Lecture 9: Constraint Resolution

CS4200 Compiler Construction

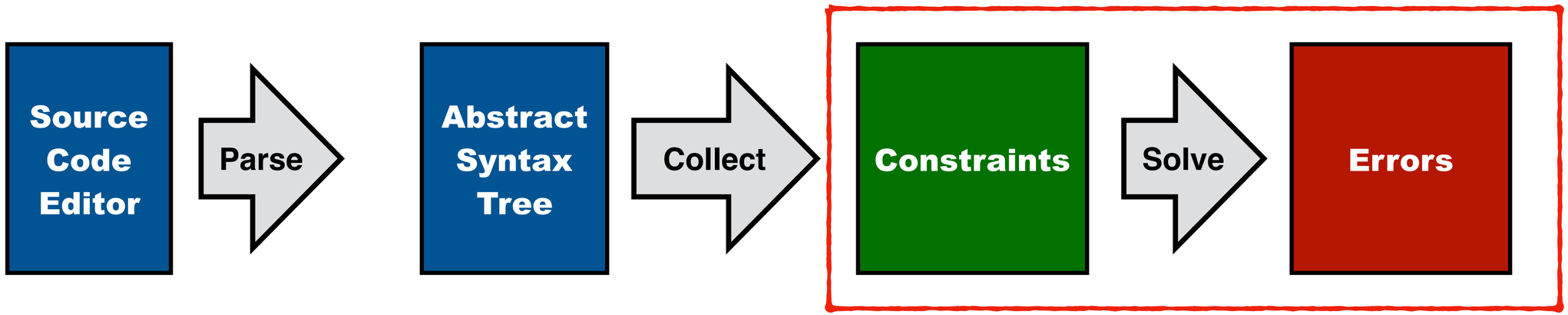
Hendrik van Antwerpen

TU Delft

September 2018

TU Delft

This lecture



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.

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

Separating type checking into constraint generation and constraint solving provides more declarative definition of type checkers. This paper introduces a constraint language integrating name resolution into constraint resolution through scope graph constraints.

This is the basis for the design of the NaBL2 static semantics specification language.

PEPM 2016

https://doi.org/10.1145/2847538.2847543

A Constraint Language for Static Semantic Analysis based on Scope Graphs

Hendrik van Antwerpen
Delft University of Technology
h.vanantwerpen@student.tudelft.nl

Pierre Néron
Delft University of Technology
p.j.m.neron@tudelft.nl

Andrew Tolmach
Portland State University
tolmach@pdx.edu

Eelco Visser
Delft University of Technology
visser@acm.org

Guido Wachsmuth
Delft University of Technology
guwac@acm.org

Abstract

In previous work, we introduced *scope graphs* as a formalism for describing program binding structure and performing name resolution in an AST-independent way. In this paper, we show how to use scope graphs to build static semantic analyzers. We use *constraints* extracted from the AST to specify facts about binding, typing, and initialization. We treat name and type resolution as separate building blocks, but our approach can handle language constructs—such as record field access—for which binding and typing are mutually dependent. We also refine and extend our previous scope graph theory to address practical concerns including ambiguity checking and support for a wider range of scope relationships. We describe the details of constraint generation for a model language that illustrates many of the interesting static analysis issues associated with modules and records.

1. Introduction

Language workbenches [6] are tools that support the implementation of full-fledged programming environments for (domain-specific) programming languages. Ongoing research investigates how to reduce implementation effort by factoring out language-independent implementation concerns and providing high-level meta-languages for the specification of syntactic and semantic aspects of a language [18]. Such meta-languages should (i) have a clear and clean underlying theory; (ii) handle a broad range of common language features; (iii) be declarative, but be realizable by practical algorithms and tools; (iv) be factored into language-specific and language-independent parts, to maximize re-use; and (v) apply to erroneous programs as well as to correct ones.

In recent work we show how name resolution for lexically-scoped languages can be formalized in a way that meets these criteria [14]. The name binding structure of a program is captured in a *scope graph* which records identifier declarations and references and their scoping relationships, while abstracting away program details. Its basic building blocks are *scopes*, which correspond to sets of program points that behave uniformly with respect to resolution. A scope contains identifier declarations and references, each tagged with its position in the original AST. Scopes can be connected by edges representing lexical nesting or import of named collections of declarations such as modules or records. A scope graph is constructed from the program AST using a language-dependent traversal, but thereafter, it can be processed in a largely language-independent way. A *resolution calculus* gives a formal definition of what it means for a reference to resolve to a declaration. Resolutions are described as paths in the scope graph obeying certain (language-specific) criteria; a given reference may resolve to one, none, or many declarations. A derived *resolution algorithm* computes the set of declarations to which each reference resolves, and is sound and complete with respect to the calculus.

In this paper, we refine and extend the scope graph framework of [14] to construct a full framework for static semantic analysis. In essence, this involves uniting a type checker with our existing name resolution machinery. Ideally, we would like to keep these two aspects separated as much as possible for maximum modularity. And indeed, for many language constructs, a simple two-stage approach—name resolution using the scope graph followed by a separate type checking step–would work. But the full story is more complicated, because sometimes name resolution also depends on type resolution. For example, in a language that uses dot notation for object field projection, determining the resolution of `x` in the expression `r.x` requires first determining the object type of `r`, which in turn requires name resolution again. Thus, our framework requires a unified mechanism for expressing and solving arbitrarily interdependent naming and typing resolution problems.

To address this challenge, we base our framework on a language of *constraints*. Term equality constraints are a standard choice for describing type inference problems while abstracting away from the details of an AST in a particular language. Adopting constraints to describe both typing and scoping requirements has the advantage of uniform notation, and, more importantly, provides a clean way to combine naming and typing problems. In particular, we extend our previous work to support *incomplete* scope graphs, which correspond to constraint sets with (as yet) unresolved variables.

