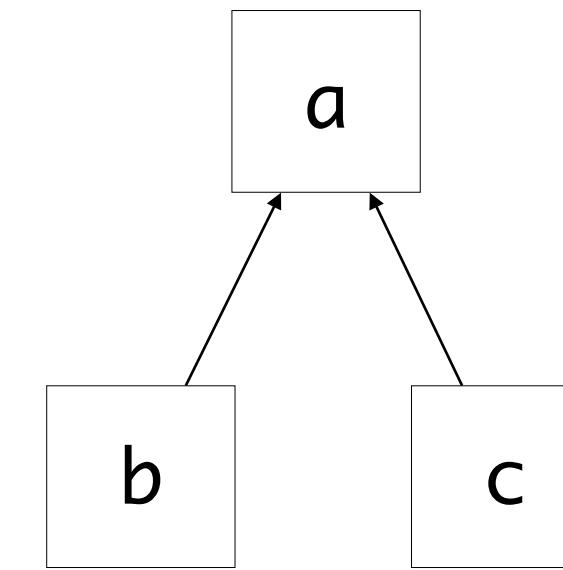
# Set Representatives

$a == b$   
 $c == a$   
 $u == w$   
 $v == u$   
 $x == v$



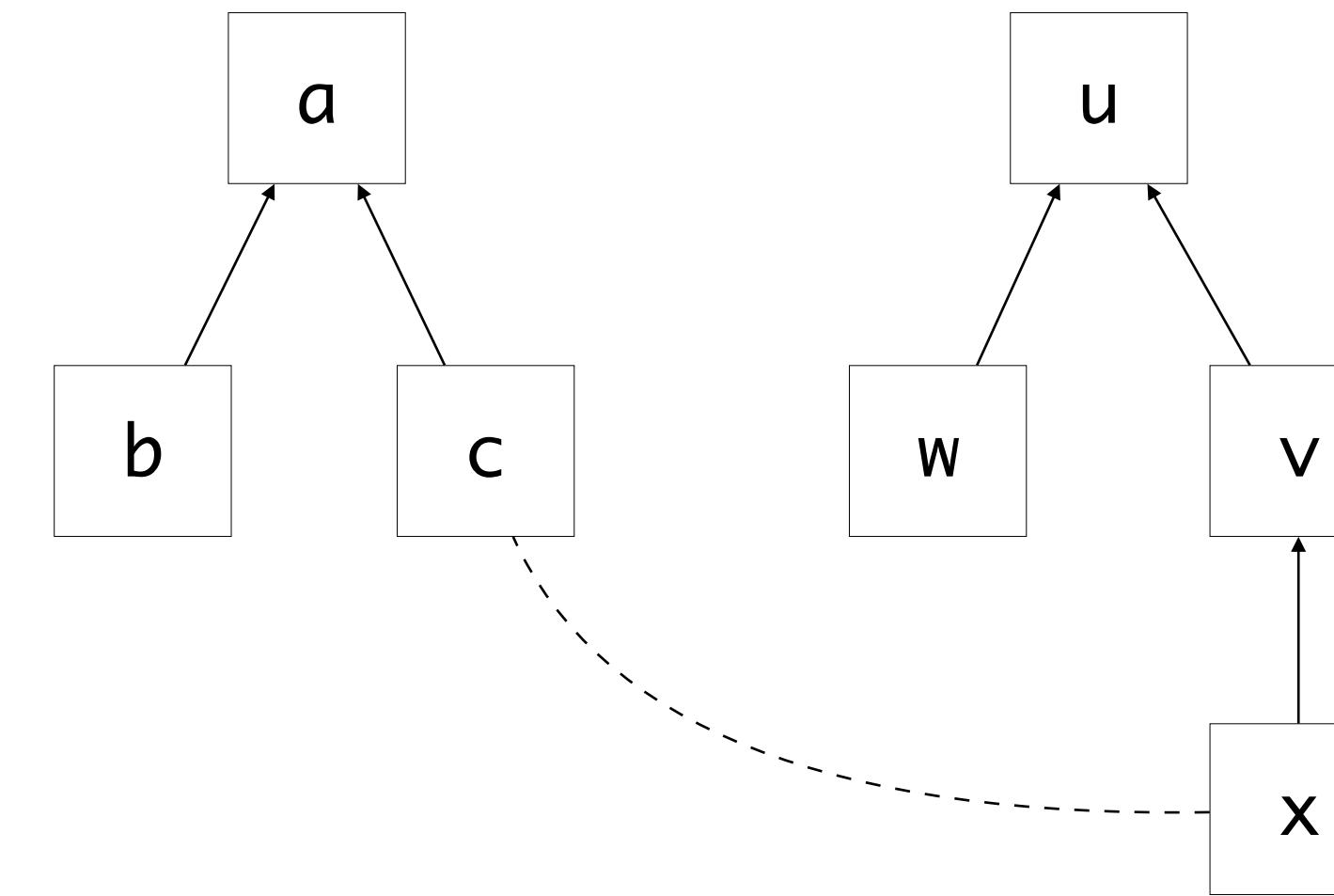
# Set Representatives

$a == b$   
 $c == a$   
 $u == w$   
 $v == u$   
 $x == v$



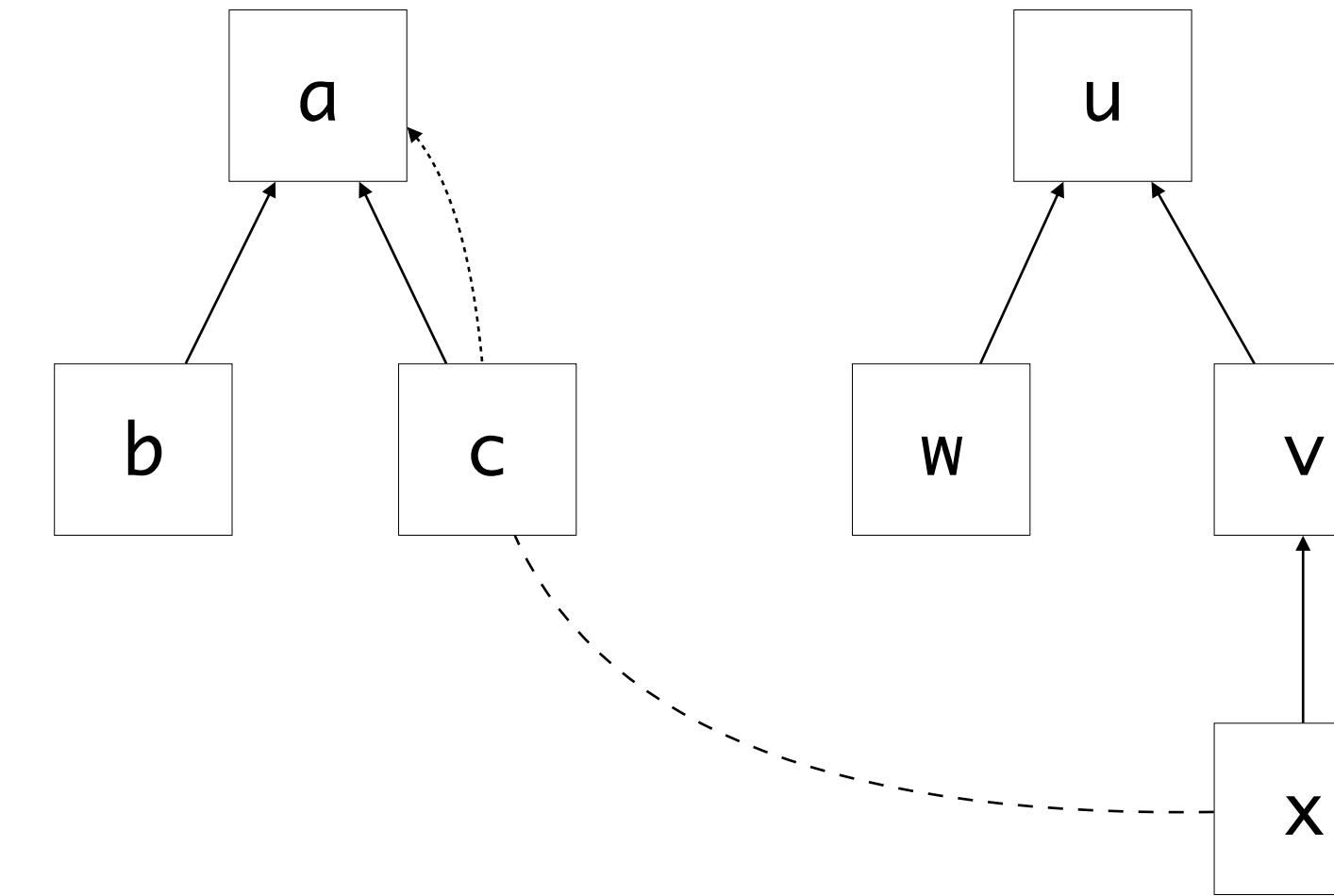
# Set Representatives

$a == b$   
 $c == a$   
 $u == w$   
 $v == u$   
 $x == v$   
 $x == c$



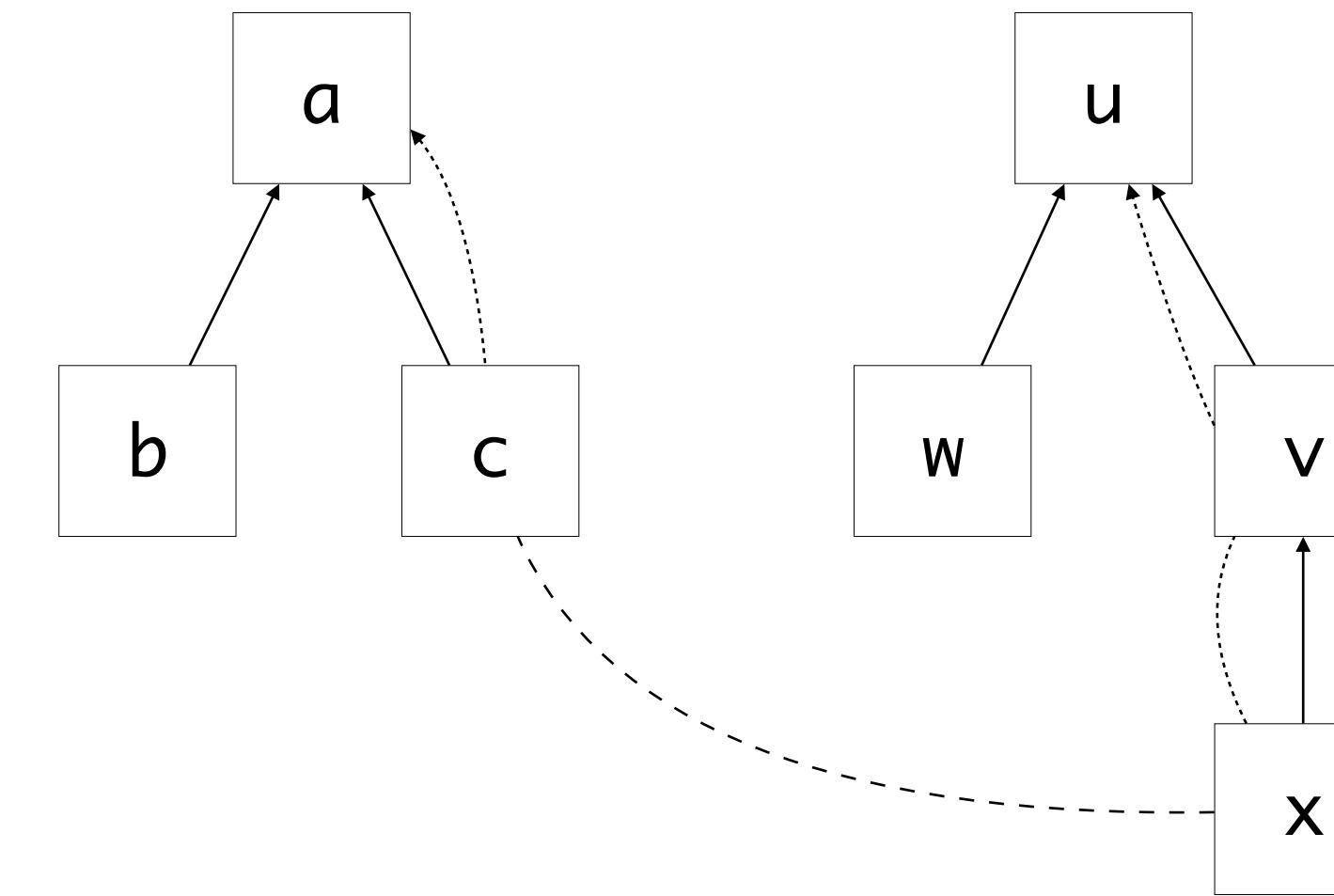
# Set Representatives

$a == b$   
 $c == a$   
 $u == w$   
 $v == u$   
 $x == v$   
 $x == c$



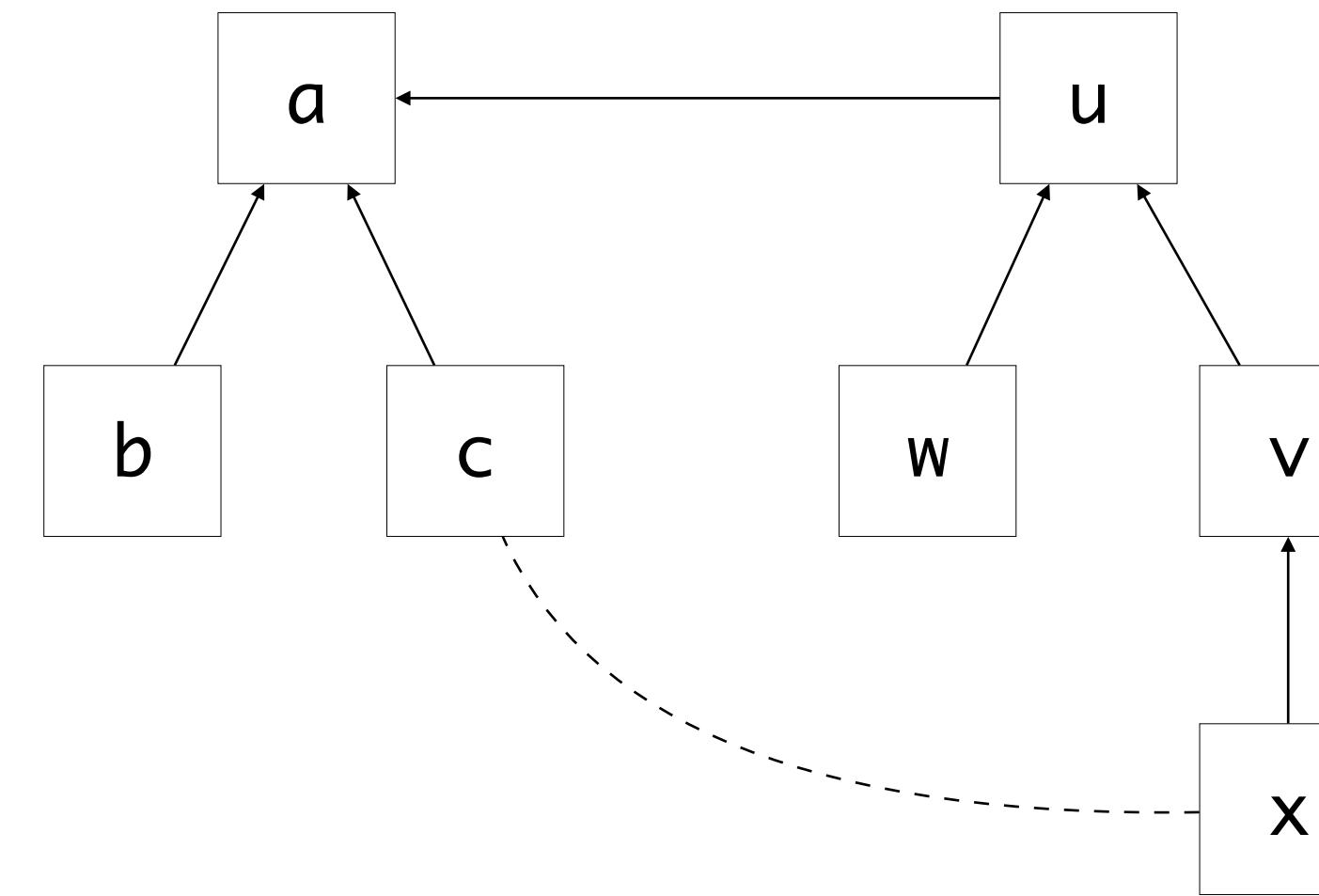
# Set Representatives

```
a == b  
c == a  
u == w  
v == u  
x == v  
x == c
```



# Set Representatives

$a == b$   
 $c == a$   
 $u == w$   
 $v == u$   
 $x == v$   
 $x == c$



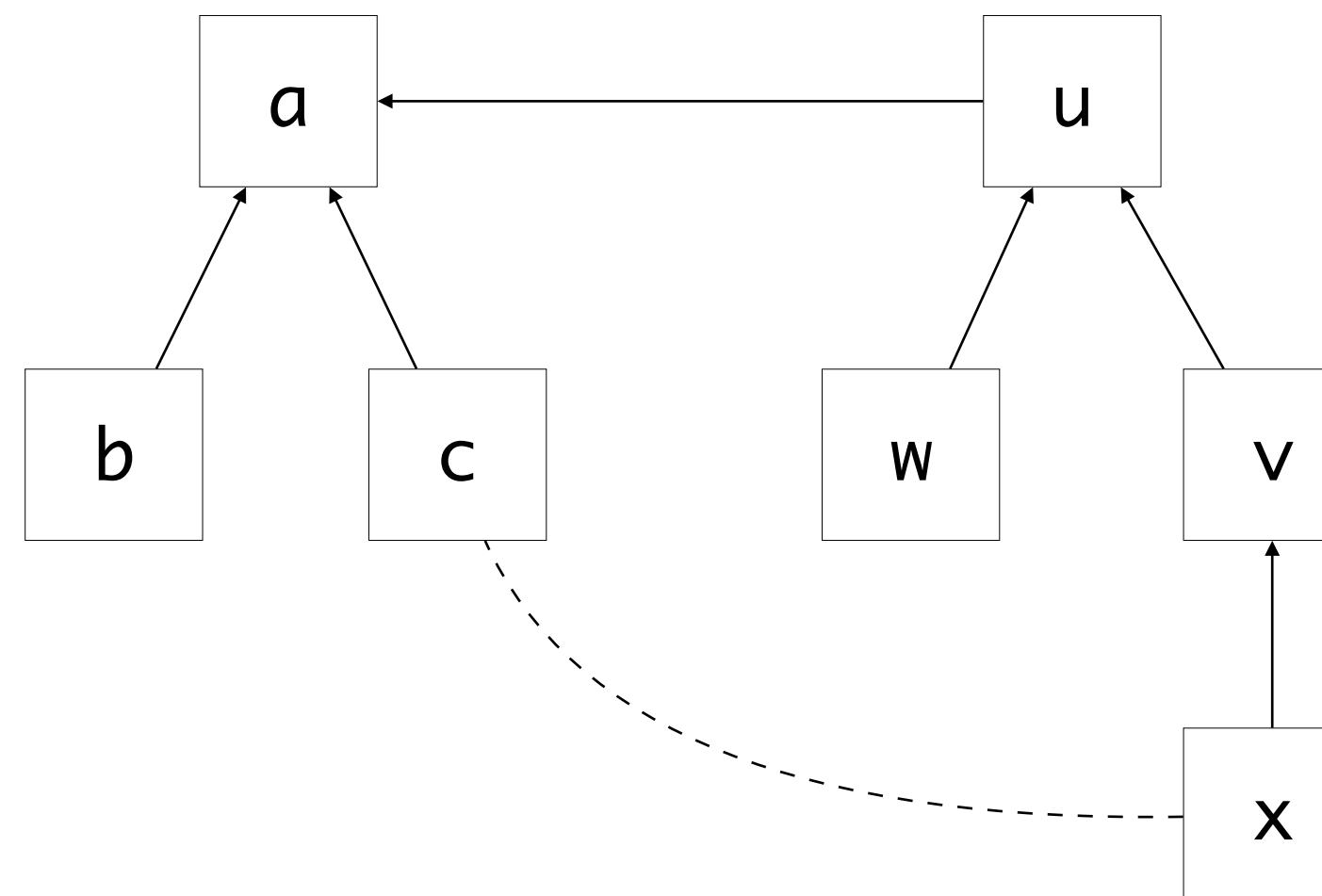
# Set Representatives

```
FIND(a):  
    b := rep(a)  
    if b == a:  
        return a  
    else  
        return FIND(b)
```

