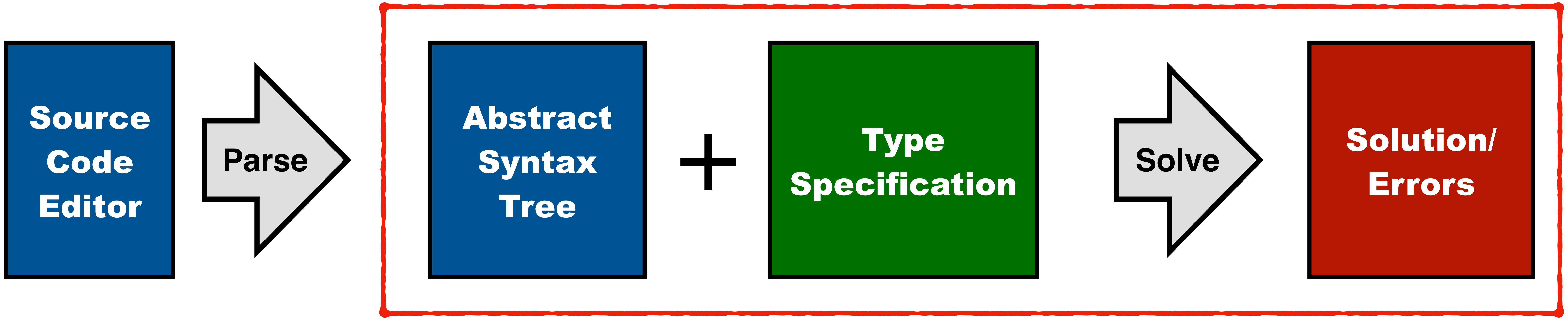
Constraint Semantics and Constraint Resolution

Hendrik van Antwerpen
Eelco Visser

TU Delft

CS4200 | Compiler Construction | October 1, 2020

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.

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

CHAPTER 8

Unification theory

Franz Baader

Wayne Snyder

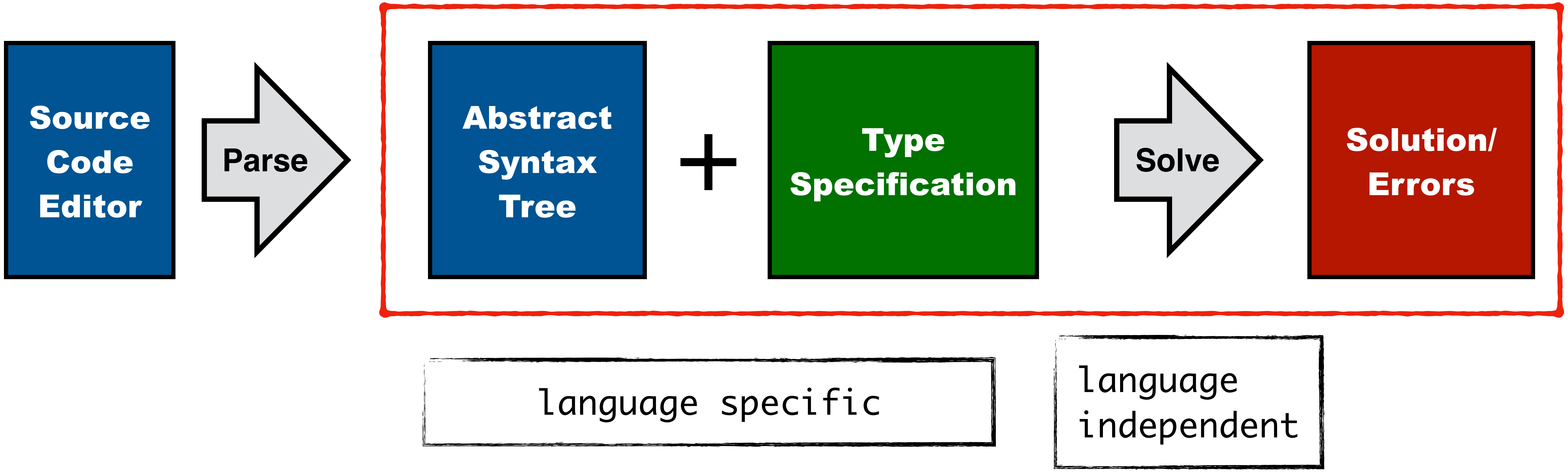
SECOND READERS: Paliath Narendran, Manfred Schmidt-Schauss, and Klaus Schulz.

Contents

HANDBOOK OF AUTOMATED REASONING
Edited by Alan Robinson and Andrei Voronkov

© Elsevier Science Publishers B.V., 2001

Type Checking with Specifications


Source Code Editor
Parse
Abstract Syntax Tree
+
Type Specification
Solve
Solution/ Errors
language specific
language independent

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

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?

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

- 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?

When are constraints satisfied?