Our new framework continues to satisfy the criteria outlined above. (i) The resolution calculus and standard term equality constraint theory provide a solid language-independent theory for name and type resolution. (ii) Our framework supports type checking and inference for statically typed, monomorphic languages with user-defined types, and can also express uniqueness and completeness requirements on declarations and initializers. The framework inherits from scope graphs the ability to model a broad range of binding patterns, including many variants of lexical scoping, records, and modules. (iii) The constraint language has a declarative semantics given by a constraint satisfaction relation, which employs the resolution calculus to define the semantics of name resolution relative to a scope graph. We define a constraint resolution algorithm based on our previous name resolution algorithm,

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.

Prototype of Statix is available in Spoofax HEAD, but not ready for use in project yet.

The future

OOPSLA 2018

To appear

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.

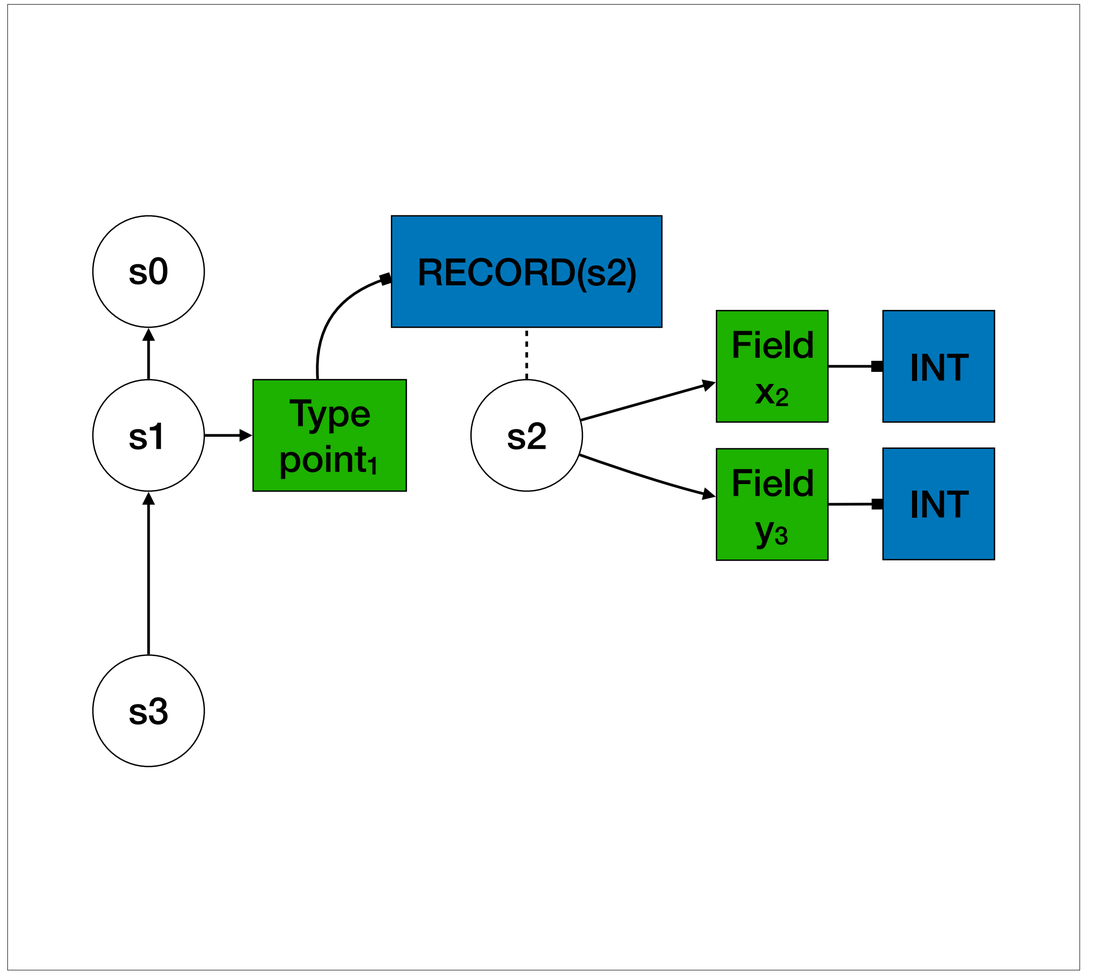
2475-1421/2018/11-ART114
https://doi.org/10.1145/3276484

Tiger in NaBL2

Record Definitions

```
                                                                    s0
let
  type point1 = { x2 : int, y3 : int }     s2
  var p4 := point5{ x6 = 4, y7 = 5 }                                s1
in                                                                  s3
  p8.x9
end
```