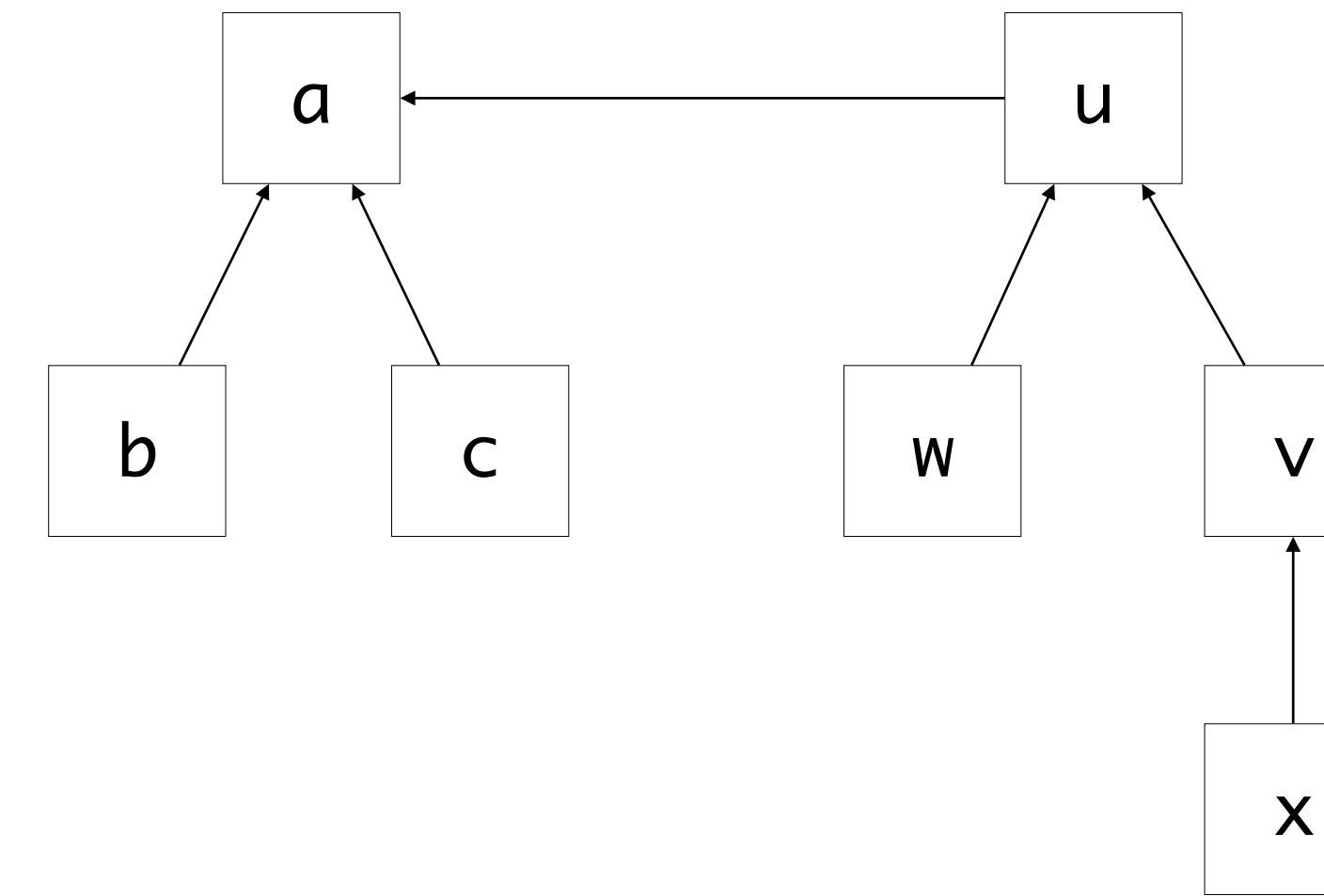
```
UNION(a1,a2):  
    b1 := FIND(a1)  
    b2 := FIND(a2)  
    LINK(b1,b2)
```

```
LINK(a1,a2):  
    rep(a1) := a2
```

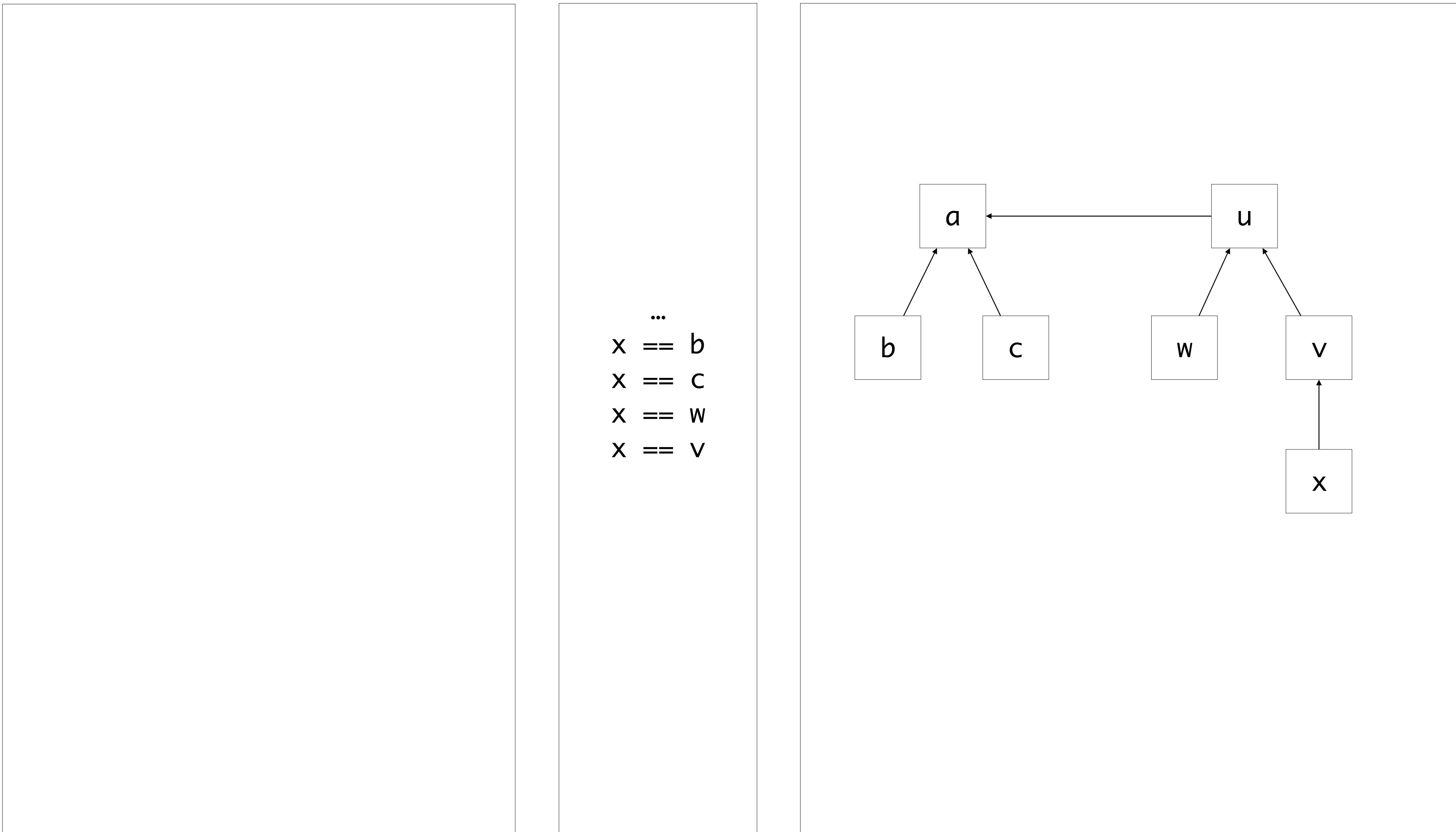
```
a == b  
c == a  
u == w  
v == u  
x == v  
x == c
```