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

Semantics of (a Subset of) Statix Constraints

Syntax

```
C = t == t               // equality
  | r in s |-> d         // name resolution (short for query var … in s |-> [d])
  | C /\ C               // conjunction
```

Declarative semantics

```
G,ϕ ⊨ t == u                if ϕ(t) = ϕ(u)

G,ϕ ⊨ r in s |-> d          if  ϕ(r) = x
                            and ϕ(d) = x
                            and ϕ(s) = #i
                            and x resolves to x from #i in G

G,ϕ ⊨ C1 /\ C2              if G,ϕ ⊨ C1 and G,ϕ ⊨ C2
```

Using the Semantics

Program

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

Program constraints

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

Unifier ϕ (model)

```
ϕ = { ty1 -> INT(),
      ty2 -> INT(),
      ty3 -> FUN(INT(),ty5),
      ty4 -> INT(),
      d1  -> "i",
      d2  -> "f"
    }
```

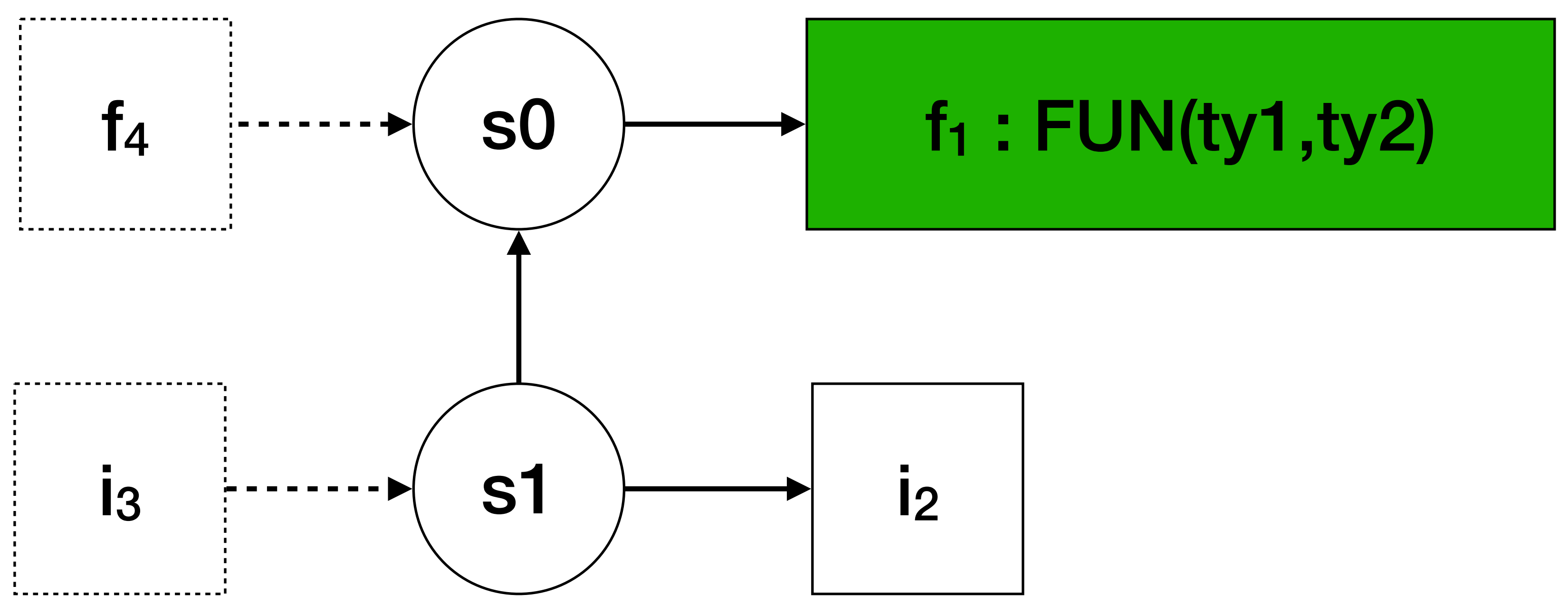
Constraint semantics

```
G,ϕ ⊨ t == u
    if ϕ(t) = ϕ(u)

G,ϕ ⊨ r in s |-> d
    if  ϕ(r) = x
    and ϕ(d) = x
    and ϕ(s) = #i
    and x resolves to x from #i in G

G,ϕ ⊨ C1 /\ C2
    if  G,ϕ ⊨ C1
    and G,ϕ ⊨ C2
```

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()
"i" in #s1 |-> d1
ty2 == INT()
"f" in #s0 |-> d2
ty3 == FUN(ty4,ty5)
ty4 == INT()
…
```

Unifier ϕ (model)

```
ϕ = { ty1 -> INT(),
      ty2 -> INT(),
      ty3 -> FUN(INT(),ty5),
      ty4 -> INT(),
      d1  -> "i",
      d2  -> "f"
    }
```

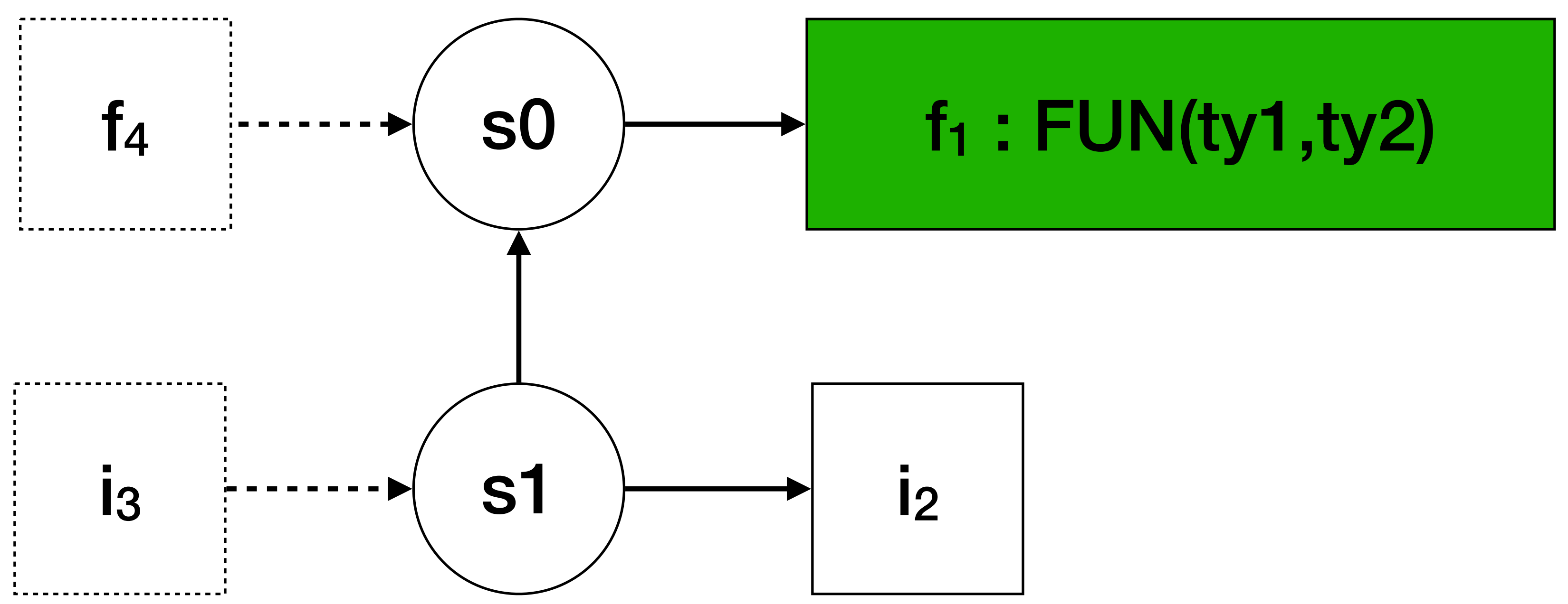
Constraint semantics