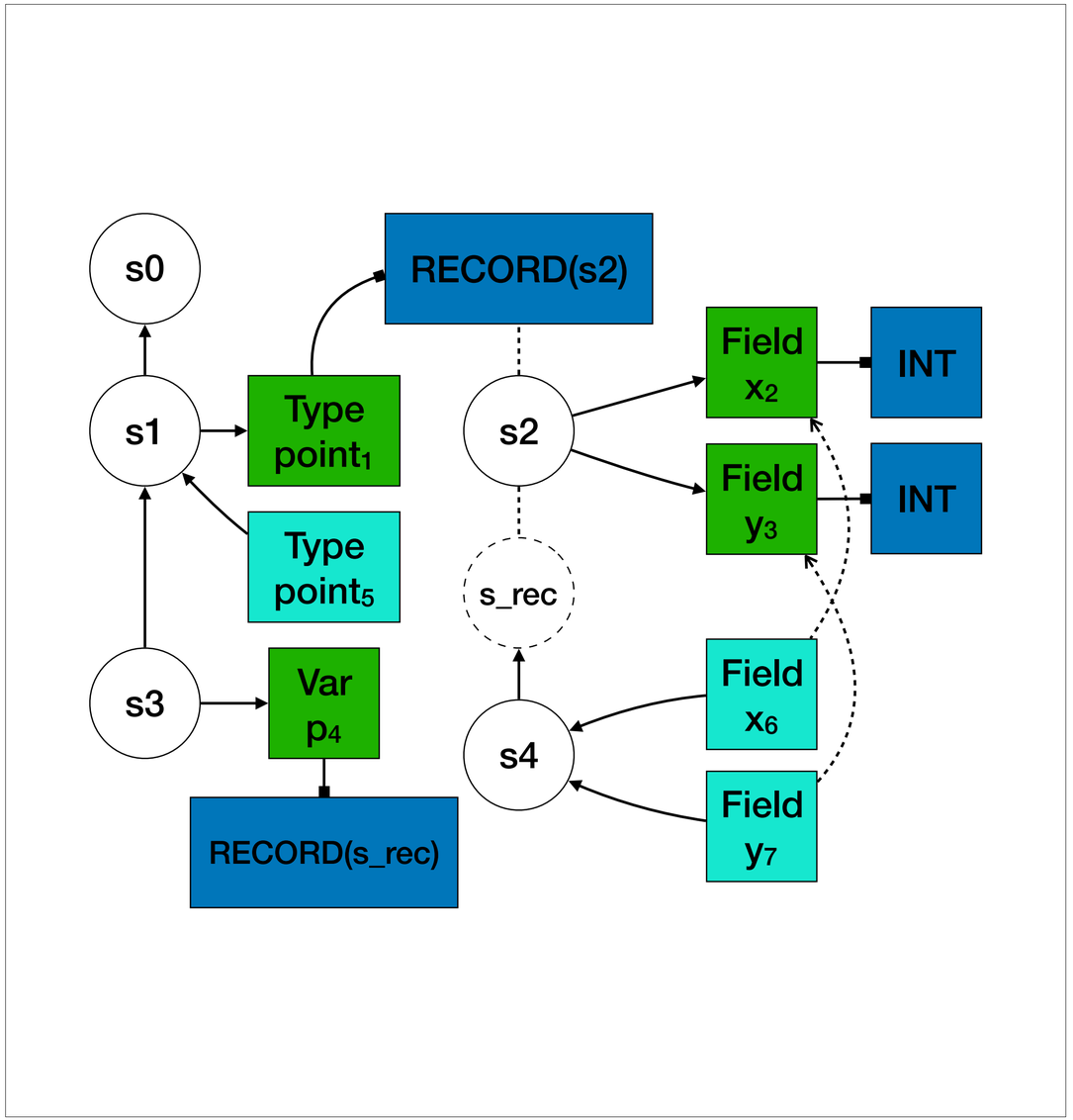
```
[[ RecordTy(fields) ^ (s) : ty ]] :=
    new s_rec,
    ty == RECORD(s_rec),
    NIL() <! ty,
    distinct/name D(s_rec)/Field,
    Map2[[ fields ^ (s_rec, s) ]].

[[ Field(x, t) ^ (s_rec, s_outer) ]] :=
    Field{x} <- s_rec,
    Field{x} : ty !,
    [[ t ^ (s_outer) : ty ]].
```



Record Creation

```
                                                    s0
let
  type point1 = { x2 : int, y3 : int }     s2
  var p4 := point5{ x6 = 4, y7 = 5 }       s4      s1
in                                                 s3
  p8.x9
end
```