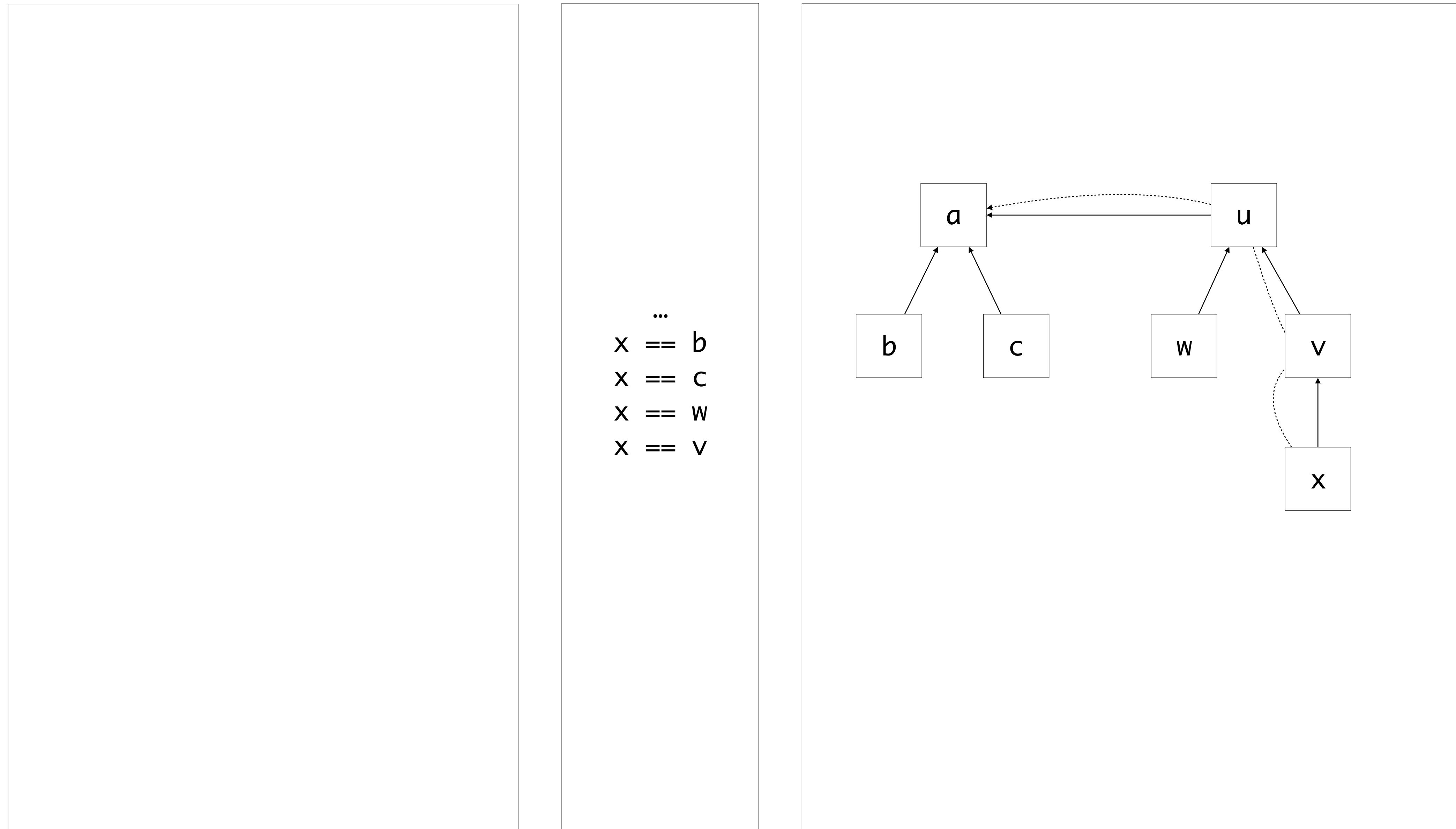
# Path Compression



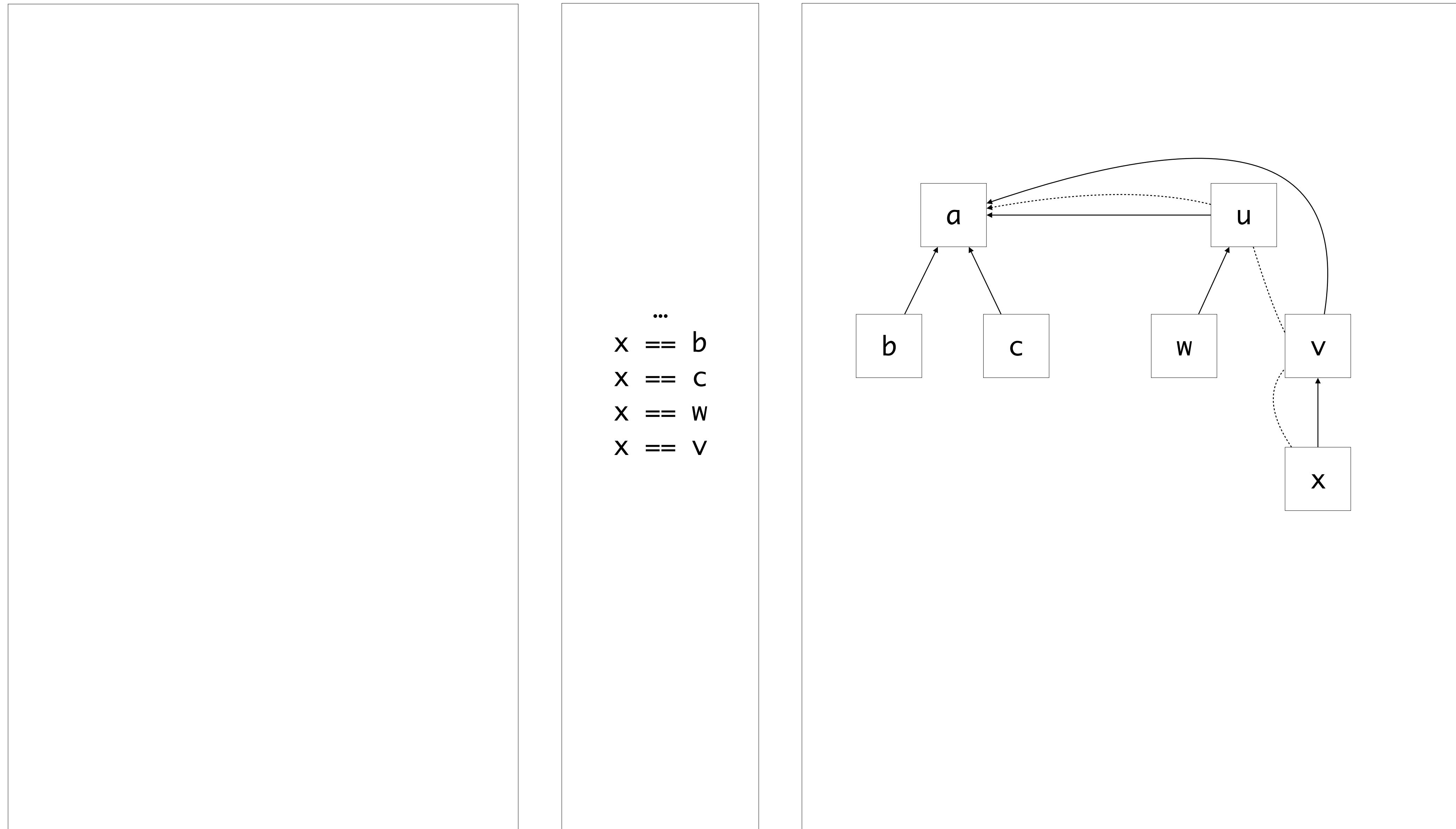
# Path Compression



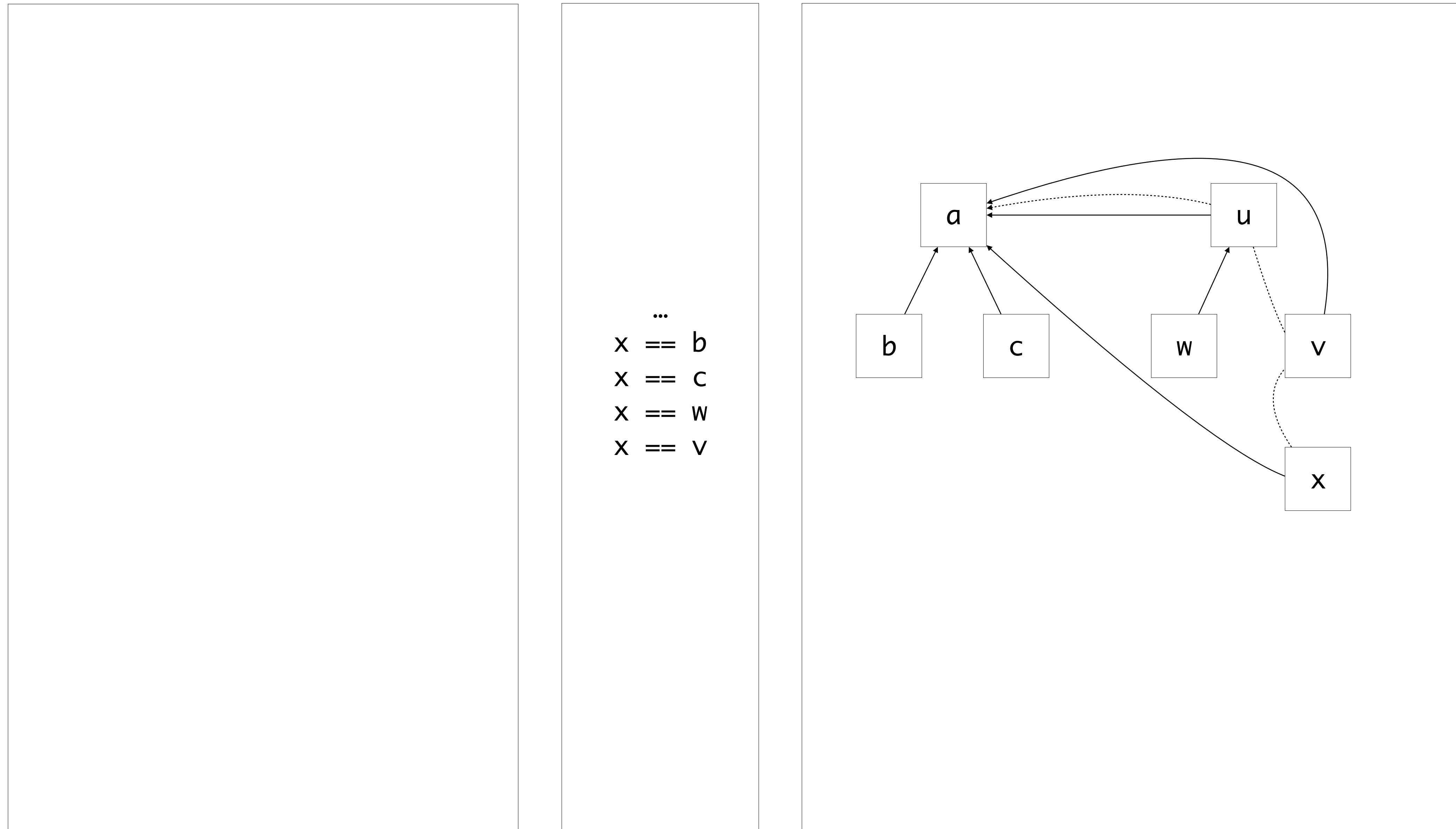
# Path Compression



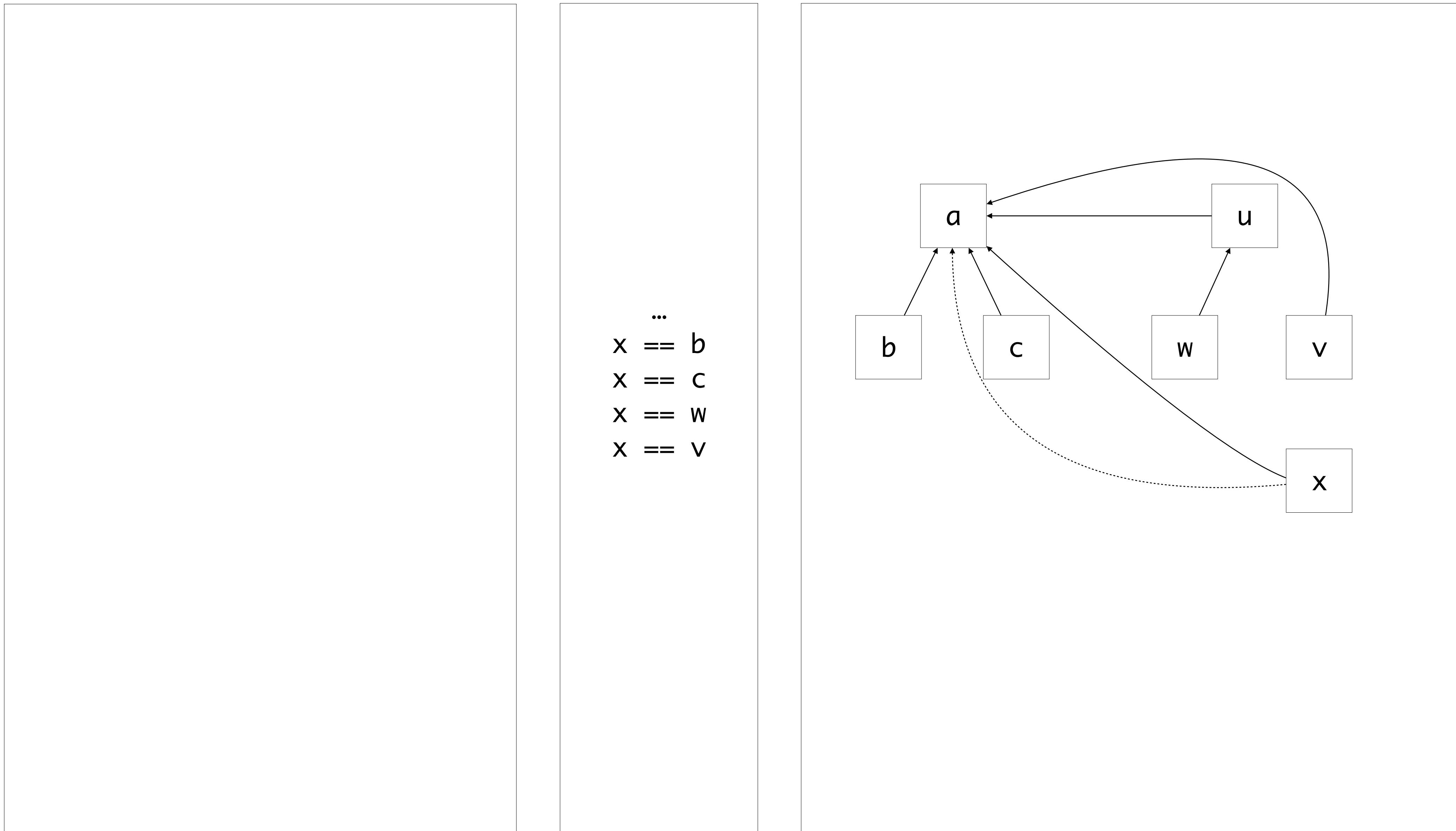
# Path Compression



# Path Compression



# Path Compression



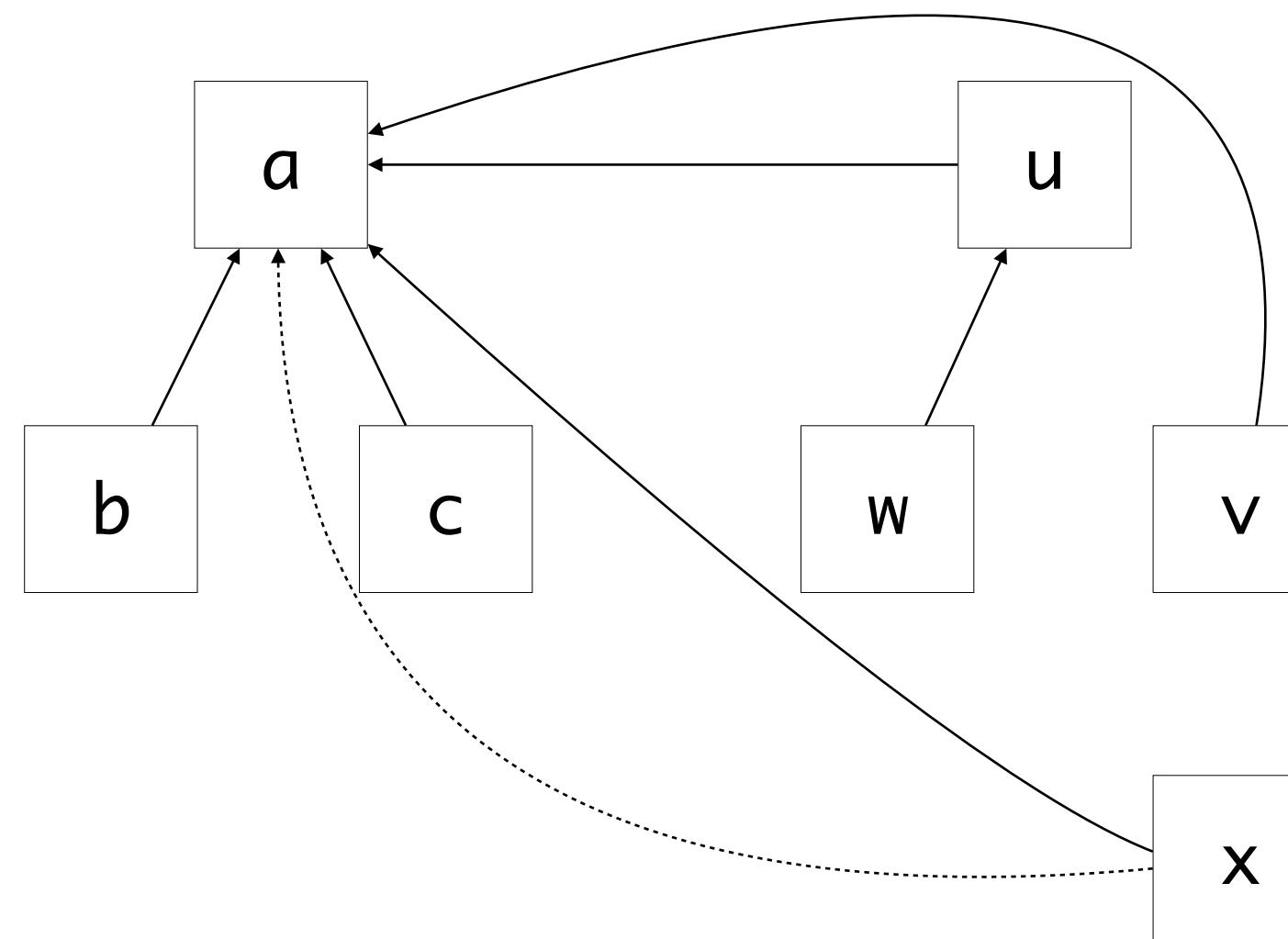
# Path Compression

```
FIND(a):  
    b := rep(a)  
    if b == a:  
        return a  
    else:  
        b := FIND(b)  
        rep(a) := b  
        return b
```

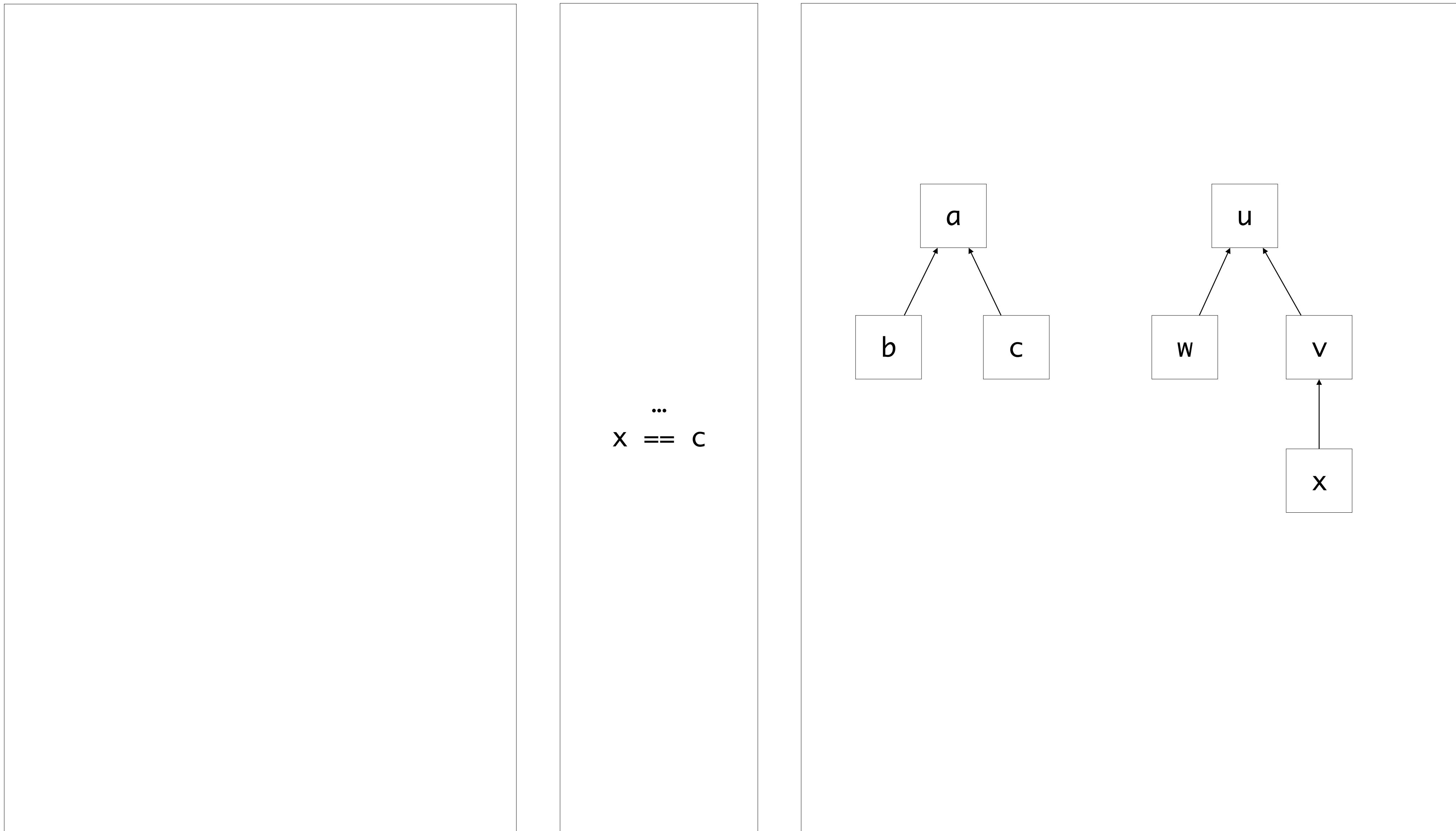
```
UNION(a1,a2):  
    b1 := FIND(a1)  
    b2 := FIND(a2)  
    LINK(b1,b2)
```

```
LINK(a1,a2):  
    rep(a1) := a2
```