```
G,ϕ ⊨ t == u
    if ϕ(t) = ϕ(u)

G,ϕ ⊨ r in s |-> d
    if  ϕ(r) = x
    and ϕ(d) = x
    and ϕ(s) = #i
    and x resolves to x from #i in G

G,ϕ ⊨ C₁ /\ C₂
    if  G,ϕ ⊨ C₁
    and G,ϕ ⊨ C₂
```

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

Unifier ϕ (model)

```
ϕ = { ty1 -> INT(),
      ty2 -> INT(),
      ty3 -> FUN(INT(),ty5),
      ty4 -> INT(),
      d1  -> "i",
      d2  -> "f"
    }
```

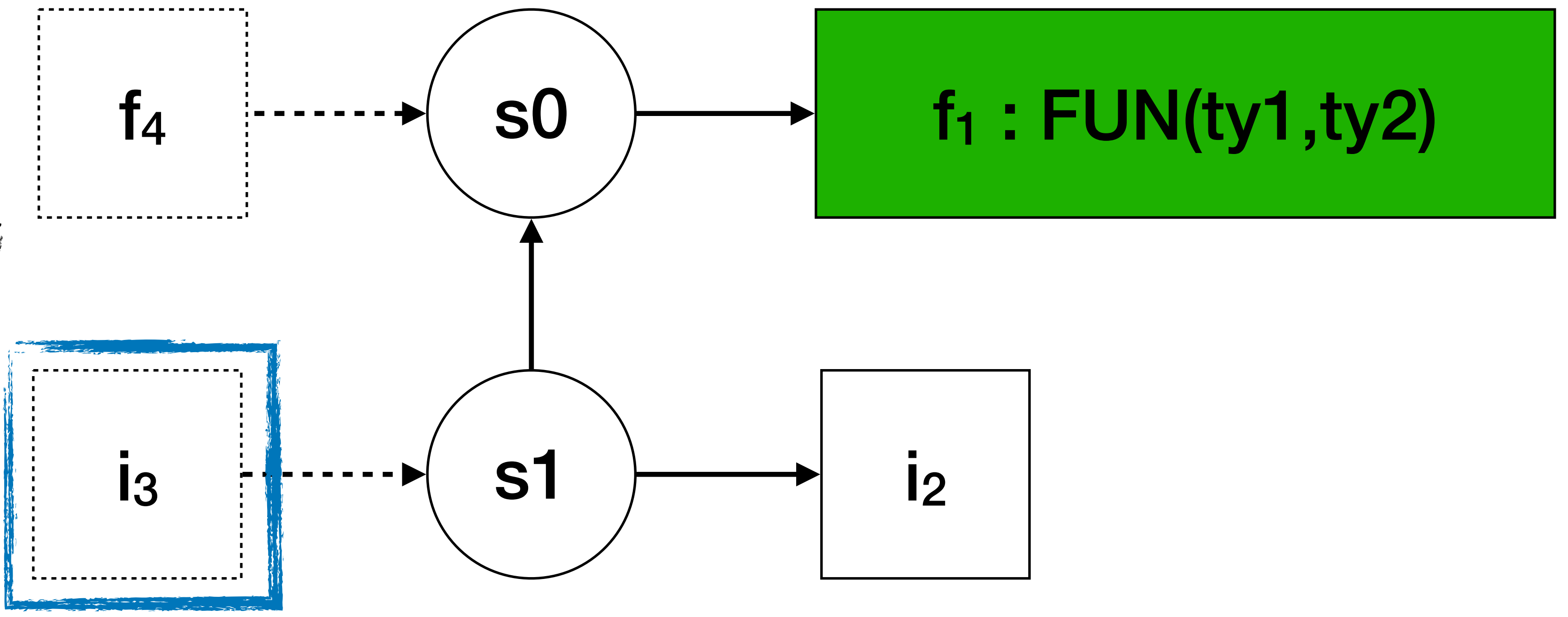
Constraint semantics

```
G,ϕ ⊨ t == u
    if ϕ(t) = ϕ(u)

G,ϕ ⊨ r in s |-> d
    if  ϕ(r) = x
    and ϕ(d) = x
    and ϕ(s) = #i
    and x resolves to x from #i in G