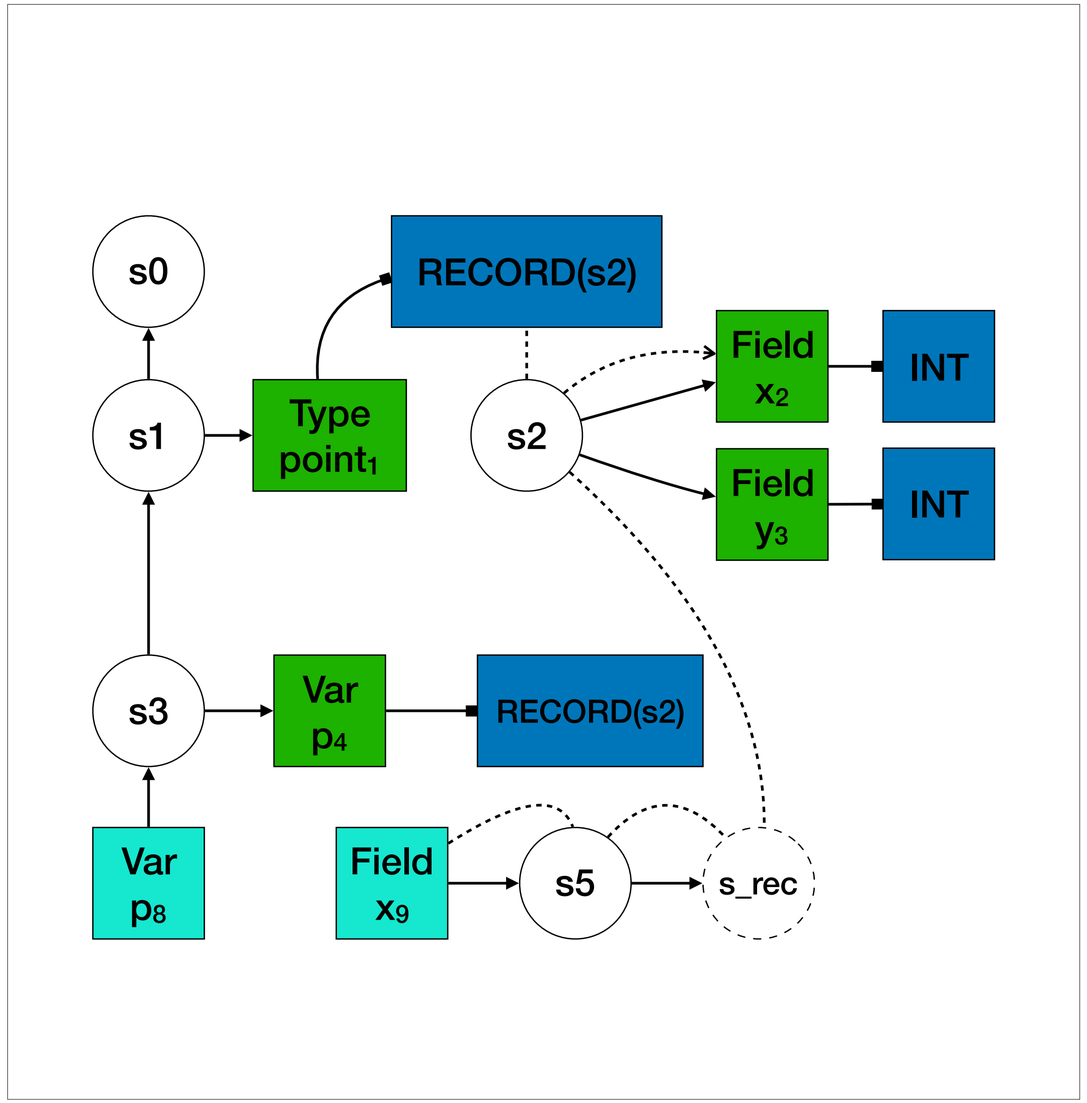
```
[[ r@Record(t, inits) ^ (s) : ty ]] :=
    [[ t ^ (s) : ty ]],
    ty == RECORD(s_rec),
    new s_use, s_use -I-> s_rec,
    D(s_rec)/Field subseteq/name
        R(s_use)/Field,
    distinct/name R(s_use)/Field,
    Map2[[ inits ^ (s_use, s) ]].

[[ InitField(x, e) ^ (s_use, s) ]] :=
    Field{x} -> s_use,
    Field{x} |-> d,
    d : ty1,
    [[ e ^ (s) : ty2 ]],
    ty2 <? ty1.
```



Record Field Access

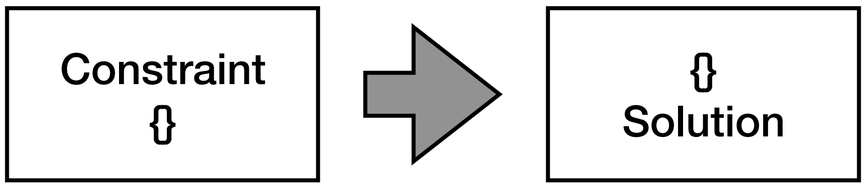
```
                                                              s0
let
  type point1 = { x2 : int, y3 : int }     s2
  var p4 := point5{ x6 = 4, y7 = 5 }       s4          s1
in                                                     s3
  p8.x9   s5
end
```

```
[[ FieldVar(e, f) ^ (s) : ty ]] :=
    [[ e ^ (s) : ty_e ]],
    ty_e == RECORD(s_rec),
    new s_use, s_use -I-> s_rec,
    Field{f} -> s_use,
    Field{f} |-> d,
    d : ty.
```



Solving Constraints

Solving by Rewriting



Solving by Rewriting

```
<C; G, s> ---> <C; G, s>
```

```
      <t == u, C; G, s> ---> <C; G, s'>      where unify(s,t,u) = s'

  <s1 -L-> s2, C; G, s> ---> <C; G', s>      where G + {s1 -L-> s2} = G'

<Ns{x@i} |-> t, C; G, s> ---> <t == d; G, s> where resolve(G,Ns{x@i}) = d
```

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

Solving by Rewriting

Solver = rewrite system

- Rewrite a constraints set + solution
- Simplifying and eliminating constraints
  - Constraint selecting is non-deterministic
  - Partial order is enforced 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
- Conjecture: Partly true for NaBL2
  - Up to variable and scope names
  - Only if all constraints are reduced

Constraint 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?

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

When are constraints satisfied?

- Formally described by the semantics
- Written as $G, s \models C$
- Satisfied in a model (substitution + scope graph)
- Describes for every type of constraint when it is satisfied

Semantics of (a Subset of) NaBL2 Constraints

Constraint syntax

```
C = t == t        // equality
  | r |-> d       // resolution
  | C /\ C        // conjunction
```

Constraint semantics

$$
\begin{array}{ll}
G,s \models t == u & \text{if } s(t) = s(u) \\
\\
G,s \models r \mid\!\!\to d & \text{if } s(r) = \mathrm{Var}\{x\ @i\} \\
 & \text{and } s(d) = \mathrm{Var}\{x\ @j\} \\
 & \text{and } \mathrm{Var}\{x\ @i\} \text{ resolves to } \mathrm{Var}\{x\ @j\} \text{ in } G \\
\\
G,s \models C_1 \wedge C_2 & \text{if } G,s \models C_1 \text{ and } G,s \models C_2
\end{array}
$$