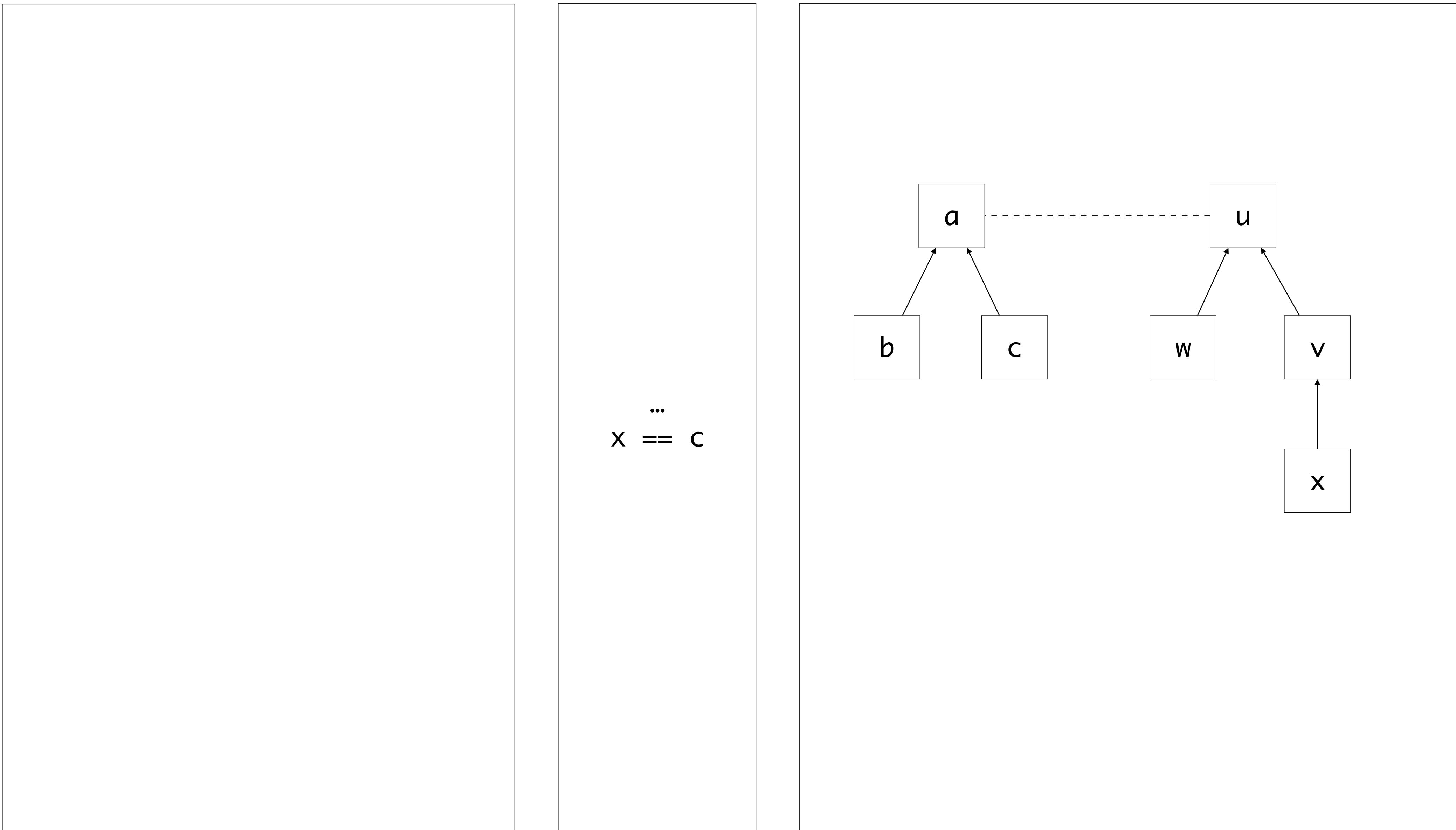
...  
x == b  
x == c  
x == w  
x == v



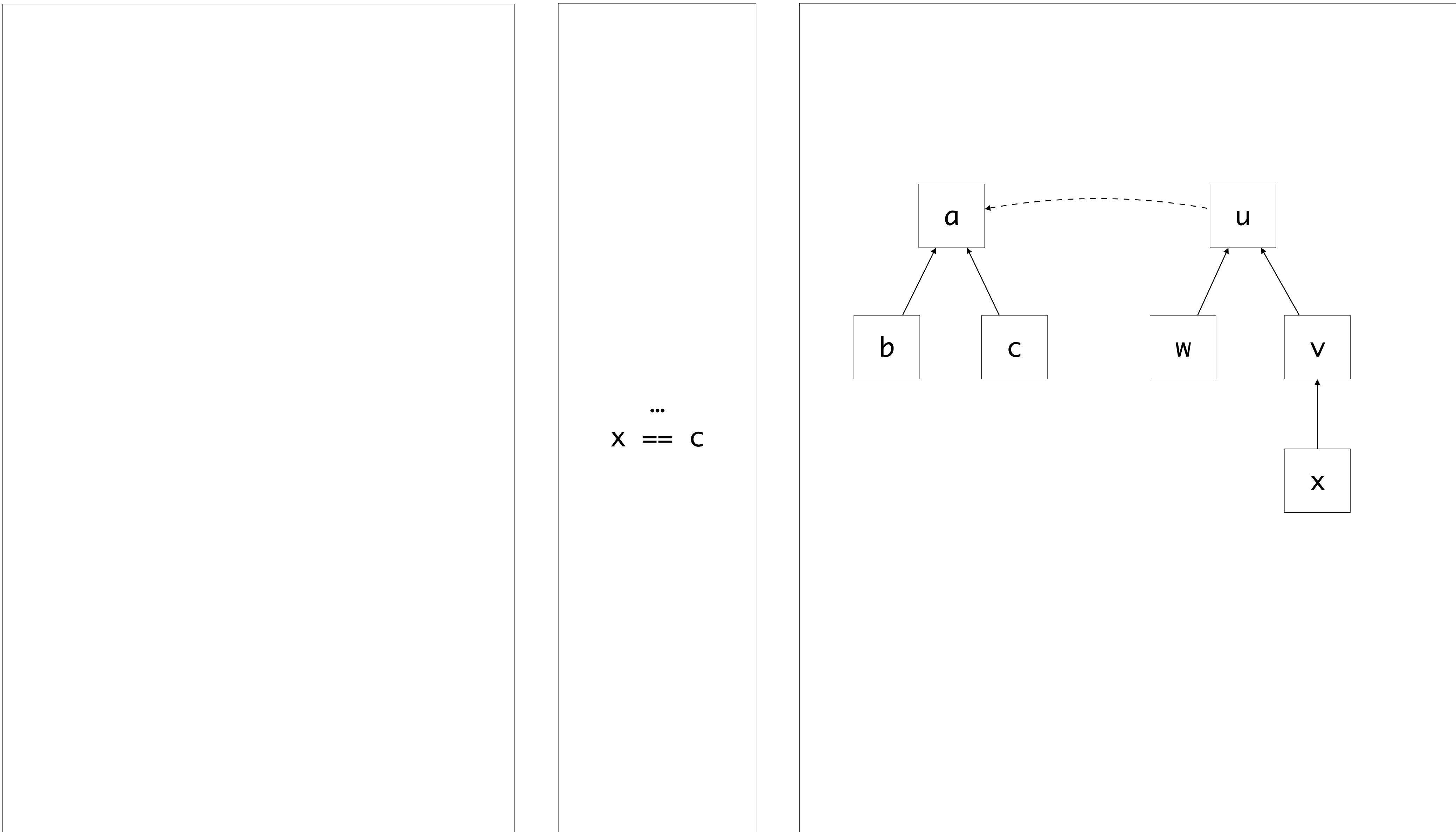
# Tree Balancing



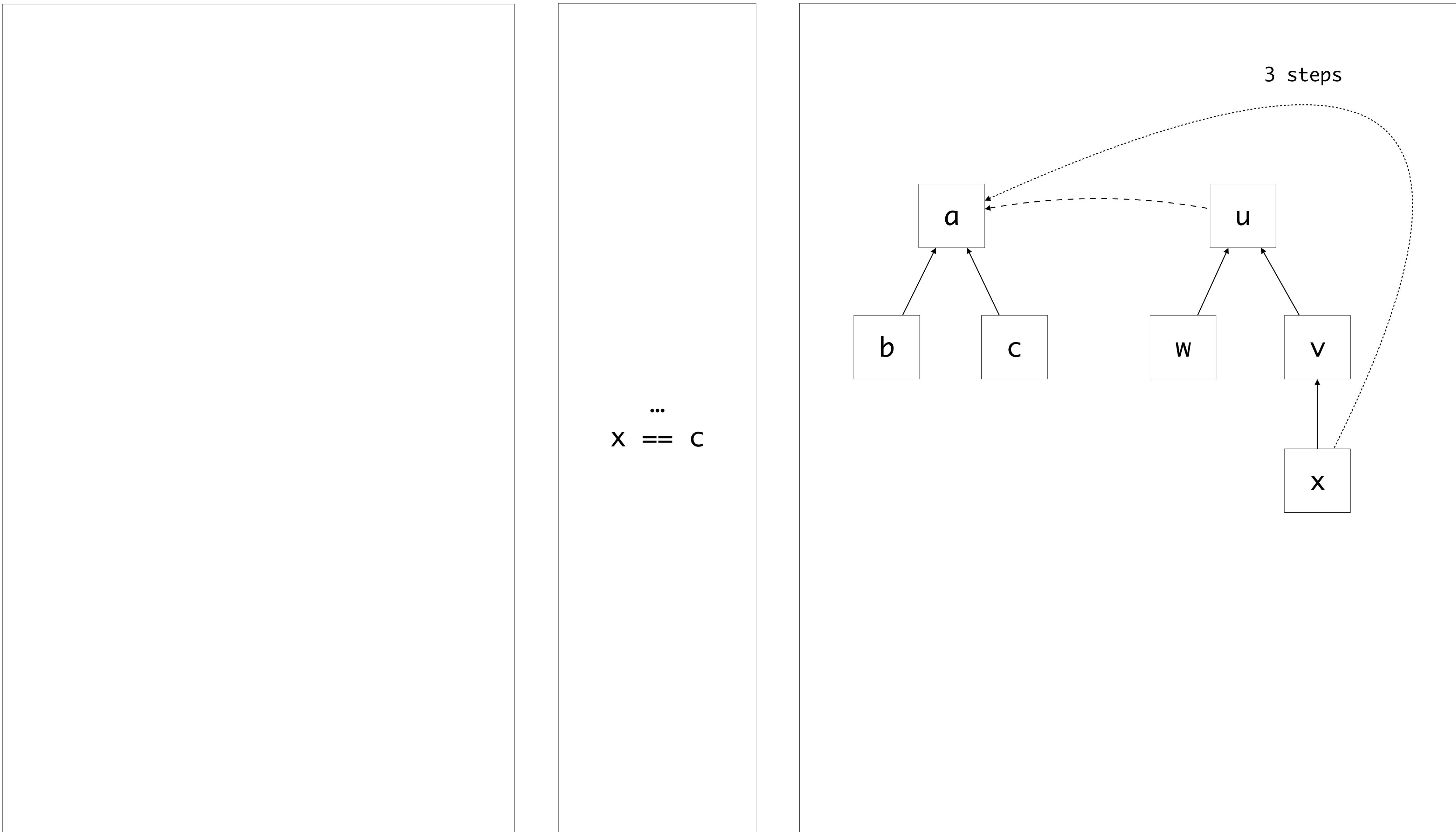
# Tree Balancing



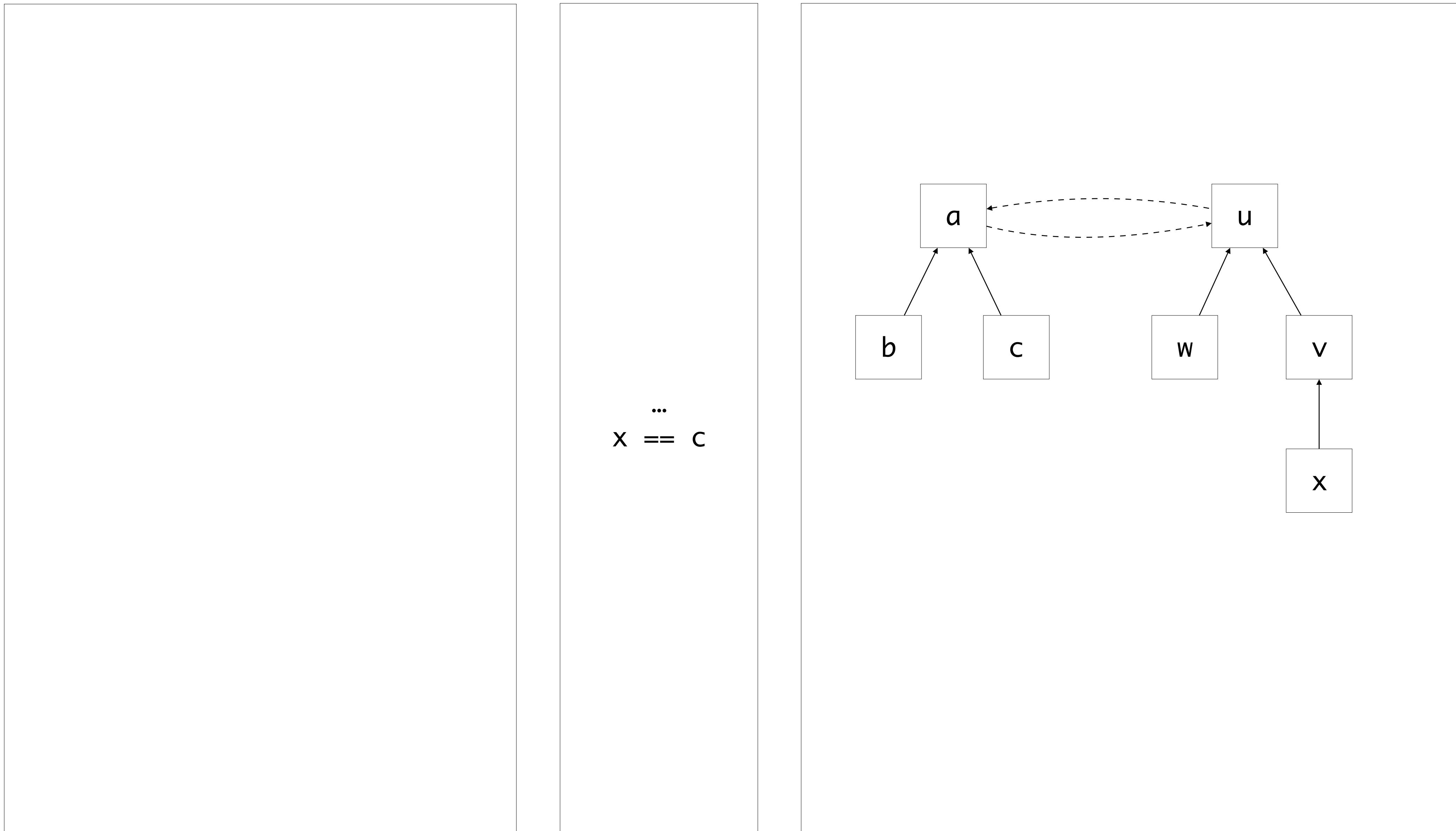
# Tree Balancing



# Tree Balancing

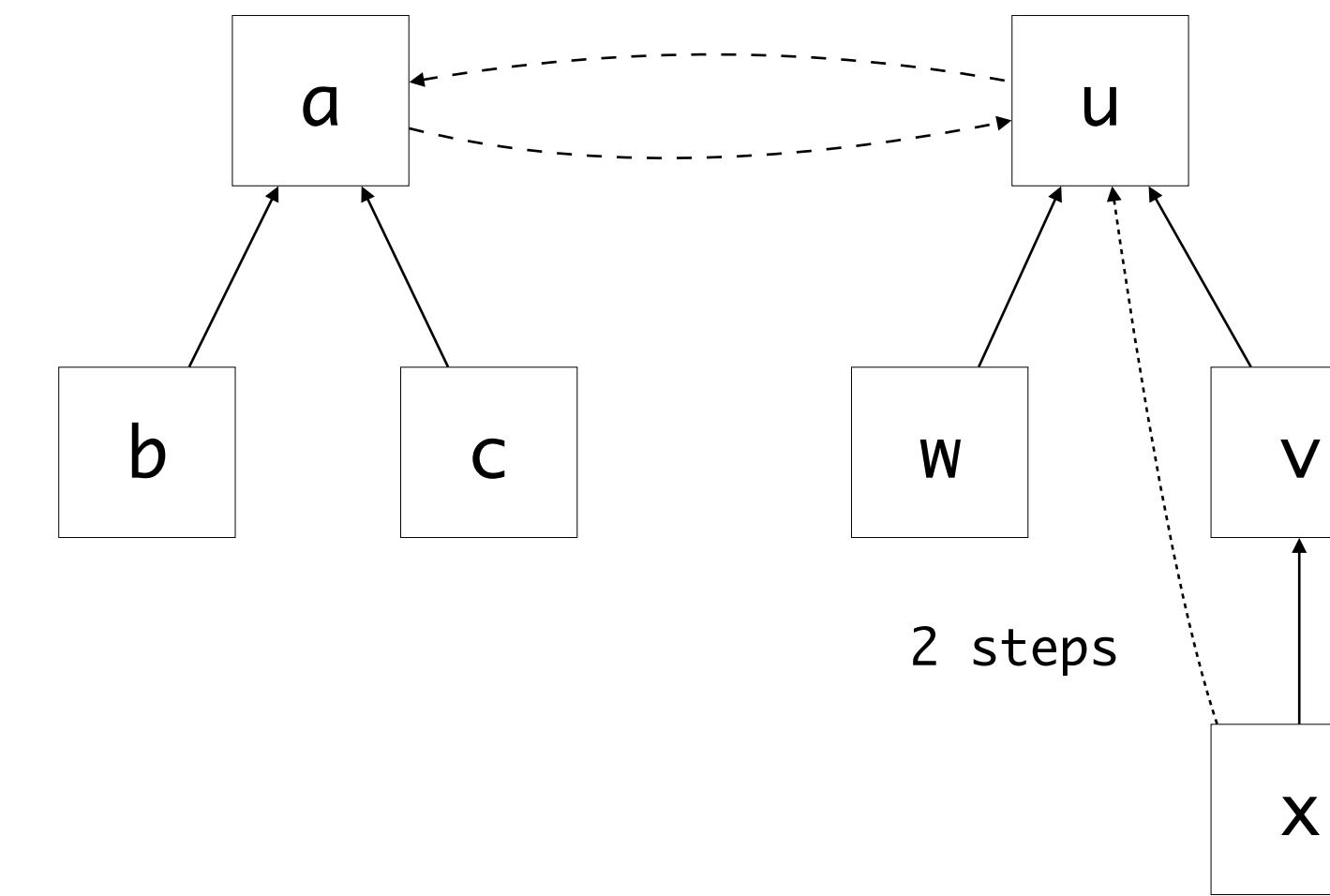


# Tree Balancing

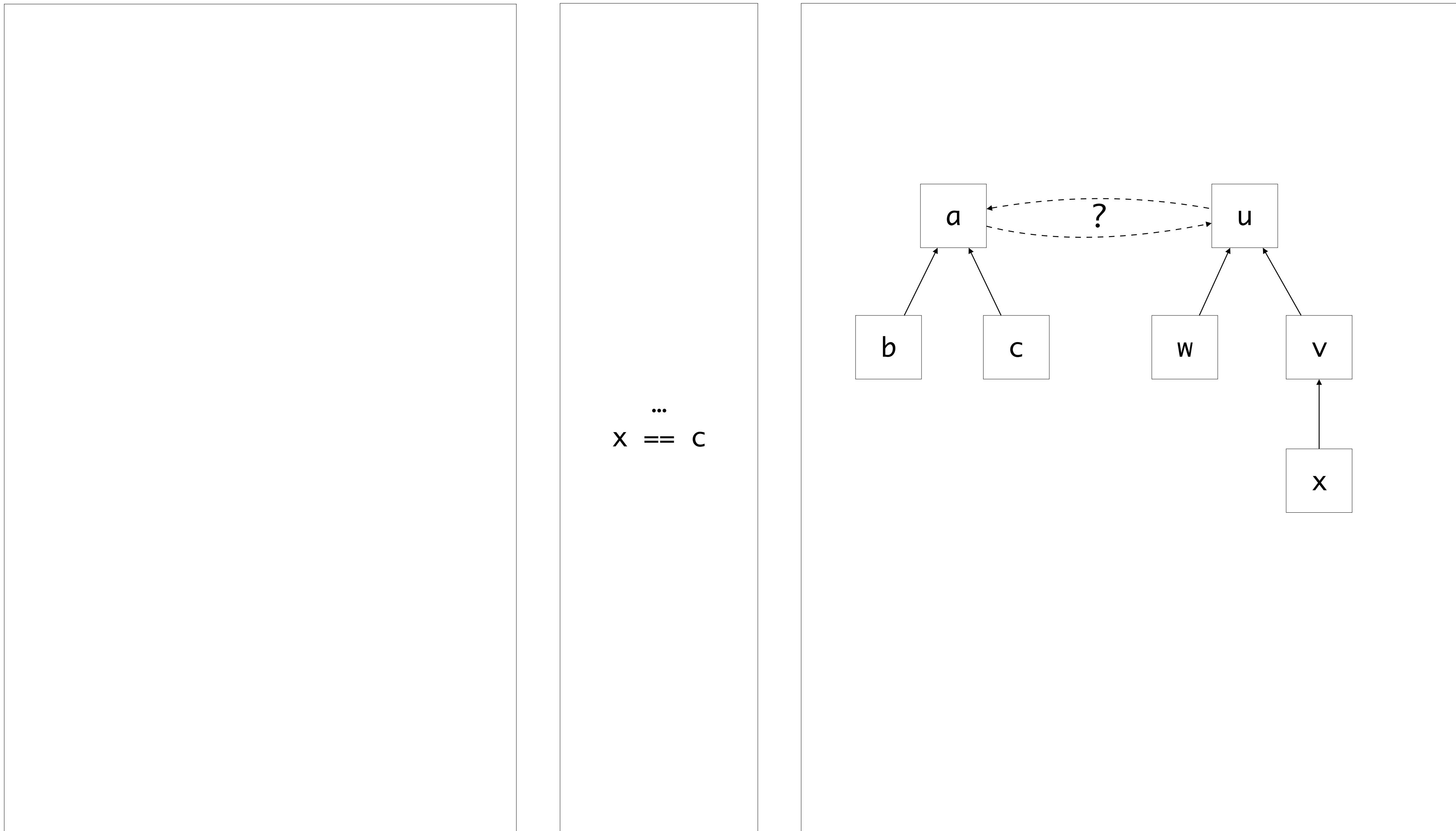


# Tree Balancing

$x == c$   
...

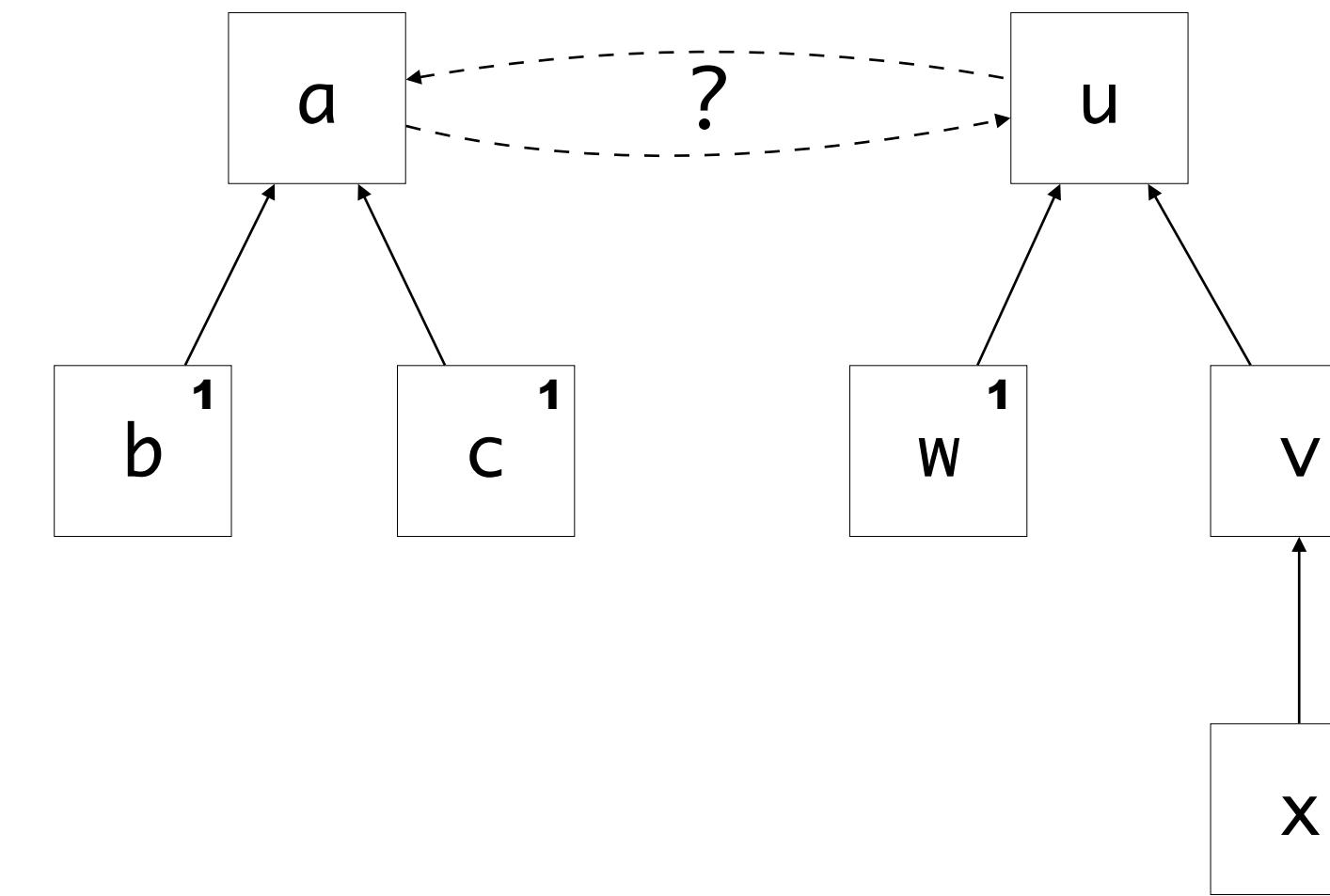


# Tree Balancing



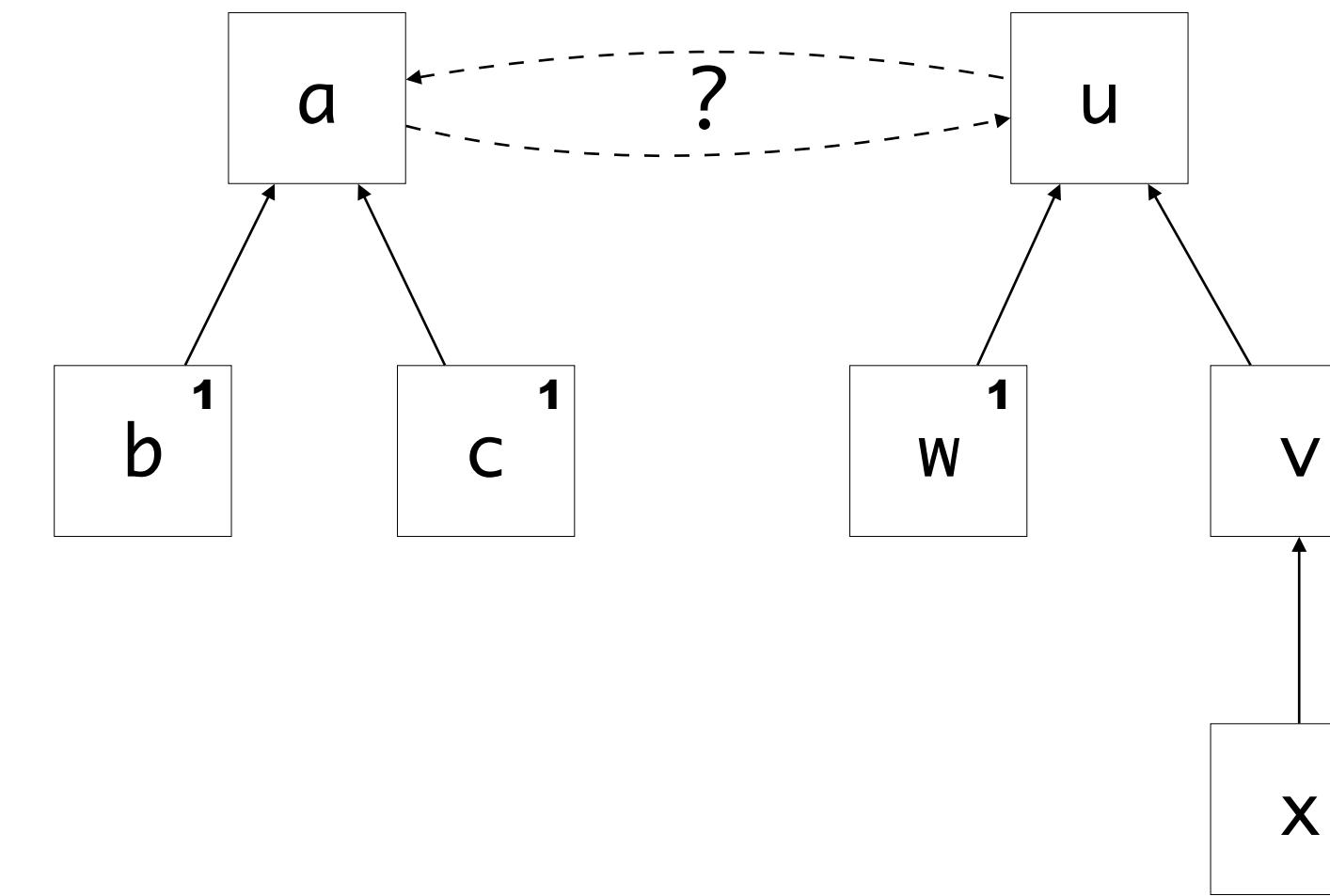
# Tree Balancing

$x == c$   
...