G,ϕ ⊨ C1 /\ C2
    if  G,ϕ ⊨ C1
    and G,ϕ ⊨ C2
```

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()
"i" in #s1 |-> d1
ty2 == INT()
"f" in #s0 |-> d2
ty3 == FUN(ty4,ty5)
ty4 == INT()
…
```

Unifier ϕ (model)

```
ϕ = { ty1 -> INT(),
      ty2 -> INT(),
      ty3 -> FUN(INT(),ty5),
      ty4 -> INT(),
      d1  -> "i",
      d2  -> "f"
    }
```

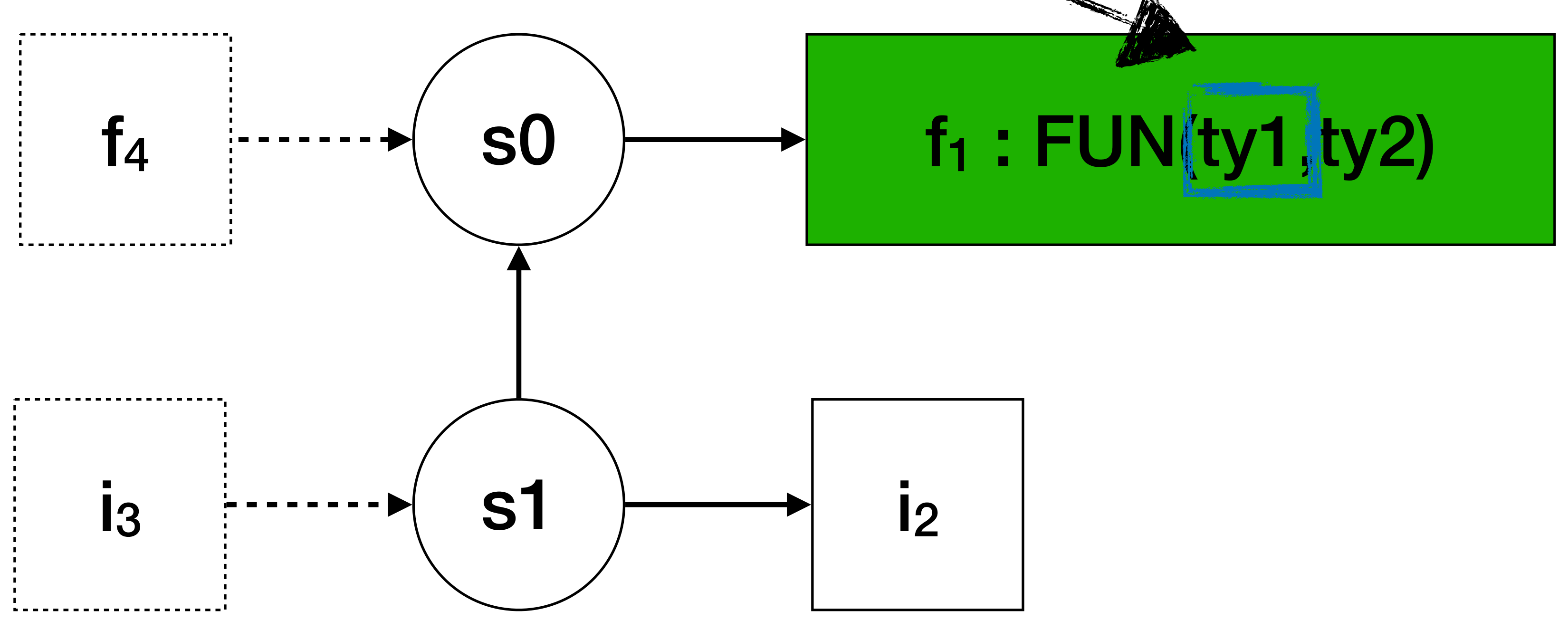
Constraint / logic variables

Constraint semantics

```
G,ϕ ⊨ t == u
    if ϕ(t) = ϕ(u)

G,ϕ ⊨ r in s |-> d
    if  ϕ(r) = x
    and ϕ(d) = x
    and ϕ(s) = #i
    and x resolves to x from #i in G

G,ϕ ⊨ C1 /\ C2
    if  G,ϕ ⊨ C1
    and G,ϕ ⊨ C2
```



Different Kinds of Variables

Program

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

Program constraints

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

Unifier ϕ (model)

```
ϕ = { ty1 -> INT(),
      ty2 -> INT(),
      ty3 -> FUN(INT(),ty5),
      ty4 -> INT(),
      d1  -> "i",
      d2  -> "f"
    }
```