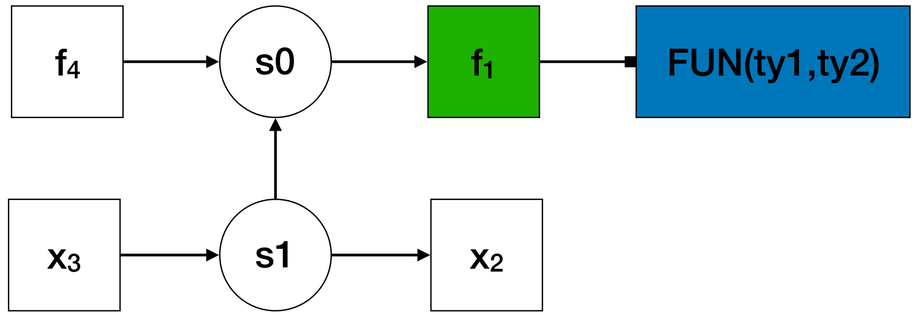
Using the Semantics

```
let
  function f1(x2 : int) : int =
    x3 + 1
in
  f4(14)
end
```

```
ty1 == INT()
INT() == INT()
Var{x @3} |-> d1
ty2 == INT()
Var{f @4} |-> d2
ty3 == FUN(ty4,ty5)
ty4 == INT()
…
```

```
s = { ty1 -> INT(),
      ty2 -> INT(),
      ty3 -> FUN(INT(),ty5),
      ty4 -> INT(),
      d1  -> Var{x @2},
      d2  -> Var{f @1}
    }
```

```
G,s ⊨ t == u
    if s(t) = s(u)

G,s ⊨ r |-> d
    if  s(r) = Var{x @i}
    and s(d) = Var{x @j}
    and Var{x @i} resolves to Var{x @j} in G

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



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, s>, then $G, s \models C$
- Completeness:
  - If an s exists such that $G, s \models C$, then the solver returns it
  - If no such s exists, the solver fails

Principality

- The solver finds the most general s

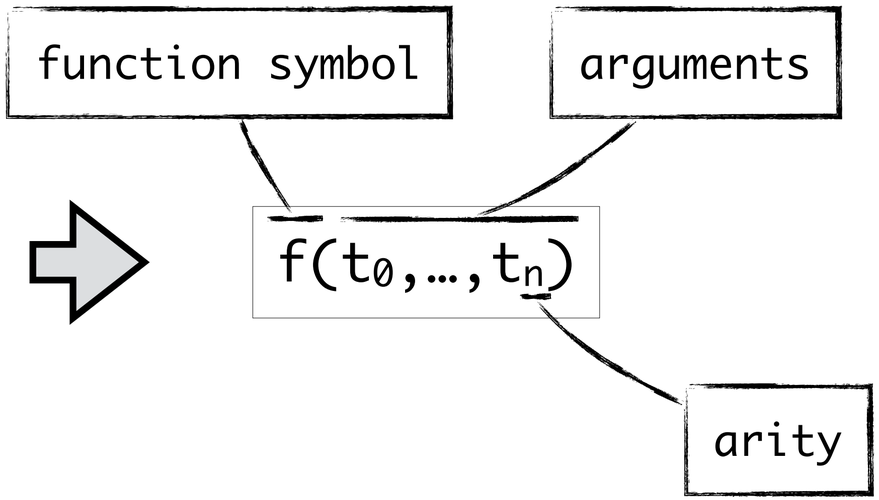
Term Equality & Unification

Syntactic Terms

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

Generic Terms

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



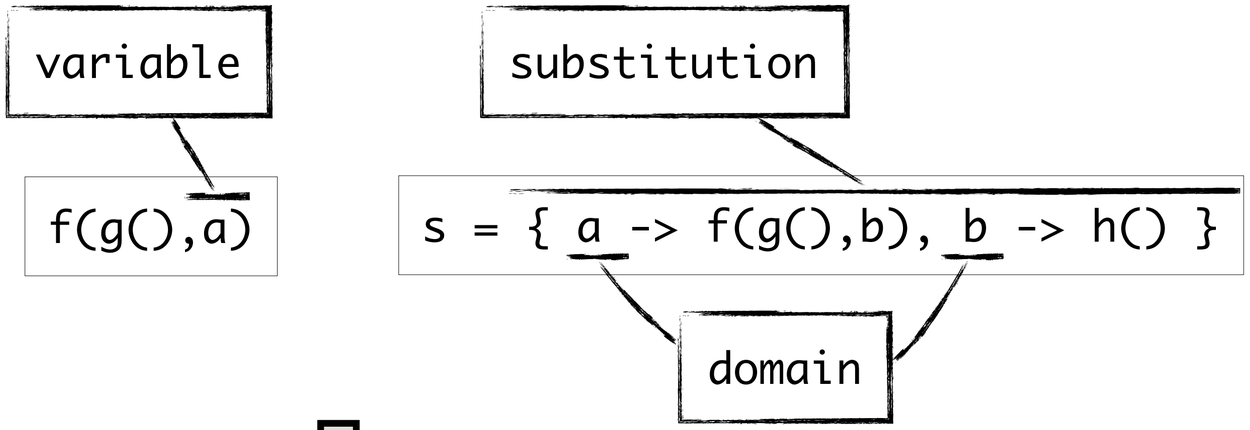
Syntactic Equality

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



$$
\begin{aligned}
s(a) &= t && \text{if } \{ a \to t \} \text{ in } s \\
s(a) &= a && \text{otherwise} \\
s(f(t_0,\ldots,t_n)) &= f(s(t_0),\ldots,s(t_n))
\end{aligned}
$$

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

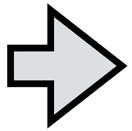
ground term: a term without variables

Unifiers

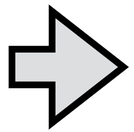
terms	t, u
functions	f, g, h
variables	a, b, c
substitution	s

unifier: a substitution that makes terms equal

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

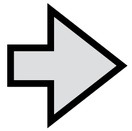


a -> h()
b -> g()

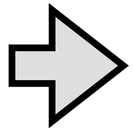


f(h(),g()) == f(h(),g())

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

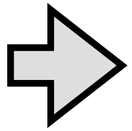


a -> f(h())
b -> h()



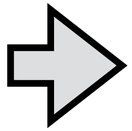
g(f(h()),f(h())) == g(f(h()),f(h()))

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



no unifier, f != g

f(b,b) == b

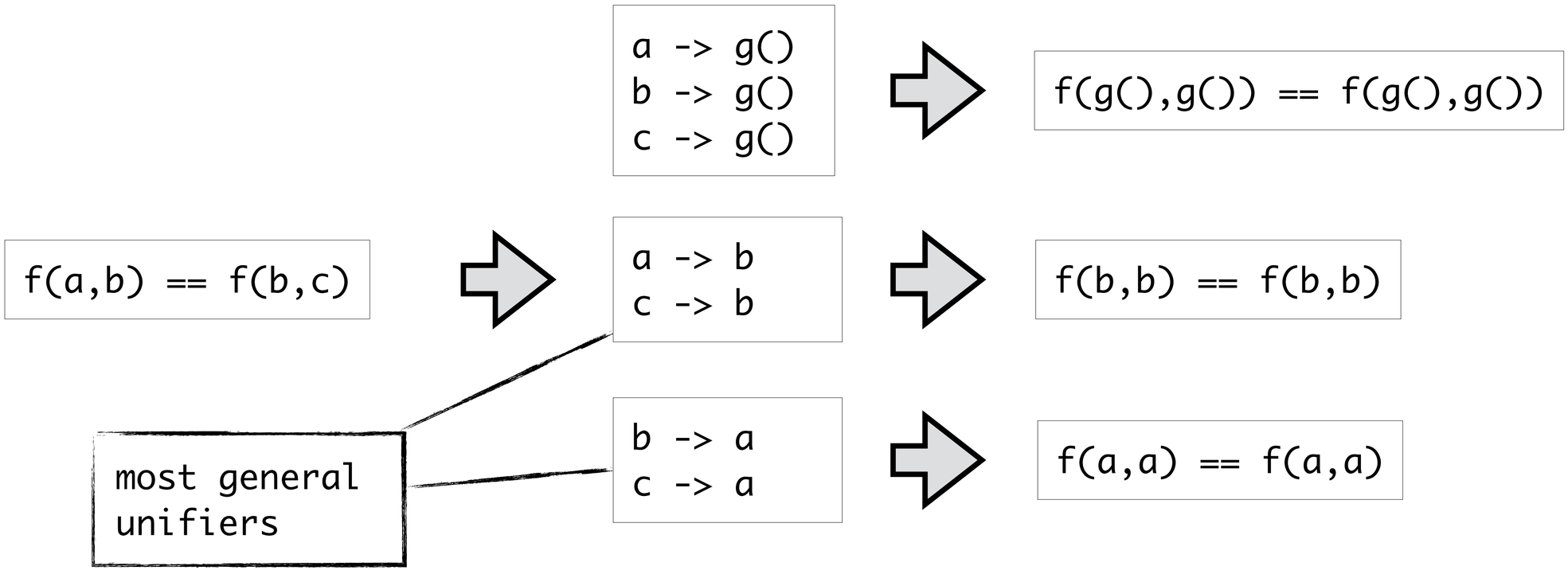


b -> f(b,b)

not idempotent

Most General Unifiers

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



Most General Unifiers

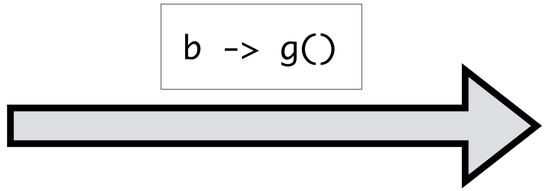
```
terms         t, u
functions     f, g, h
variables     a, b, c
substitution  s
```

every unifier is an instance of a most general unifier

(implicit) identity case