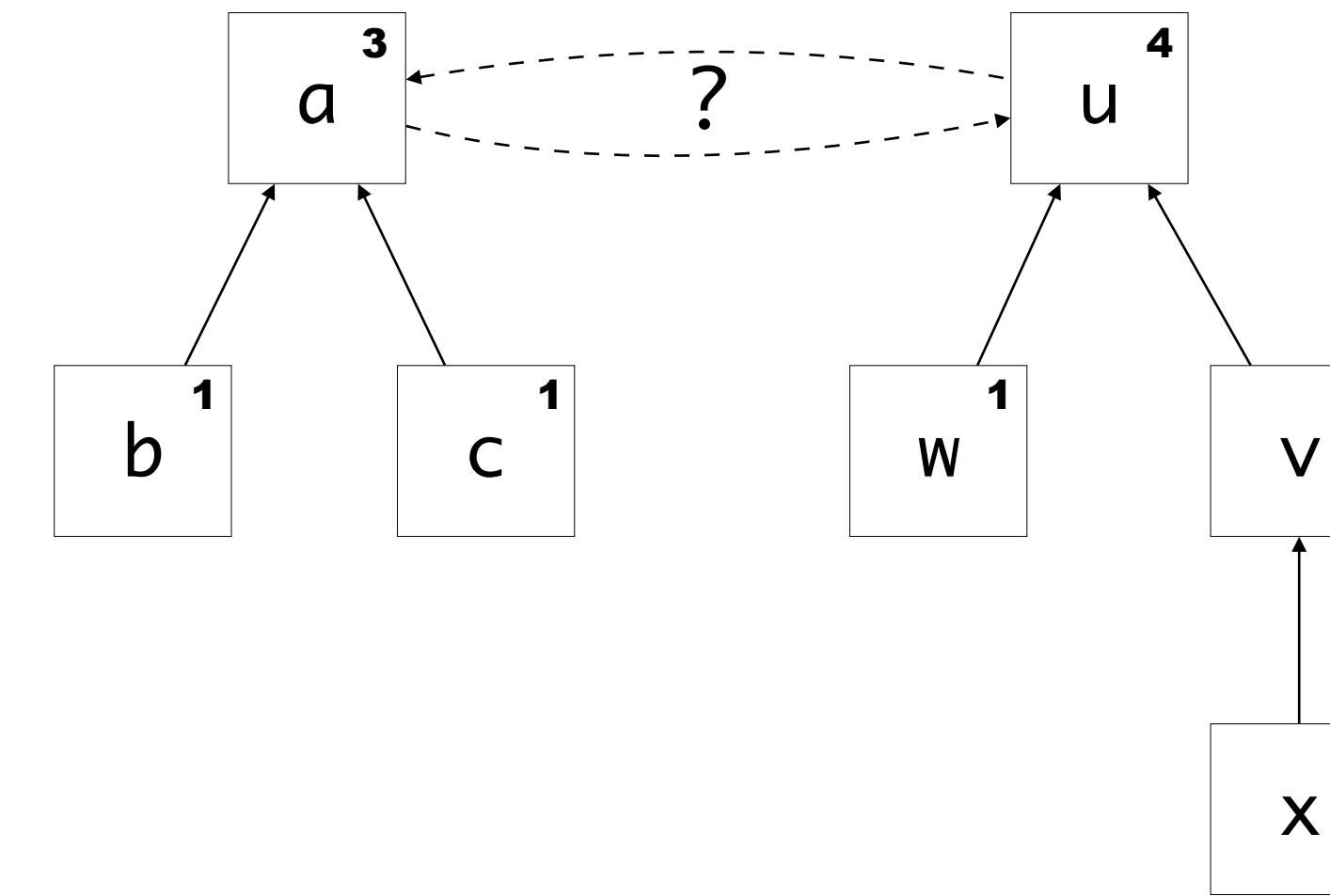
# Tree Balancing

$x == c$   
...

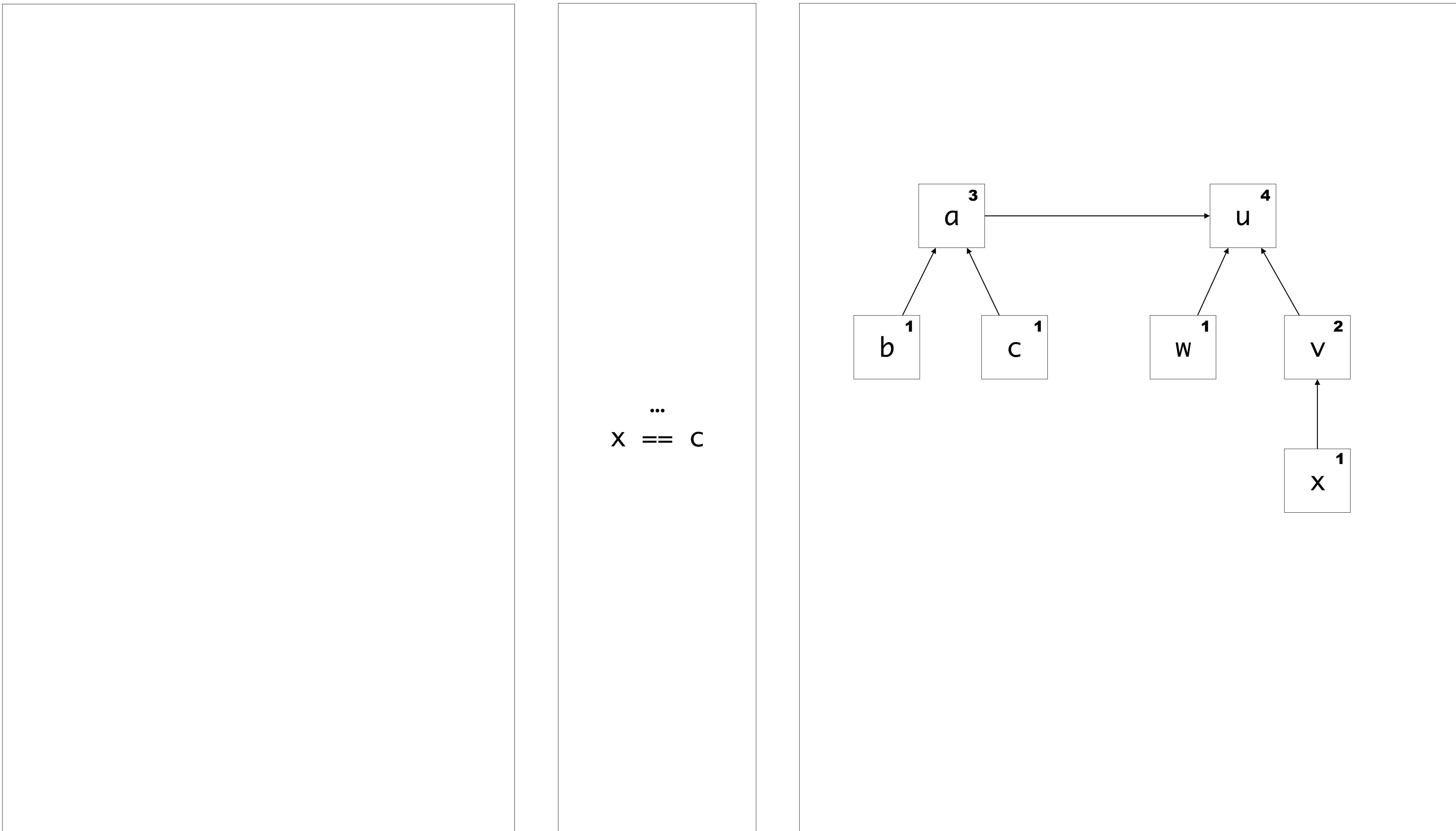


# Tree Balancing

$x == c$   
...



# Tree Balancing



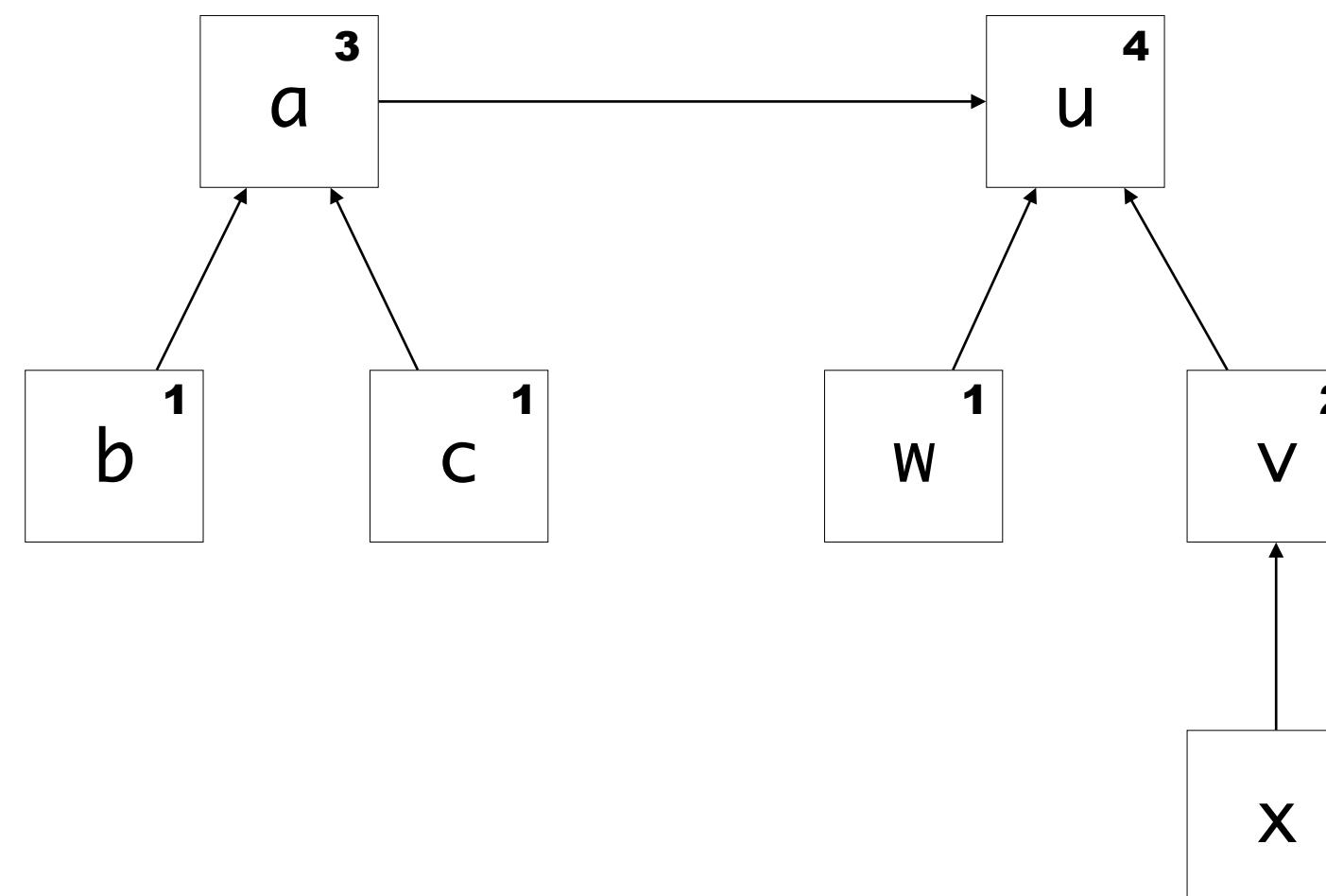
# Tree Balancing

```
FIND(a):
    b := rep(a)
    if b == a:
        return a
    else
        b := FIND(b)
        rep(a) := b
        return b
```

```
UNION(a1,a2):
    b1 := FIND(a1)
    b2 := FIND(a2)
    LINK(b1,b2)
```

```
LINK(a1,a2):
    if size(a2) > size(a1):
        rep(a1) := a2
        size(a2) += size(a1)
    else:
        rep(a2) := a1
        size(a1) += size(a2)
```

...  
x == c

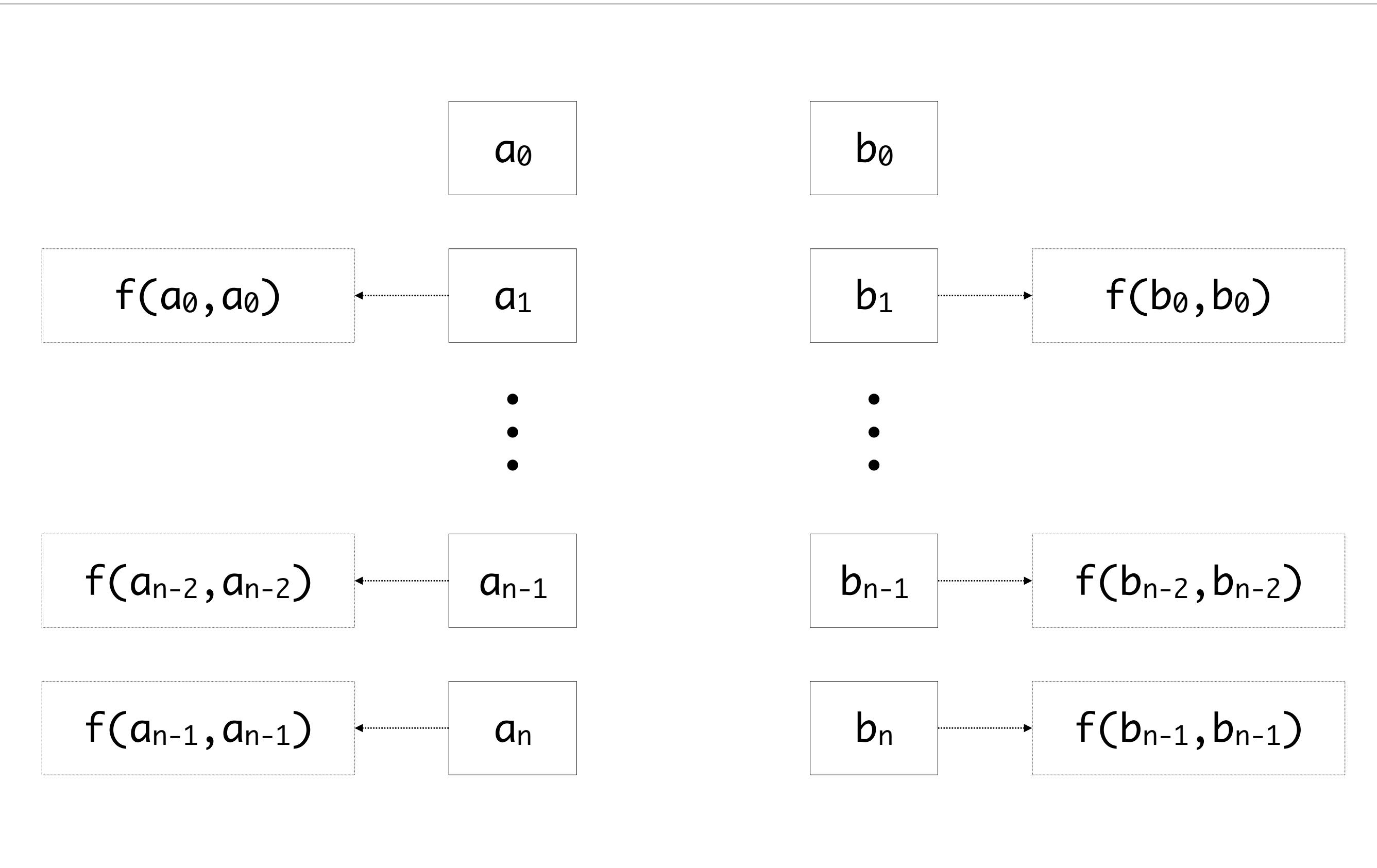


# The Complex Case

$$h(a_1, \dots, a_n, f(b_0, b_0), \dots, f(b_{n-1}, b_{n-1}), a_n) == \\ h(f(a_0, a_0), \dots, f(a_{n-1}, a_{n-1}), b_1, \dots, b_{n-1}, b_n)$$

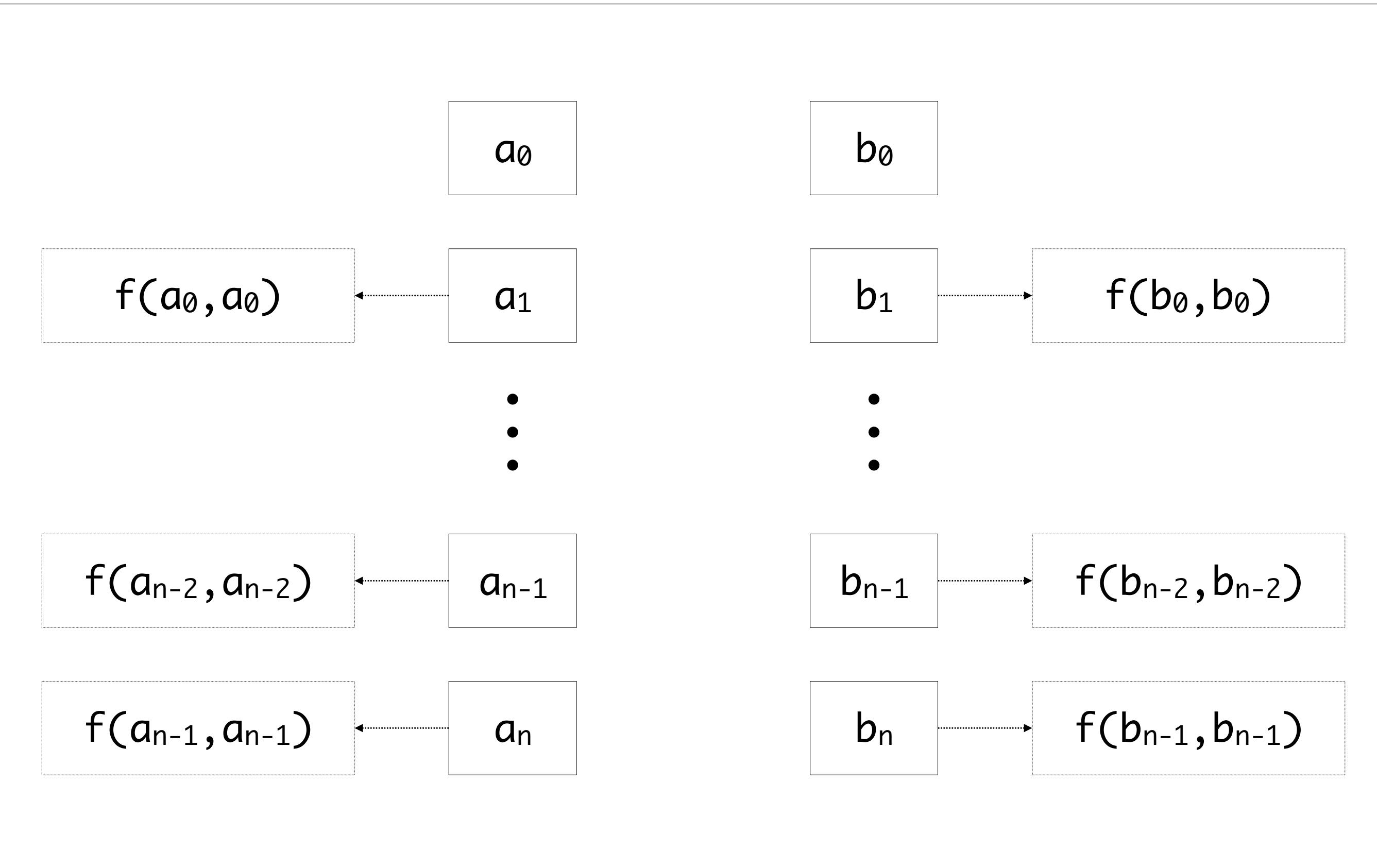
# The Complex Case

$$h(a_1, \dots, a_n, f(b_0, b_0), \dots, f(b_{n-1}, b_{n-1}), a_n) == \\ h(f(a_0, a_0), \dots, f(a_{n-1}, a_{n-1}), b_1, \dots, b_{n-1}, b_n)$$