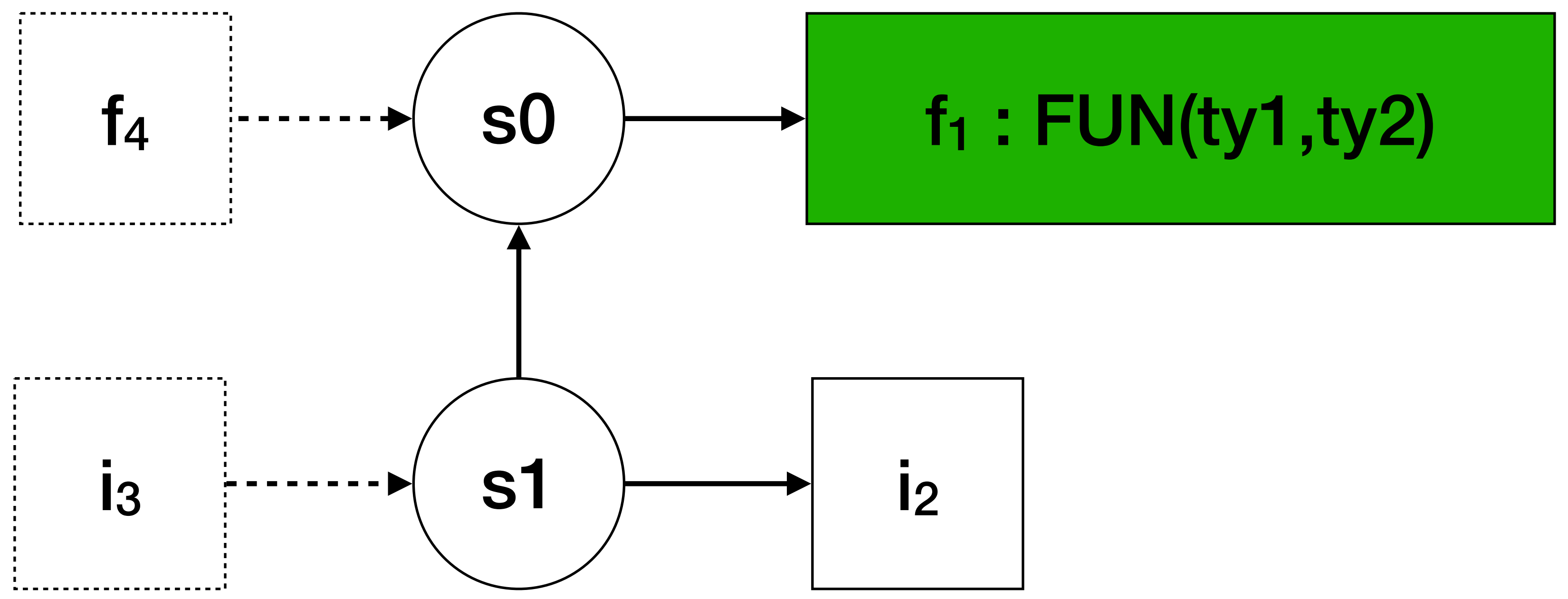
Constraint semantics

```
G,ϕ ⊨ t == u
    if ϕ(t) = ϕ(u)

G,ϕ ⊨ r in s |-> d
    if  ϕ(r) = x
    and ϕ(d) = x
    and ϕ(s) = #i
    and x resolves to x from #i in G

G,ϕ ⊨ C1 /\ C2
    if  G,ϕ ⊨ C1
    and G,ϕ ⊨ C2
```

Semantics meta-variables



Type Checking

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))].
```

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

Inferring the Type of a Parameter

Unknown S!

```
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?
- 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;
    }
}
```

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()) + ?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 |-> [(_, (_, T))].
```

How to check types?

What are challenges when implementing a type checker?

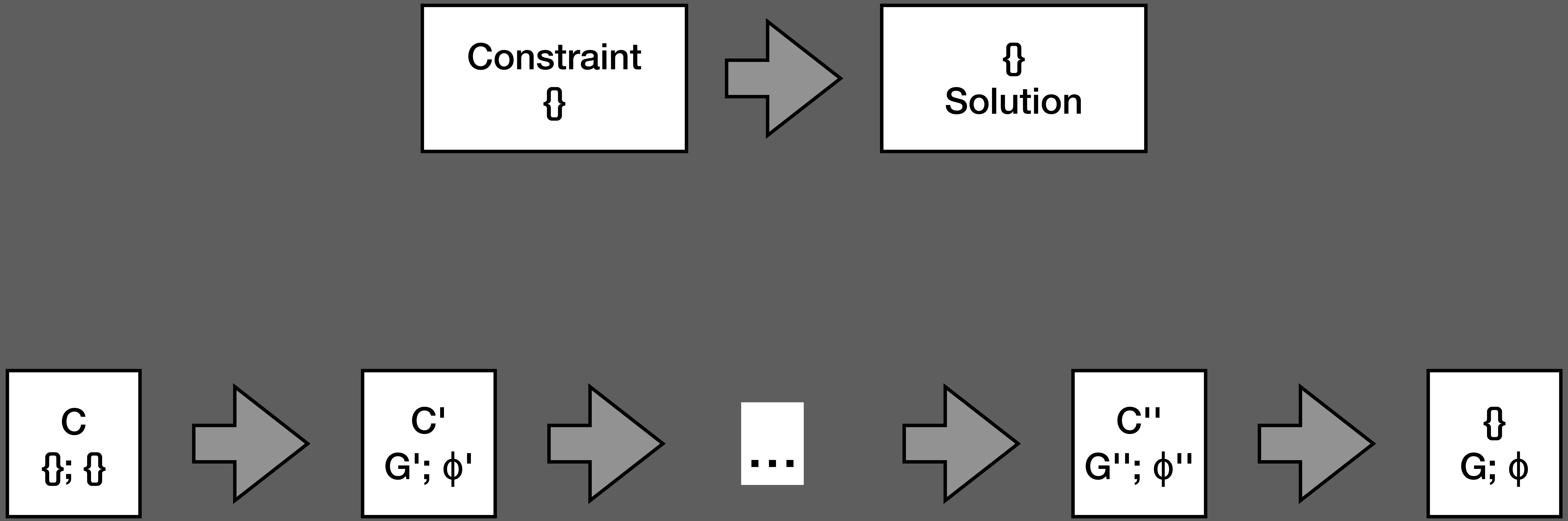
- Collecting necessary binding information before using it
- Gradually learning type information

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

Non-deterministic constraint selection

```
<C; G, ϕ> ⟶ <C; G, ϕ>
```

```
<t == u, C; G, ϕ> ⟶ <C; G, ϕ'> where unify(ϕ,t,u) = ϕ'

<s1 -L→ s2, C; G, ϕ> ⟶ <C; G', ϕ> where ϕ(s1) = #i, ϕ(s2) = #j,
                                        G + {#i -L→ #j} = G'

<r in s ↦ t, C; G, ϕ> ⟶ <t == d; G, ϕ> where ϕ(r) = Ns{x}, ϕ(s) = #i,
                                        resolve(G, #i, Ns{x}) = d
```

```
def solve(C):
  if <C; {}, {}> ⟶* <{}; G, ϕ>:
    return <G, ϕ>
  else:
    fail
```

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

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?