```
a -> b
b -> b
c -> b
```

b -> g()

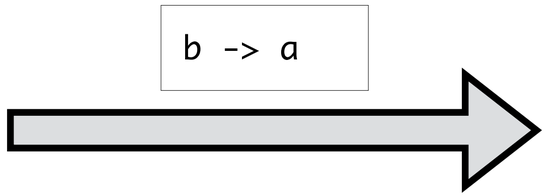


```
a -> g()
b -> g()
c -> g()
```

most general unifiers are related by renaming substitutions

```
a -> b
b -> b
c -> b
```

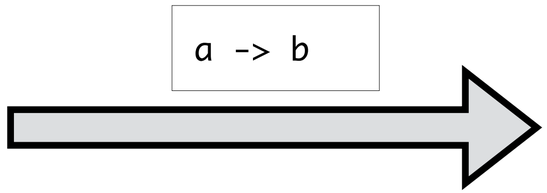
b -> a



```
a -> a
b -> a
c -> a
```

```
a -> a
b -> a
c -> a
```

a -> b



```
a -> b
b -> b
c -> b
```

Unification

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

```
global s
def unify(t, u):
  if t is a variable:
    t := s(t)
  if u is a variable:
    u := s(u)
  if 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:
    s += { s -> t }
```

t == a
instantiate variable

u == b
instantiate variable

equal terms

t == k(t₀,...,t₅), u == k(u₀,...,u₅)
matching terms

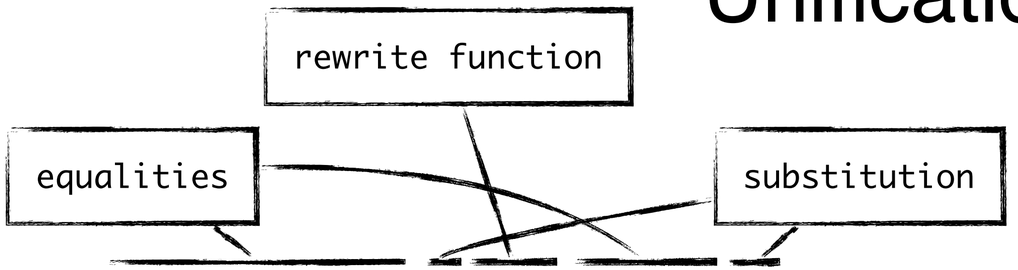
t == k(t₀,...,t₅), u == f(u₀,...,u₃)
mismatching terms

t == k(t₀,...,t₅), u == b
mismatching terms

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

t == a, u == k(u₀,...,u₅)
extend unifier

Unification (rewriting)




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