# The Complex Case

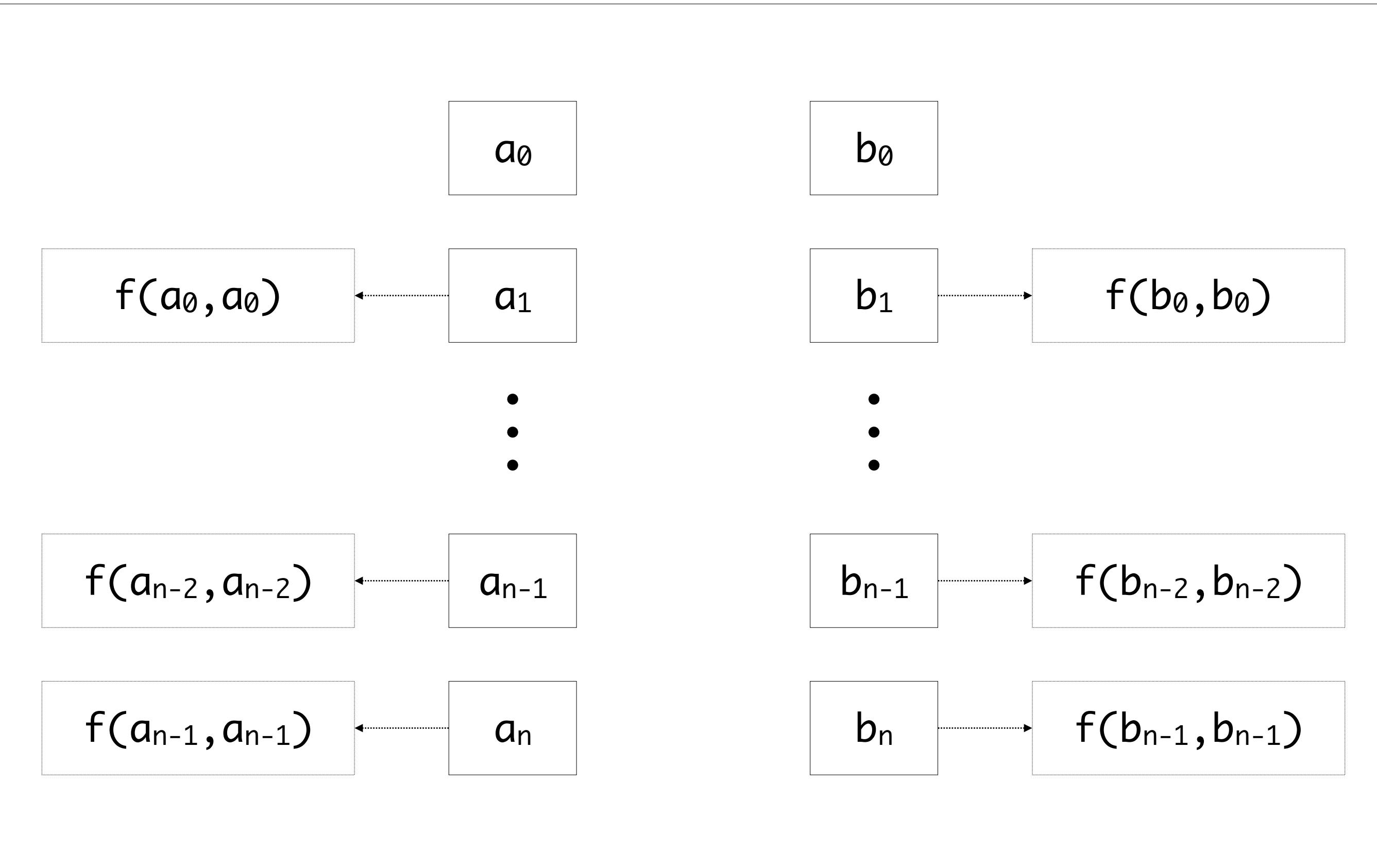
$$h(a_1, \dots, a_n, f(b_0, b_0), \dots, f(b_{n-1}, b_{n-1}), a_n) == \\ h(f(a_0, a_0), \dots, f(a_{n-1}, a_{n-1}), b_1, \dots, b_{n-1}, b_n)$$



$$a_n == b_n$$

# The Complex Case

$$h(a_1, \dots, a_n, f(b_0, b_0), \dots, f(b_{n-1}, b_{n-1}), a_n) == \\ h(f(a_0, a_0), \dots, f(a_{n-1}, a_{n-1}), b_1, \dots, b_{n-1}, b_n)$$

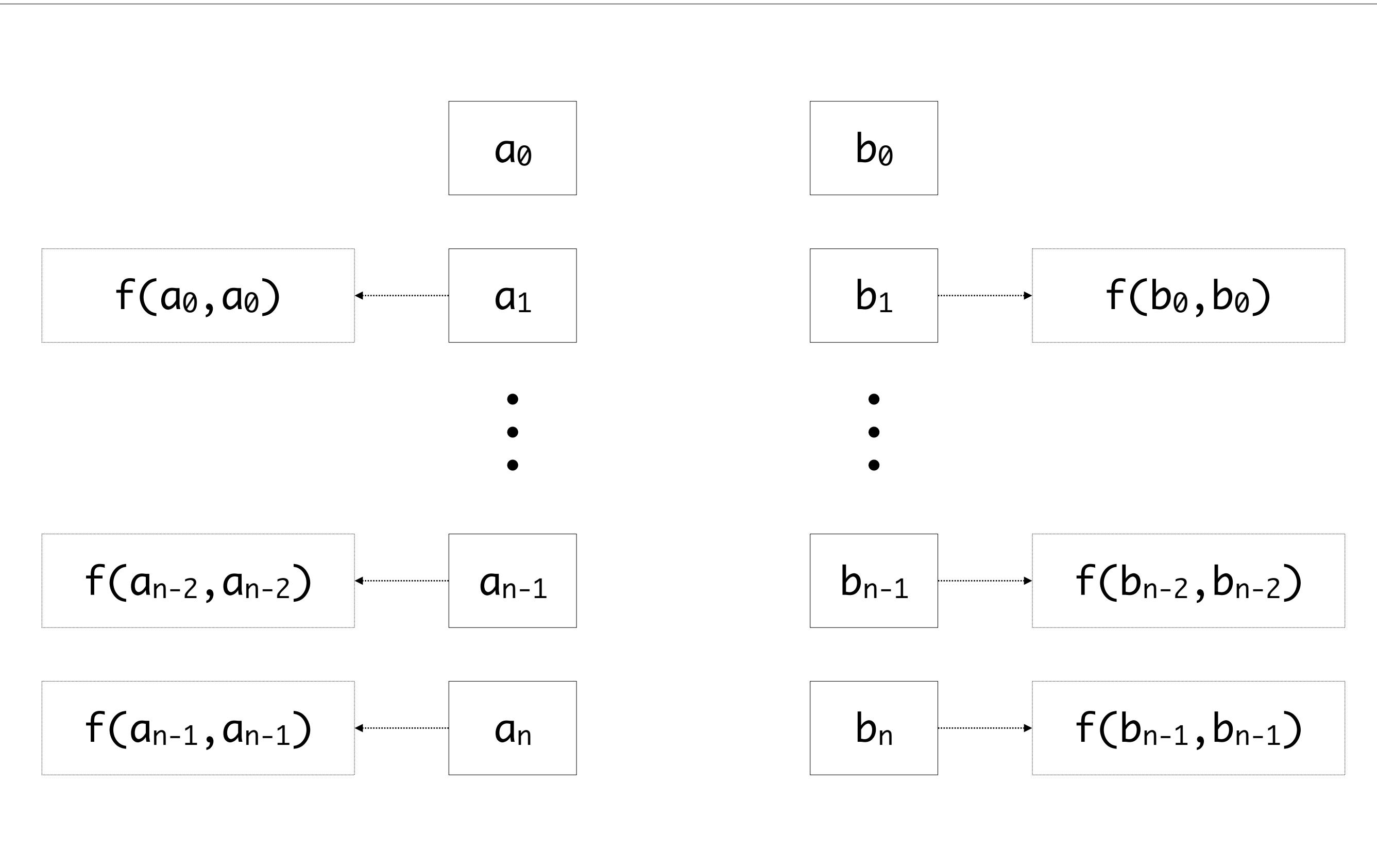


$$a_n == b_n$$

$$f(a_{n-1}, a_{n-1}) == f(b_{n-1}, b_{n-1})$$

# The Complex Case

$$h(a_1, \dots, a_n, f(b_0, b_0), \dots, f(b_{n-1}, b_{n-1}), a_n) == \\ h(f(a_0, a_0), \dots, f(a_{n-1}, a_{n-1}), b_1, \dots, b_{n-1}, b_n)$$



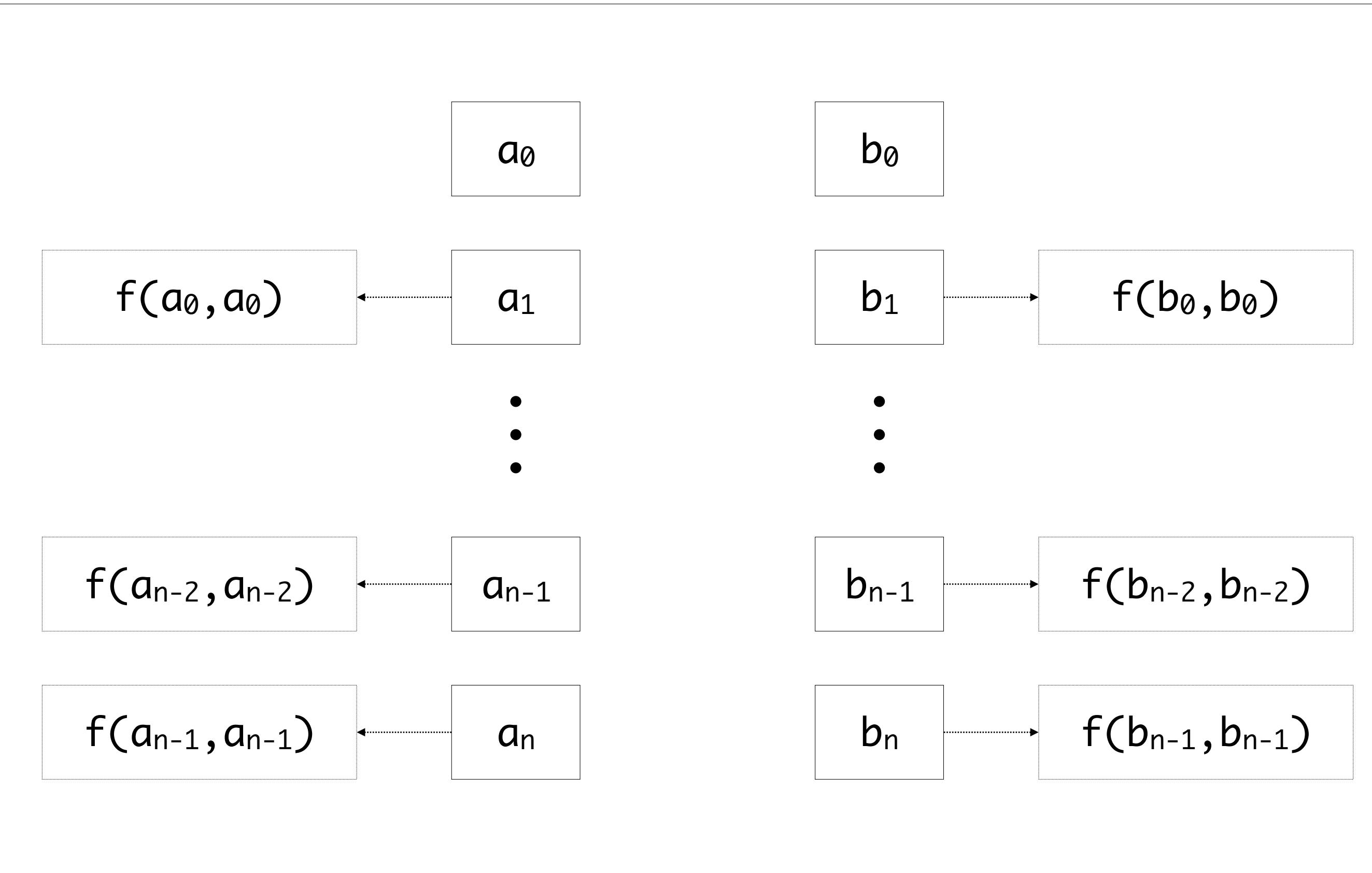
$$a_n == b_n$$

$$f(a_{n-1}, a_{n-1}) == f(b_{n-1}, b_{n-1})$$

$$a_{n-1} == b_{n-1} \quad a_{n-1} == b_{n-1}$$

# The Complex Case

$$h(a_1, \dots, a_n, f(b_0, b_0), \dots, f(b_{n-1}, b_{n-1}), a_n) == \\ h(f(a_0, a_0), \dots, f(a_{n-1}, a_{n-1}), b_1, \dots, b_{n-1}, b_n)$$



$$a_n == b_n$$

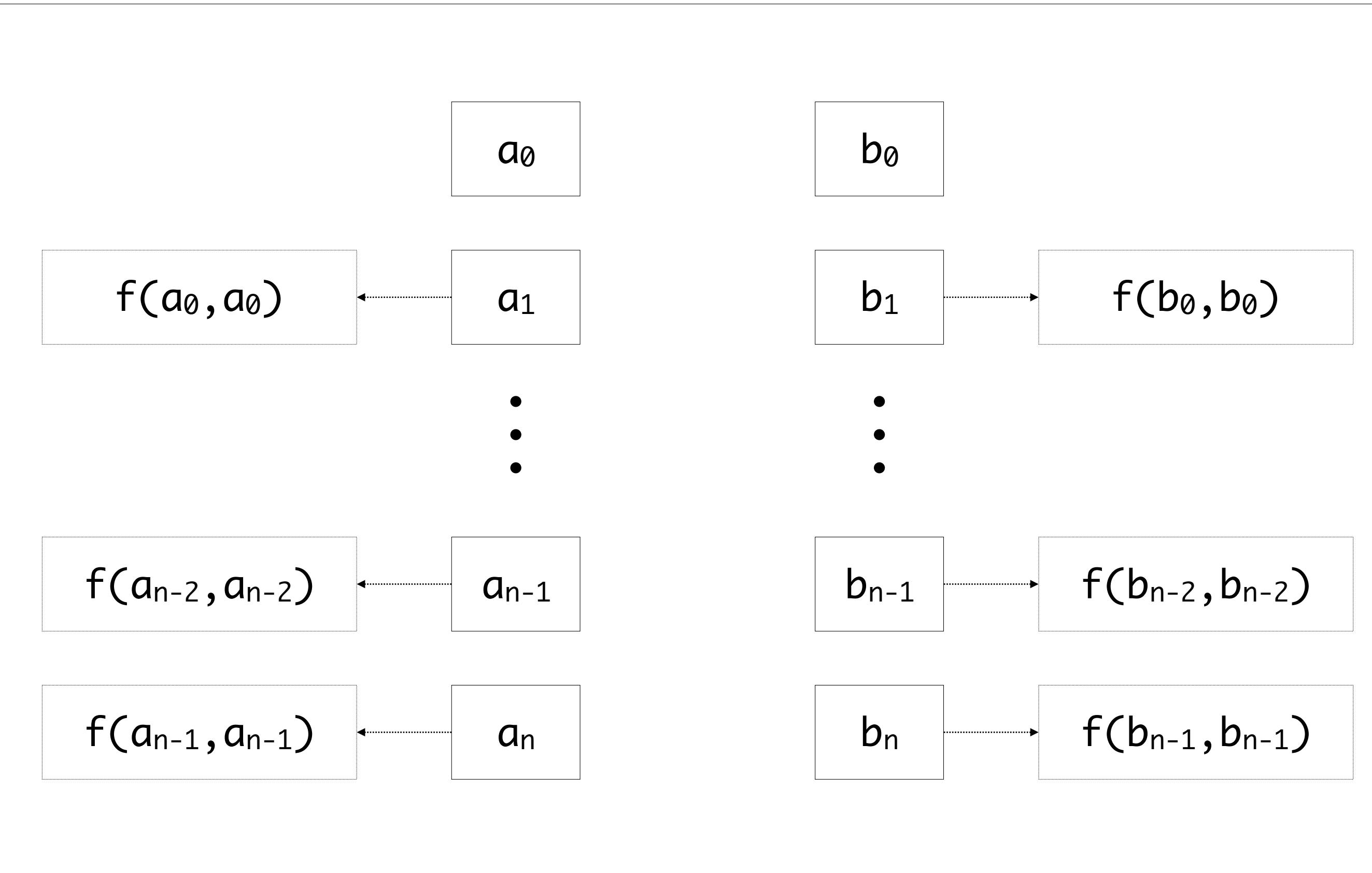
$$f(a_{n-1}, a_{n-1}) == f(b_{n-1}, b_{n-1})$$

$$a_{n-1} == b_{n-1} \quad a_{n-1} == b_{n-1}$$

$$f(a_{n-2}, a_{n-2}) == f(b_{n-2}, b_{n-2})$$

# The Complex Case

$$h(a_1, \dots, a_n, f(b_0, b_0), \dots, f(b_{n-1}, b_{n-1}), a_n) == \\ h(f(a_0, a_0), \dots, f(a_{n-1}, a_{n-1}), b_1, \dots, b_{n-1}, b_n)$$



$$a_n == b_n$$

$$f(a_{n-1}, a_{n-1}) == f(b_{n-1}, b_{n-1})$$

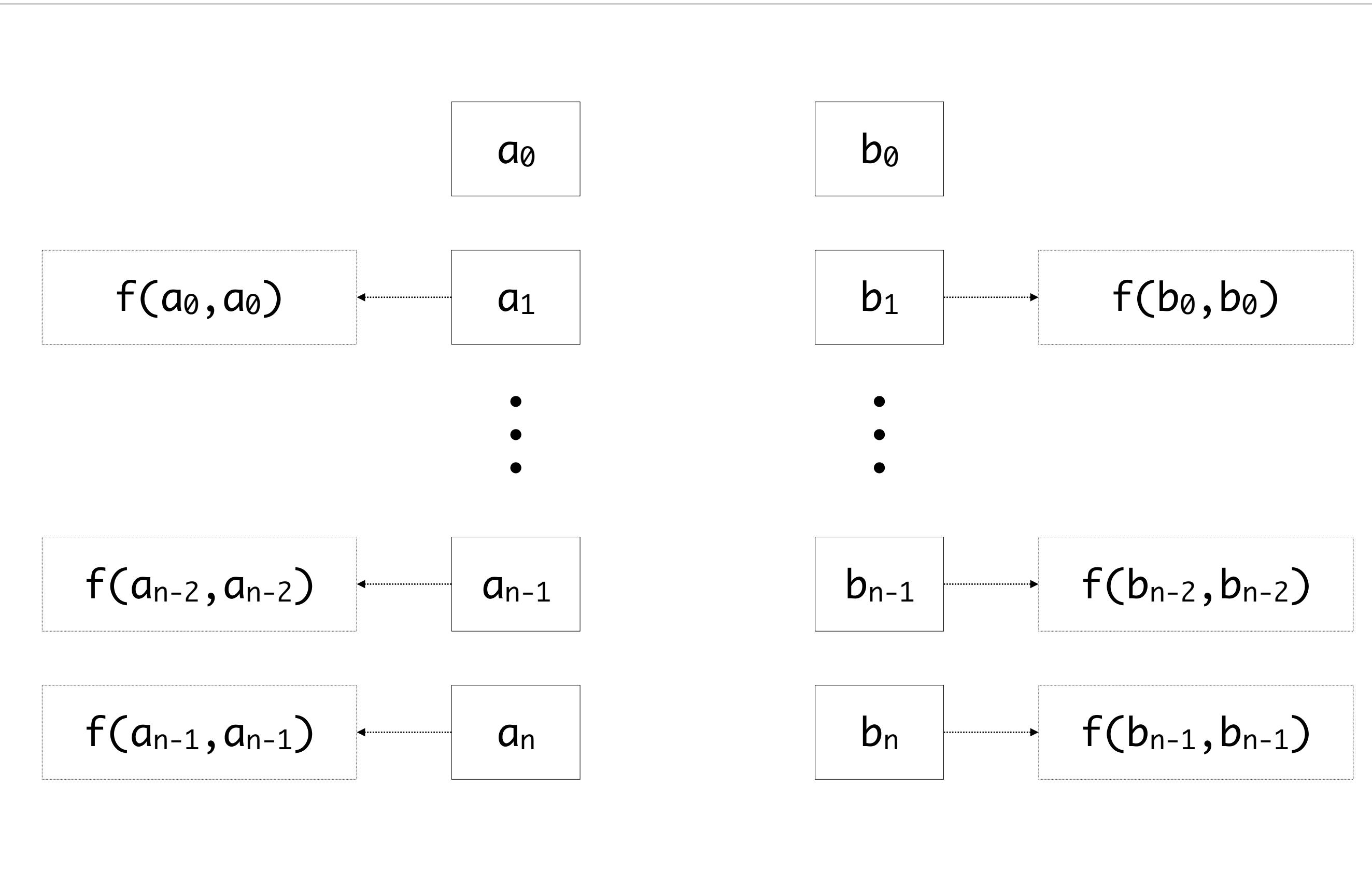
$$a_{n-1} == b_{n-1} \quad a_{n-1} == b_{n-1}$$