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

How are these related?

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

Term Equality
& Unification

Syntactic Terms

Generic Terms

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

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

function symbol

arguments

$f(t_0,\ldots,t_n)$

arity

Syntactic Equality

$f(t_0,\ldots,t_n) == g(u_0,\ldots,u_m)$ 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	ϕ

variable

f(g(),a)

substitution

ϕ = { a -> f(g(),b), b -> h() }

domain

```
ϕ(a)            = t                         if { a -> t } in ϕ
ϕ(a)            = a                         otherwise
ϕ(f(t0,…,tn)) = f(ϕ(t0),…,ϕ(tn))
```

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

<u>ground term</u>: a term without variables

Unifiers

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	ϕ

<u>unifier</u>: a substitution that makes terms equal

```
f(a,g()) == f(h(),b)    ⇨    a -> h()       ⇨    f(h(),g()) == f(h(),g())
                             b -> g()

g(a,f(b)) == g(f(h()),a)  ⇨  a -> f(h())    ⇨    g(f(h()),f(h())) == g(f(h()),f(h()))
                             b -> h()

f(a,h()) == g(h(),b)    ⇨    no unifier, f != g

f(b,b) == b             ⇨    b -> f(b,b) <------ not idempotent
```

Most General Unifiers

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	ϕ

```
f(a,b) == f(b,c)  =>  a -> g()
                      b -> g()
                      c -> g()   =>  f(g(),g()) == f(g(),g())

                      a -> b
                      c -> b     =>  f(b,b) == f(b,b)

                      b -> a
                      c -> a     =>  f(a,a) == f(a,a)
```

most general unifiers

Most General Unifiers

```
terms         t, u
functions     f, g, h
variables     a, b, c
substitution  ϕ
```

every unifier is an instance of a most general unifier

(implicit) identity case

```
a -> b                 a -> g()
b -> b   =[b -> g()]=> b -> g()
c -> b                 c -> g()
```

most general unifiers are related by renaming substitutions

```
a -> b               a -> a
b -> b   =[b -> a]=> b -> a
c -> b               c -> a
```

```
a -> a               a -> b
b -> a   =[a -> b]=> b -> b
c -> a               c -> b
```

Unification

```
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(t₁,...,tₙ) and u == g(u₁,...,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

$t == f(t_1,\ldots,t_5)$, $u == f(u_1,\ldots,u_5)$
matching terms

$t == f(t_0,\ldots,t_5)$, $u == g(u_0,\ldots,u_3)$
mismatching terms

$t == f(t_0,\ldots,t_5)$, $u == b$
swap terms

$t == a$, $u == k(g(a,f()))$
recursive terms

$t == a$, $u == k(u_0,\ldots,u_5)$
extend unifier

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	ϕ

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

Efficient Unification with Union-Find

Complexity of Unification

Space complexity

- Exponential
- Representation of unifier

Time complexity

- Exponential
- Recursive calls on terms

Solution

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

terms	t, u
functions	f, g, h
variables	a, b, c
substitution	ϕ

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

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

fully applied

$a_1 \text{ -> } f(a_0,a_0)$
$a_2 \text{ -> } f(a_1,a_1)$
$a_i \text{ -> } \ldots\ 3 \text{ subterms } \ldots$
$b_1 \text{ -> } f(a_0,a_0)$
$b_2 \text{ -> } f(a_1,a_1)$
$b_i \text{ -> } \ldots\ 3 \text{ subterms } \ldots$

triangular

Set Representatives

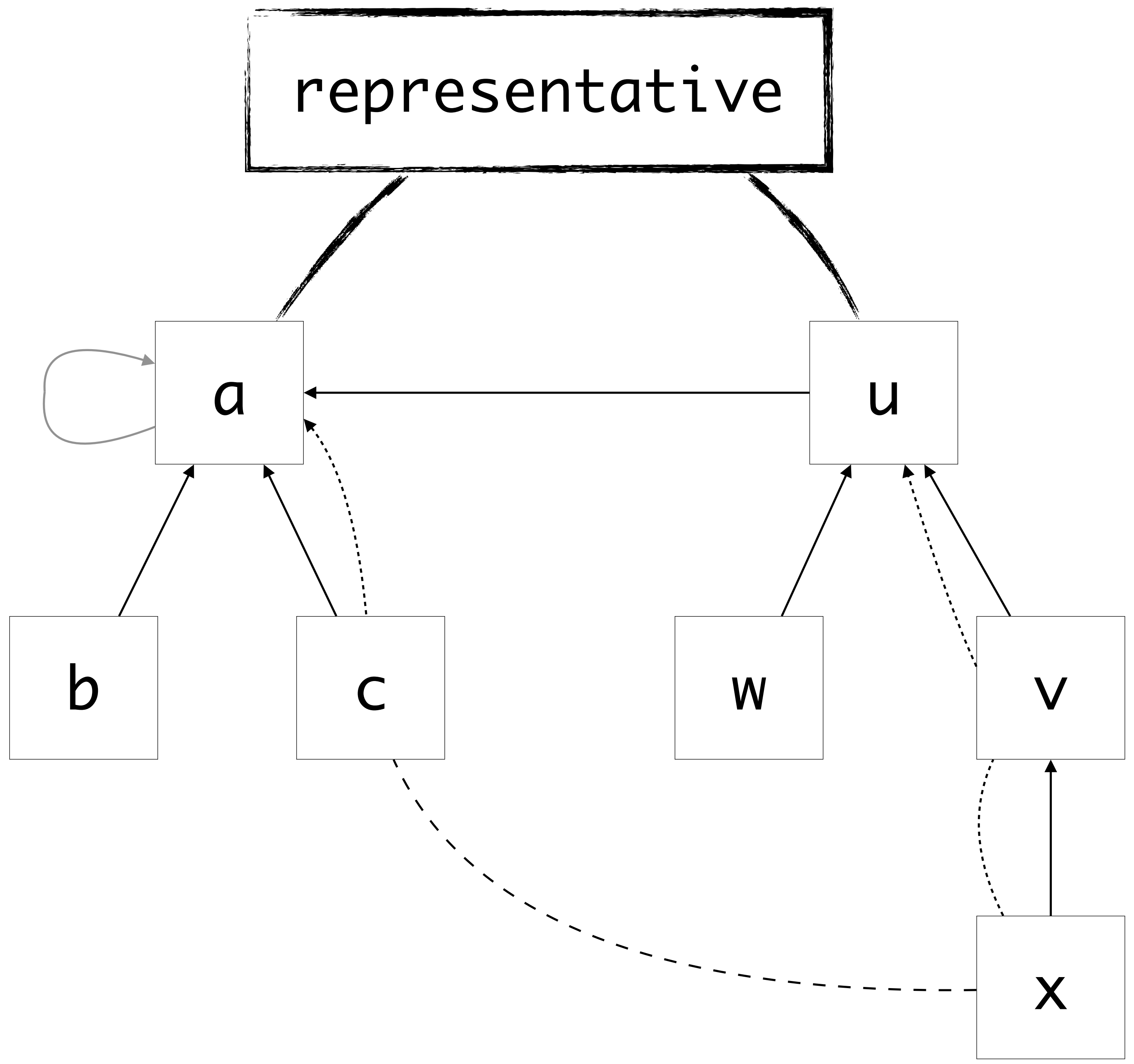
```
FIND(a):
  b := rep(a)
  if b == a:
     return a
  else
     return FIND(b)