```
{ t == u } U C; s ---> C' U C; s'

{ t == t } U C; s --->
    C; s

{ f(t0,…,tn) == f(u0,…,un) } U C; s --->
    { t0 == u0, …, tn == un } U C; s

{ f(t0,…,tn) == g(u0,…,um) } U C; s --->
    FAIL
  if f != g

{ a == t } U C; s --->
    C{a -> t}; s{a -> t}
  if a ∉ vars(t)

{ a == t } U C; s --->
    FAIL
  if a ∈ vars(t)

{ t == a } U C; s --->
    { a == t } U C; s
```

equalities C	substitution s
{ f() == f() }	{}
{}	{}
{ f(g()) == f(h()) }	{}
{ g() == h() }	FAIL
{ g(b,a) == g(a,h(b)) }	{}
{ b == a, a == h(b) }	{b -> a}
{ a == h(a) }	FAIL
{ f(g(),h(a)) == f(b,h(b)) }	{}
{ g() == b, h(a) == h(b) }	{}
{ b == g(), h(a) == h(b) }	{b -> g()}
{ h(a) == h(g()) }	{b -> g()}
{ a == g() }	{b -> g(),a -> g()}
{}	{b -> g(),a -> g()}

X

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

TU Delft

Complexity of Unification

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

Space complexity

- Exponential
- Representation of unifier

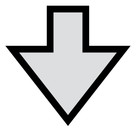
Time complexity

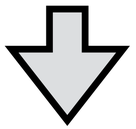
- Exponential
- Recursive calls on terms

Solution

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

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





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

fully applied

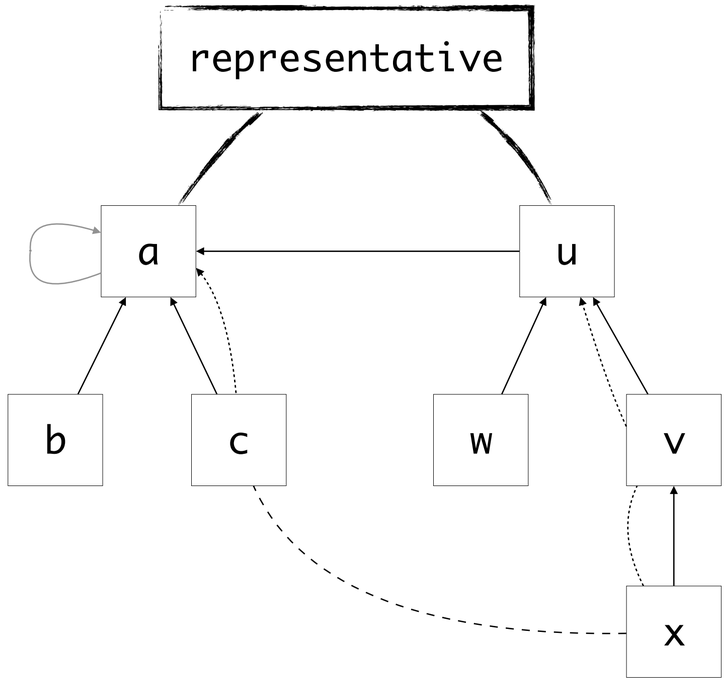
$a_1 \rightarrow f(a_0,a_0)$
$a_2 \rightarrow f(a_1,a_1)$
$a_i \rightarrow \ldots$ 3 subterms …
$b_1 \rightarrow f(a_0,a_0)$
$b_2 \rightarrow f(a_1,a_1)$
$b_i \rightarrow \ldots$ 3 subterms …

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
```

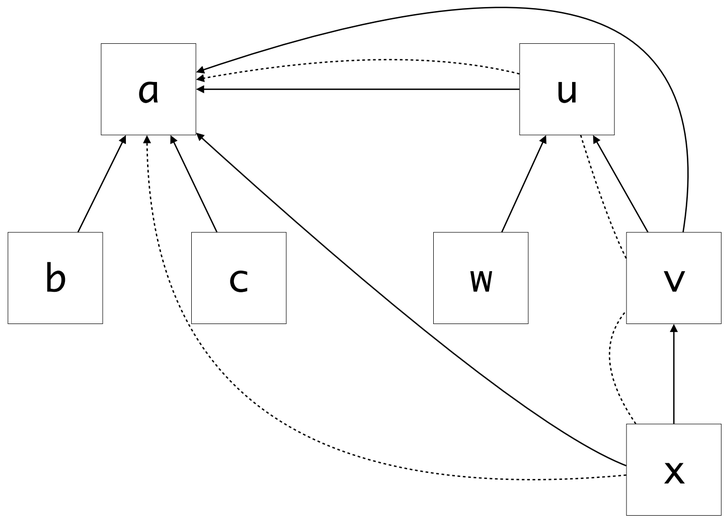
```
a == b
c == a
u == w
v == u
x == v
x == c
```