$$f(a_{n-2}, a_{n-2}) == f(b_{n-2}, b_{n-2})$$

⋮

# The Complex Case

$$h(a_1, \dots, a_n, f(b_0, b_0), \dots, f(b_{n-1}, b_{n-1}), a_n) == \\ h(f(a_0, a_0), \dots, f(a_{n-1}, a_{n-1}), b_1, \dots, b_{n-1}, b_n)$$



$$a_n == b_n$$

$$f(a_{n-1}, a_{n-1}) == f(b_{n-1}, b_{n-1})$$

$$a_{n-1} == b_{n-1} \quad a_{n-1} == b_{n-1}$$

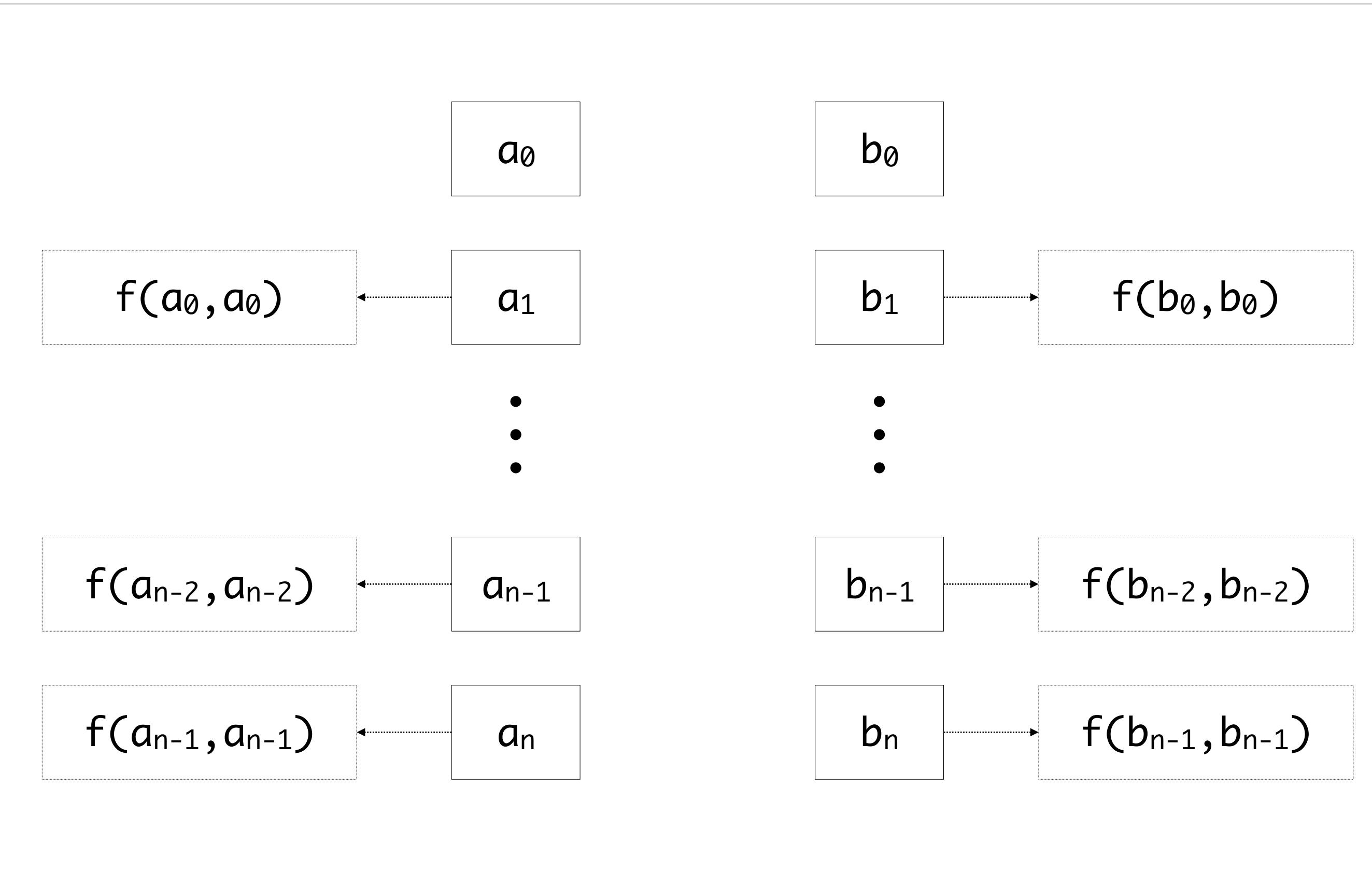
$$f(a_{n-2}, a_{n-2}) == f(b_{n-2}, b_{n-2})$$

⋮

$$a_1 == b_1 \quad a_1 == b_1$$

# The Complex Case

$$h(a_1, \dots, a_n, f(b_0, b_0), \dots, f(b_{n-1}, b_{n-1}), a_n) == \\ h(f(a_0, a_0), \dots, f(a_{n-1}, a_{n-1}), b_1, \dots, b_{n-1}, b_n)$$



$$a_n == b_n$$

$$f(a_{n-1}, a_{n-1}) == f(b_{n-1}, b_{n-1})$$

$$a_{n-1} == b_{n-1} \quad a_{n-1} == b_{n-1}$$

$$f(a_{n-2}, a_{n-2}) == f(b_{n-2}, b_{n-2})$$

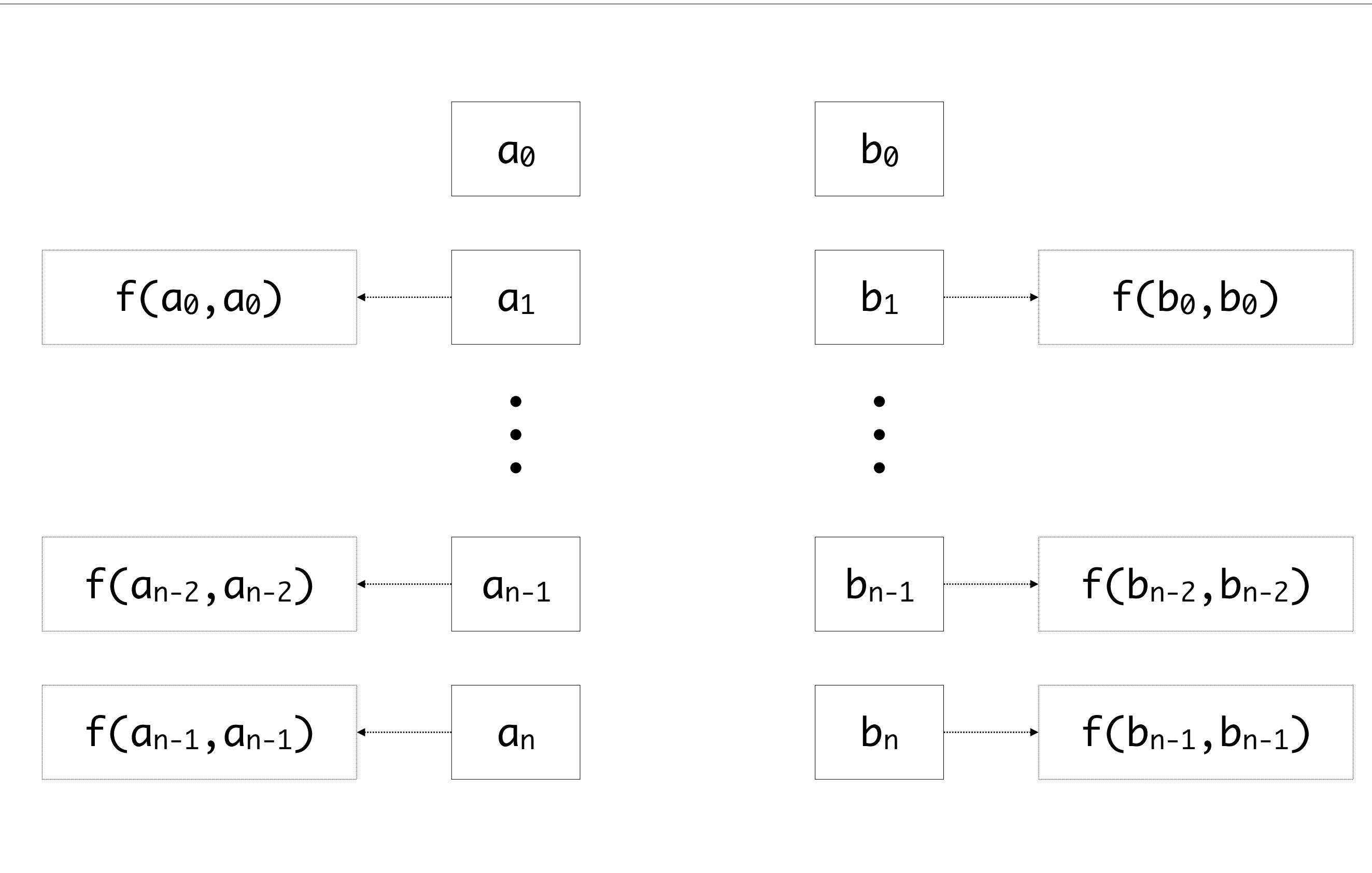
⋮

$$a_1 == b_1 \quad a_1 == b_1$$

$$f(a_0, a_0) == f(b_0, b_0)$$

# The Complex Case

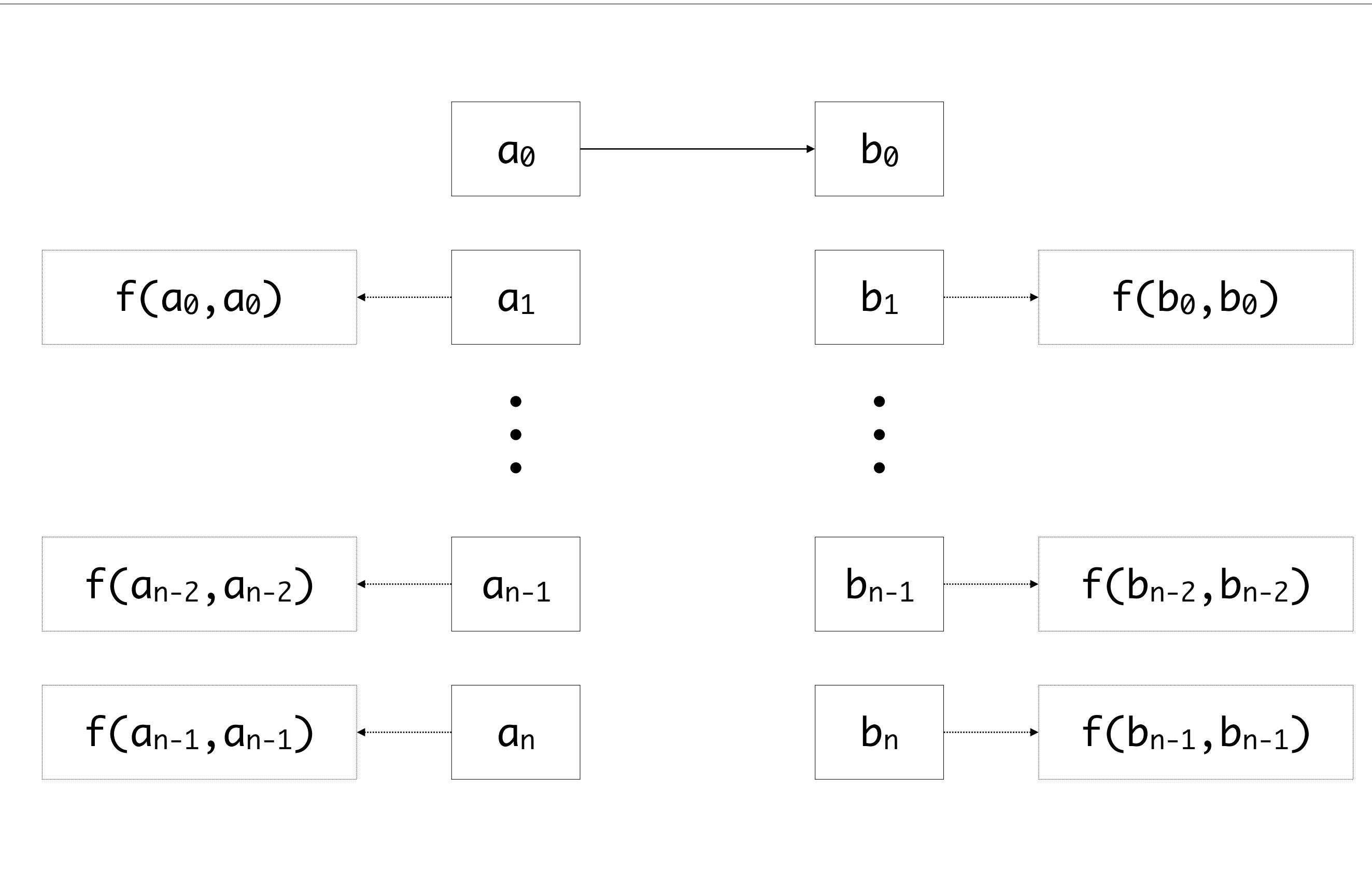
$$h(a_1, \dots, a_n, f(b_0, b_0), \dots, f(b_{n-1}, b_{n-1}), a_n) == \\ h(f(a_0, a_0), \dots, f(a_{n-1}, a_{n-1}), b_1, \dots, b_{n-1}, b_n)$$



$$\begin{aligned} a_n &== b_n \\ f(a_{n-1}, a_{n-1}) &== f(b_{n-1}, b_{n-1}) \\ a_{n-1} &== b_{n-1} & a_{n-1} &== b_{n-1} \\ f(a_{n-2}, a_{n-2}) &== f(b_{n-2}, b_{n-2}) \\ &\vdots \\ a_1 &== b_1 & a_1 &== b_1 \\ f(a_0, a_0) &== f(b_0, b_0) \\ a_0 &== b_0 & a_0 &== b_0 \end{aligned}$$

# The Complex Case

$$h(a_1, \dots, a_n, f(b_0, b_0), \dots, f(b_{n-1}, b_{n-1}), a_n) == \\ h(f(a_0, a_0), \dots, f(a_{n-1}, a_{n-1}), b_1, \dots, b_{n-1}, b_n)$$



$$a_n == b_n$$

$$f(a_{n-1}, a_{n-1}) == f(b_{n-1}, b_{n-1})$$

$$a_{n-1} == b_{n-1} \quad a_{n-1} == b_{n-1}$$

$$f(a_{n-2}, a_{n-2}) == f(b_{n-2}, b_{n-2})$$

⋮

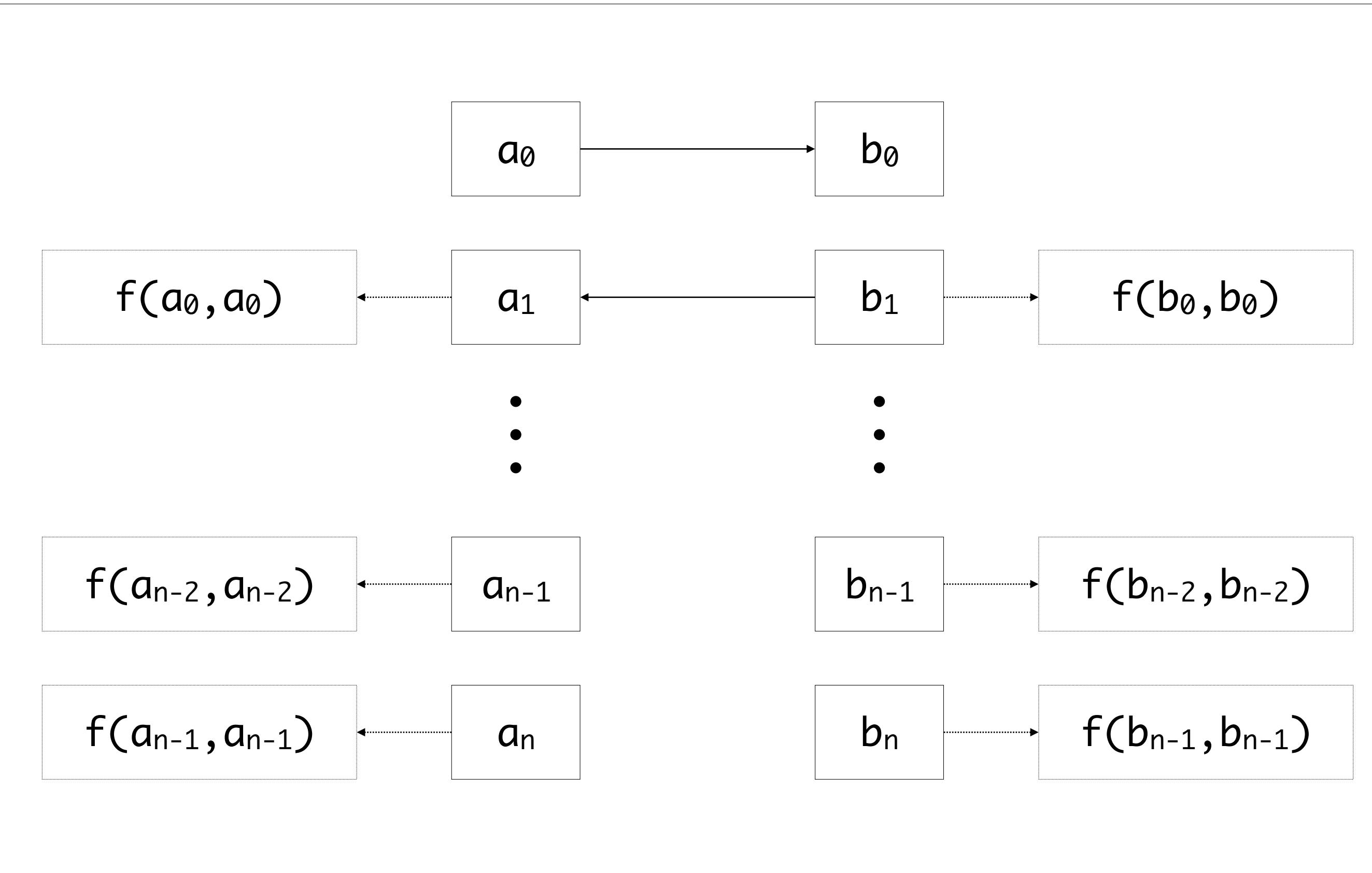
$$a_1 == b_1 \quad a_1 == b_1$$

$$f(a_0, a_0) == f(b_0, b_0)$$

$$a_0 == b_0 \quad a_0 == b_0$$

# The Complex Case

$$h(a_1, \dots, a_n, f(b_0, b_0), \dots, f(b_{n-1}, b_{n-1}), a_n) == \\ h(f(a_0, a_0), \dots, f(a_{n-1}, a_{n-1}), b_1, \dots, b_{n-1}, b_n)$$