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

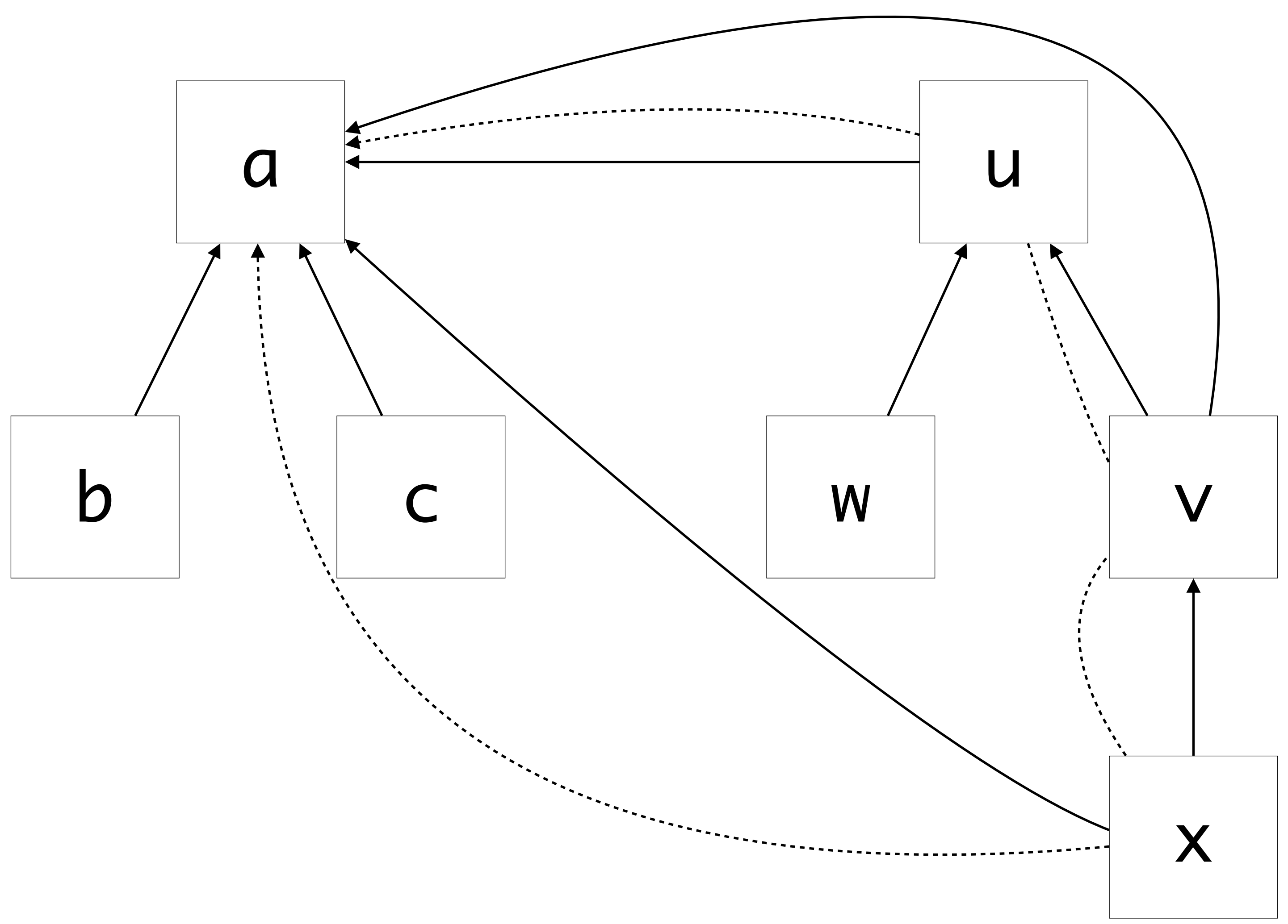
```
FIND(a):
  b := rep(a)
  if b == a:
     return a
  else
     return FIND(b)



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

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

…
x == b
x == c
x == w
x == v



Tree Balancing

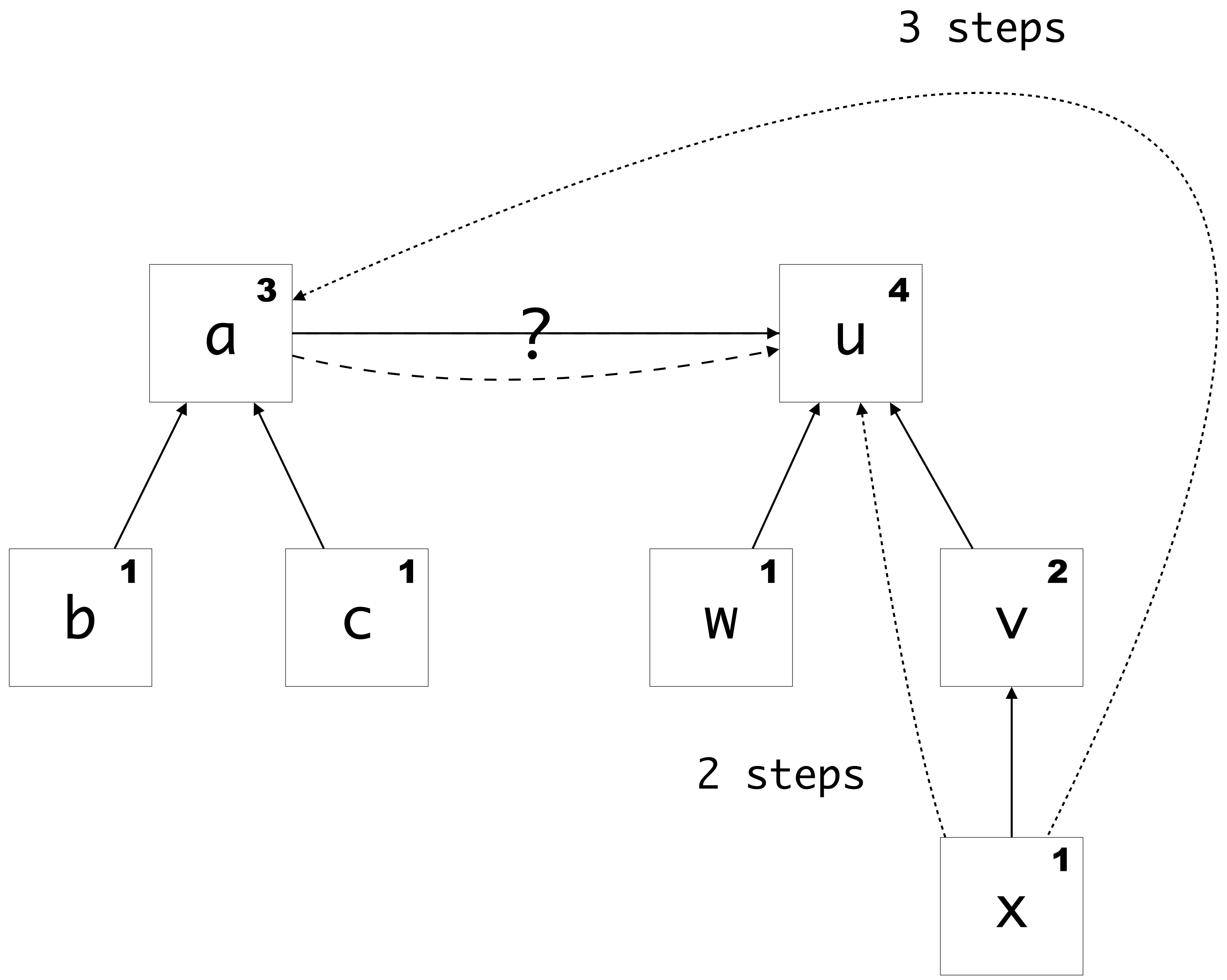
```
FIND(a):
  b := rep(a)
  if b == a:
     return a
  else
     b := FIND(b)
     rep(a) := b
     return b

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

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

…

x == c



The Complex Case

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

$$
\begin{tikzcd}
 & a_0 \arrow[r] & b_0 & \\
f(a_0,a_0) & a_1 \arrow[l, dotted] & b_1 \arrow[l] \arrow[r, dotted] & f(b_0,b_0) \\
 & \vdots & \vdots & \\
f(a_{n-2},a_{n-2}) & a_{n-1} \arrow[l, dotted] & b_{n-1} \arrow[l] \arrow[r, dotted] & f(b_{n-2},b_{n-2}) \\
f(a_{n-1},a_{n-1}) & a_n \arrow[l, dotted] \arrow[r] & b_n \arrow[r, dotted] & f(b_{n-1},b_{n-1})
\end{tikzcd}
$$

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

How about occurrence checks? Postpone!

Union-Find

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

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

Conclusion

Summary

What is the meaning of constraints?

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

Except where otherwise noted, this work is licensed under