Path Compression

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

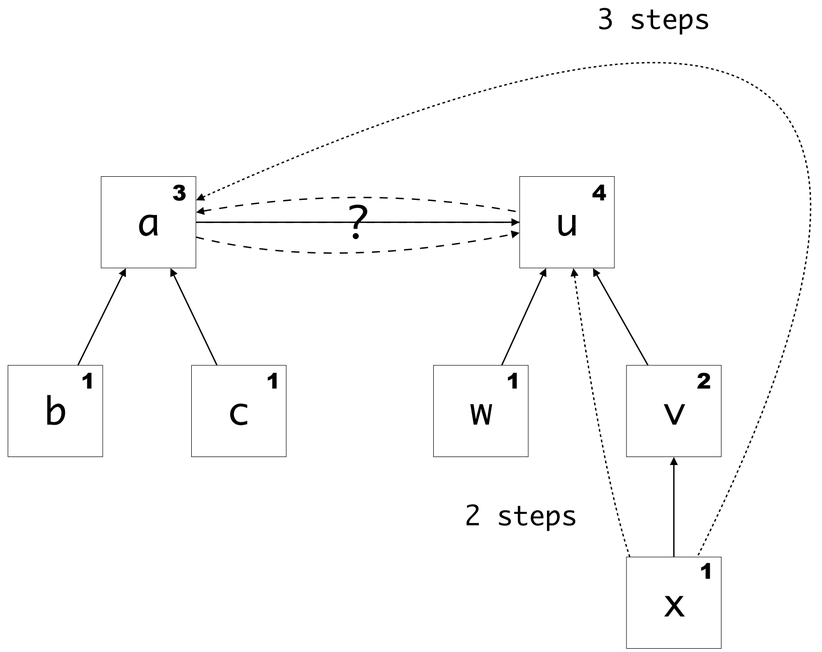


Tree Balancing

```
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)
```

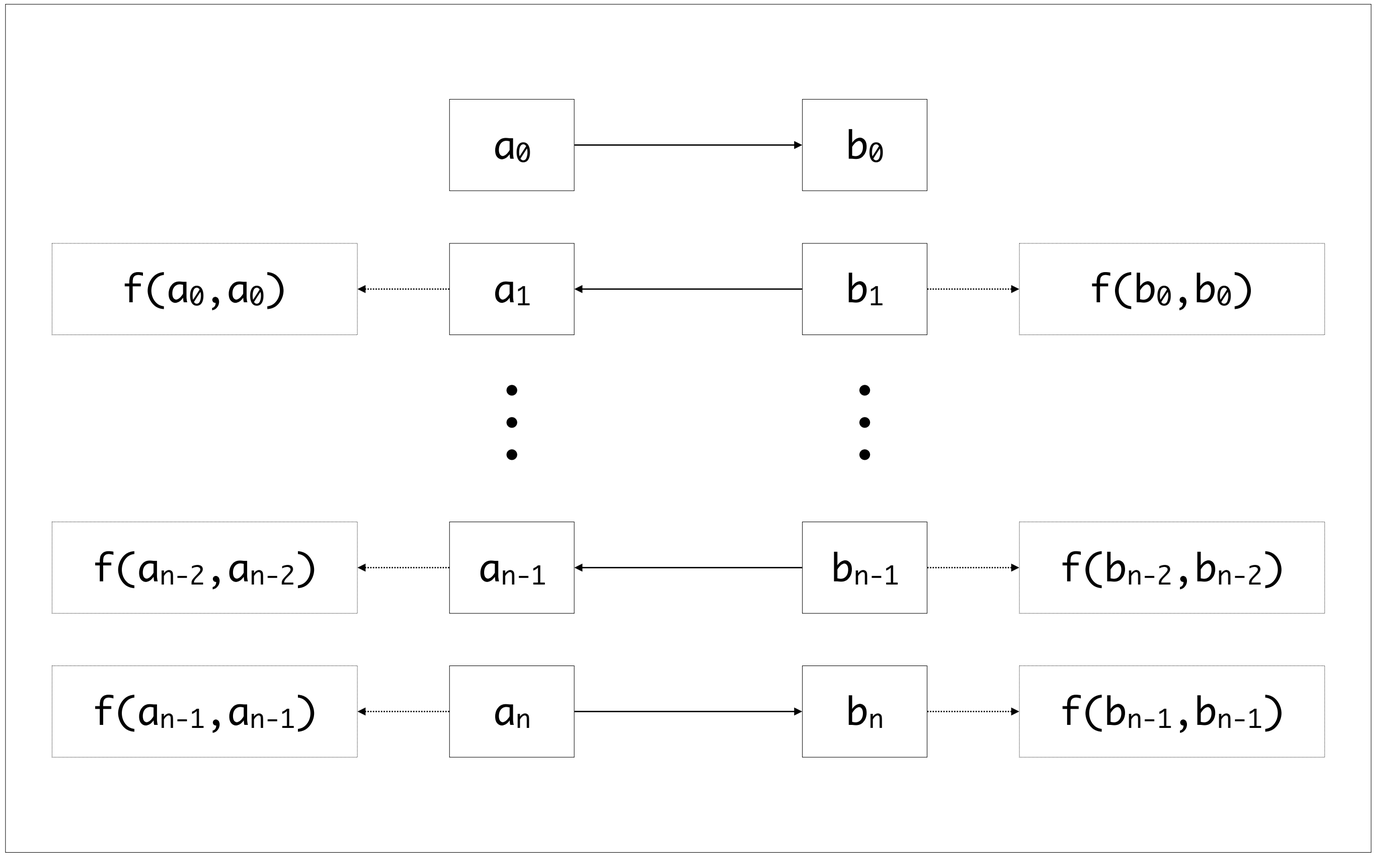
…

x == c



The Complex Case

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



$$a_n == b_n$$

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

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

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

$$\vdots$$

$$a_1 == b_1 \qquad a_1 == b_1$$

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

$$a_0 == b_0 \qquad a_0 == b_0$$

How about occurrence checks? Postpone!

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 in time (technically inverse Ackermann)
- Easy to extract triangular unifier from graph
- Postpone occurrence checks to prevent traversing (potentially) large terms

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

Conclusion

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 implements 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