$$a_n == b_n$$

$$f(a_{n-1}, a_{n-1}) == f(b_{n-1}, b_{n-1})$$

$$a_{n-1} == b_{n-1} \quad a_{n-1} == b_{n-1}$$

$$f(a_{n-2}, a_{n-2}) == f(b_{n-2}, b_{n-2})$$

⋮

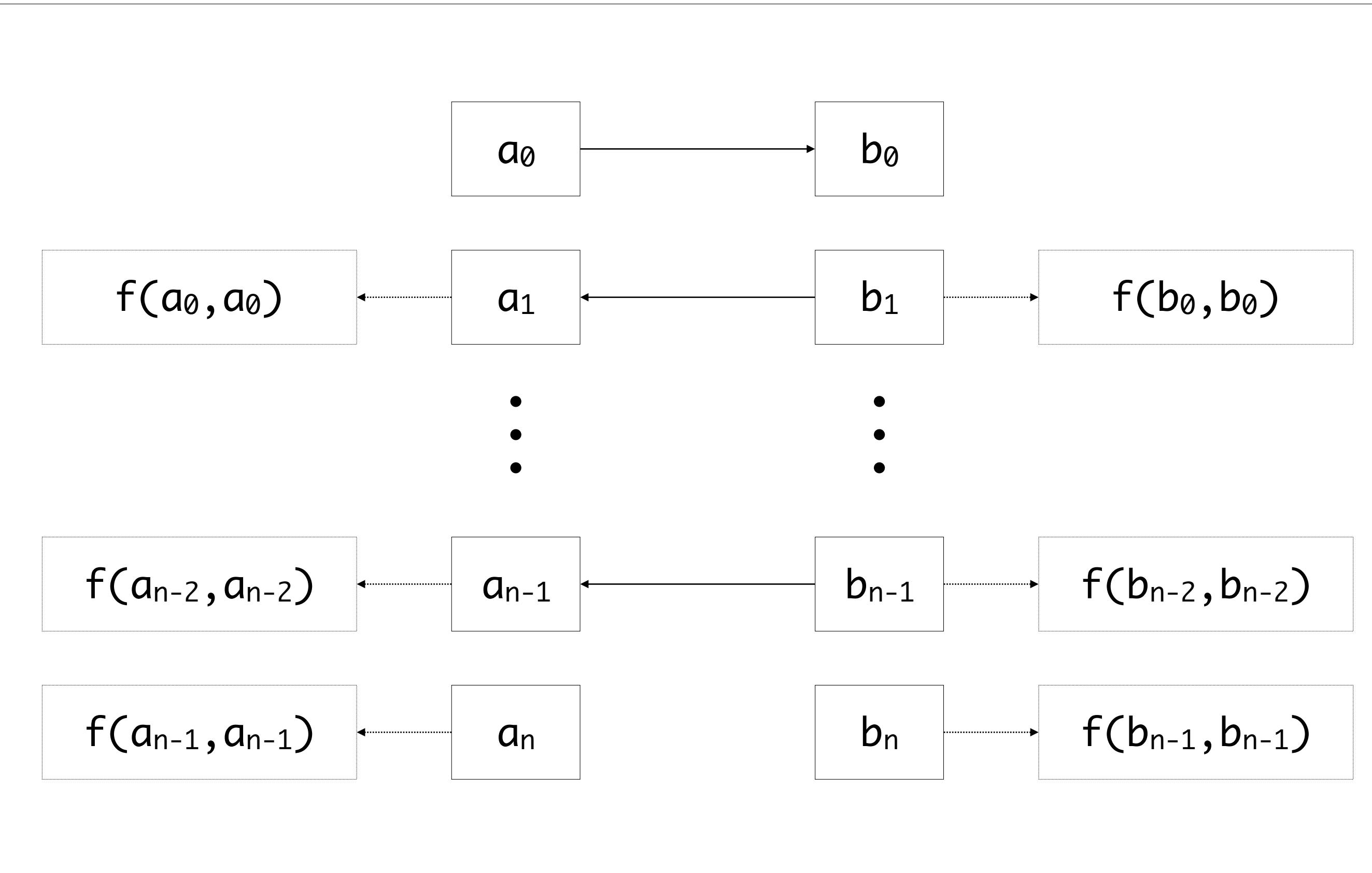
$$a_1 == b_1 \quad a_1 == b_1$$

$$f(a_0, a_0) == f(b_0, b_0)$$

$$a_0 == b_0 \quad a_0 == b_0$$

# The Complex Case

$$h(a_1, \dots, a_n, f(b_0, b_0), \dots, f(b_{n-1}, b_{n-1}), a_n) == \\ h(f(a_0, a_0), \dots, f(a_{n-1}, a_{n-1}), b_1, \dots, b_{n-1}, b_n)$$



$$a_n == b_n$$

$$f(a_{n-1}, a_{n-1}) == f(b_{n-1}, b_{n-1})$$

$$a_{n-1} == b_{n-1} \quad a_{n-1} == b_{n-1}$$

$$f(a_{n-2}, a_{n-2}) == f(b_{n-2}, b_{n-2})$$

⋮

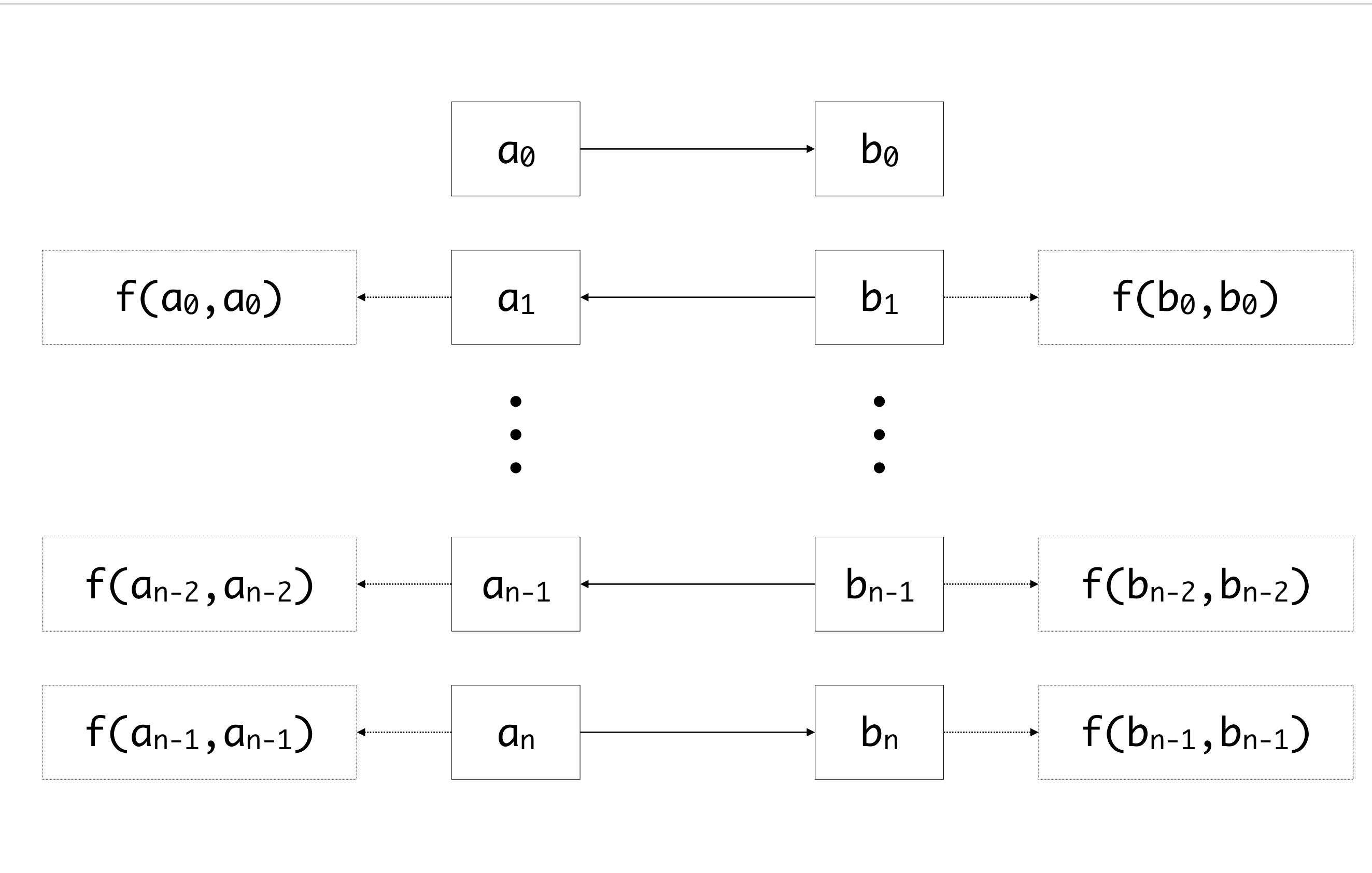
$$a_1 == b_1 \quad a_1 == b_1$$

$$f(a_0, a_0) == f(b_0, b_0)$$

$$a_0 == b_0 \quad a_0 == b_0$$

# The Complex Case

$$h(a_1, \dots, a_n, f(b_0, b_0), \dots, f(b_{n-1}, b_{n-1}), a_n) == \\ h(f(a_0, a_0), \dots, f(a_{n-1}, a_{n-1}), b_1, \dots, b_{n-1}, b_n)$$



$$a_n == b_n$$

$$f(a_{n-1}, a_{n-1}) == f(b_{n-1}, b_{n-1})$$

$$a_{n-1} == b_{n-1} \quad a_{n-1} == b_{n-1}$$

$$f(a_{n-2}, a_{n-2}) == f(b_{n-2}, b_{n-2})$$

⋮

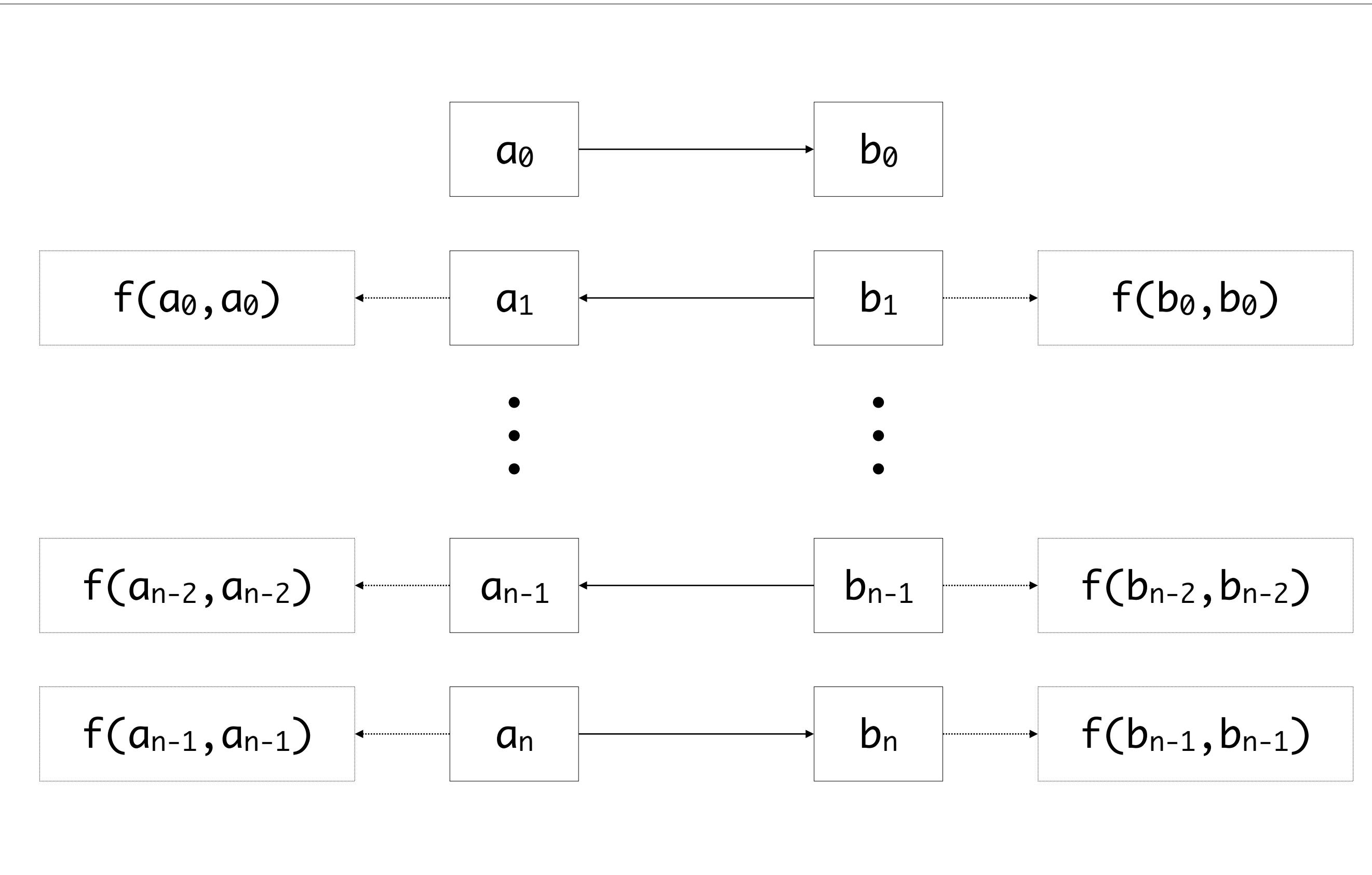
$$a_1 == b_1 \quad a_1 == b_1$$

$$f(a_0, a_0) == f(b_0, b_0)$$

$$a_0 == b_0 \quad a_0 == b_0$$

# The Complex Case

$h(a_1, \dots, a_n, f(b_0, b_0), \dots, f(b_{n-1}, b_{n-1}), a_n) ==$   
 $h(f(a_0, a_0), \dots, f(a_{n-1}, a_{n-1}), b_1, \dots, b_{n-1}, b_n)$

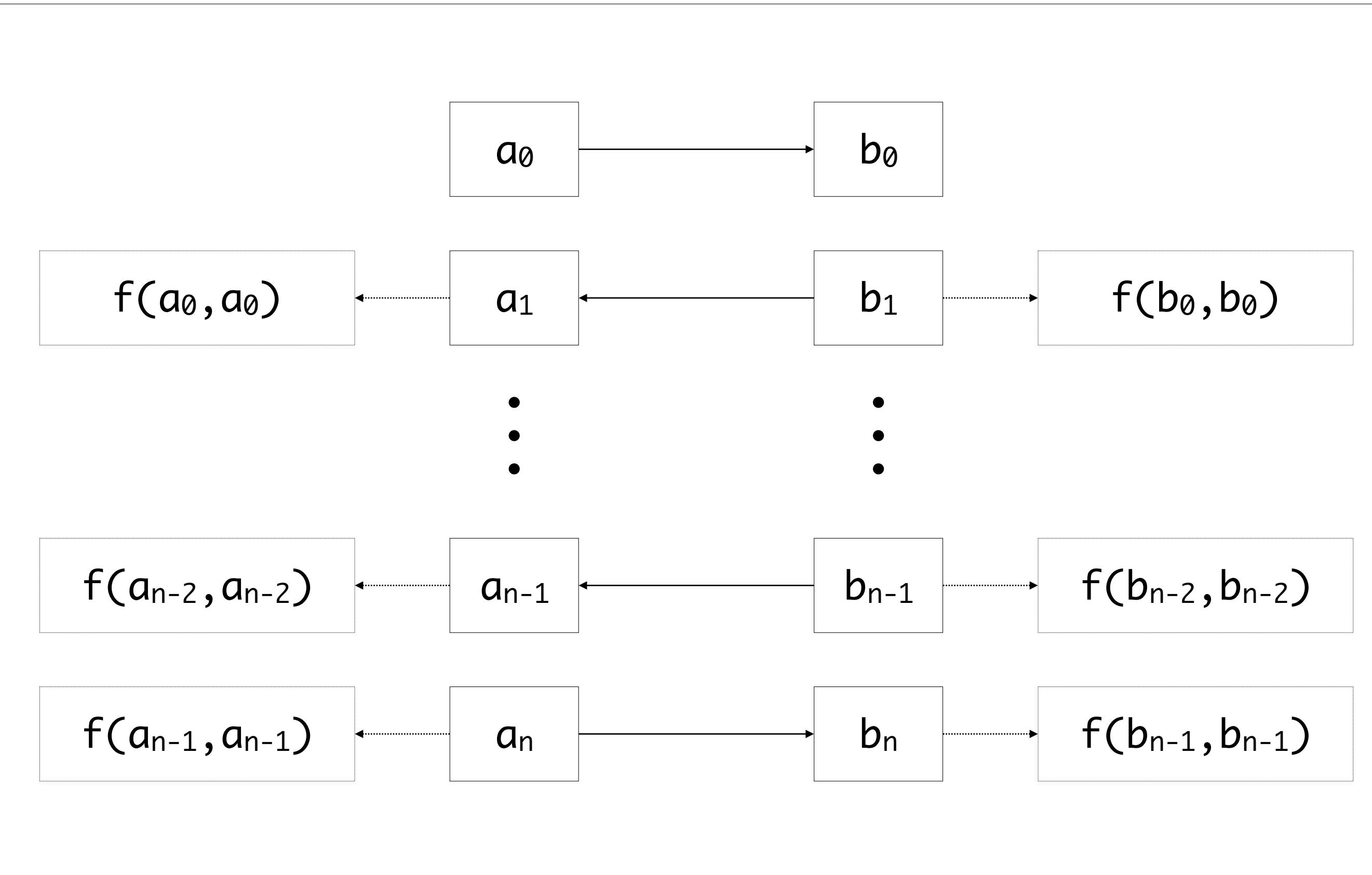


$a_n == b_n$   
 $f(a_{n-1}, a_{n-1}) == f(b_{n-1}, b_{n-1})$   
 $a_{n-1} == b_{n-1} \quad a_{n-1} == b_{n-1}$   
 $f(a_{n-2}, a_{n-2}) == f(b_{n-2}, b_{n-2})$   
 $\vdots$   
 $a_1 == b_1 \quad a_1 == b_1$   
 $f(a_0, a_0) == f(b_0, b_0)$   
 $a_0 == b_0 \quad a_0 == b_0$

How about occurrence checks?

# The Complex Case

$h(a_1, \dots, a_n, f(b_0, b_0), \dots, f(b_{n-1}, b_{n-1}), a_n) ==$   
 $h(f(a_0, a_0), \dots, f(a_{n-1}, a_{n-1}), b_1, \dots, b_{n-1}, b_n)$



$a_n == b_n$   
 $f(a_{n-1}, a_{n-1}) == f(b_{n-1}, b_{n-1})$   
 $a_{n-1} == b_{n-1}$        $a_{n-1} == b_{n-1}$   
 $f(a_{n-2}, a_{n-2}) == f(b_{n-2}, b_{n-2})$   
⋮  
 $a_1 == b_1$        $a_1 == b_1$   
 $f(a_0, a_0) == f(b_0, b_0)$   
 $a_0 == b_0$        $a_0 == b_0$

How about occurrence checks? Postpone!

# Union-Find

## Main idea

Martelli, Montanari. An Efficient Unification Algorithm. TOPLAS, 1982

# Union-Find

## Main idea

- Represent unifier as graph

# Union-Find

## Main idea

- Represent unifier as graph
- One variable represent equivalence class

# Union-Find

## Main idea

- Represent unifier as graph
- One variable represent equivalence class
- Replace substitution by union & find operations

# Union-Find

## Main idea

- Represent unifier as graph
- One variable represent equivalence class
- Replace substitution by union & find operations
- Testing equality becomes testing node identity

# Union-Find

## Main idea

- Represent unifier as graph
- One variable represent equivalence class
- Replace substitution by union & find operations
- Testing equality becomes testing node identity

## Optimizations

# Union-Find

## Main idea

- Represent unifier as graph
- One variable represent equivalence class
- Replace substitution by union & find operations
- Testing equality becomes testing node identity

## Optimizations

- Path compression make recurring lookups fast

# Union-Find

## Main idea

- Represent unifier as graph
- One variable represent equivalence class
- Replace substitution by union & find operations
- Testing equality becomes testing node identity

## Optimizations

- Path compression make recurring lookups fast
- Tree balancing keeps paths short

# Union-Find

## Main idea

- Represent unifier as graph
- One variable represent equivalence class
- Replace substitution by union & find operations
- Testing equality becomes testing node identity

## Optimizations

- Path compression make recurring lookups fast
- Tree balancing keeps paths short

## Complexity

# Union-Find

## Main idea

- Represent unifier as graph
- One variable represent equivalence class
- Replace substitution by union & find operations
- Testing equality becomes testing node identity

## Optimizations

- Path compression make recurring lookups fast
- Tree balancing keeps paths short

## Complexity

- Linear in space and almost linear (inverse Ackermann) in time

# Union-Find

## Main idea

- Represent unifier as graph
- One variable represent equivalence class
- Replace substitution by union & find operations
- Testing equality becomes testing node identity

## Optimizations

- Path compression make recurring lookups fast
- Tree balancing keeps paths short

## Complexity

- Linear in space and almost linear (inverse Ackermann) in time
- Easy to extract triangular unifier from graph

# Union-Find

## Main idea

- Represent unifier as graph
- One variable represent equivalence class
- Replace substitution by union & find operations
- Testing equality becomes testing node identity

## Optimizations

- Path compression make recurring lookups fast
- Tree balancing keeps paths short

## Complexity

- Linear in space and almost linear (inverse Ackermann) in time
- Easy to extract triangular unifier from graph
- Postpone occurrence checks to prevent traversing (potentially) large terms

Except where otherwise noted, this work is licensed under

