Declare Your Language

Chapter 7: Constraint Resolution I

Hendrik van Antwerpen

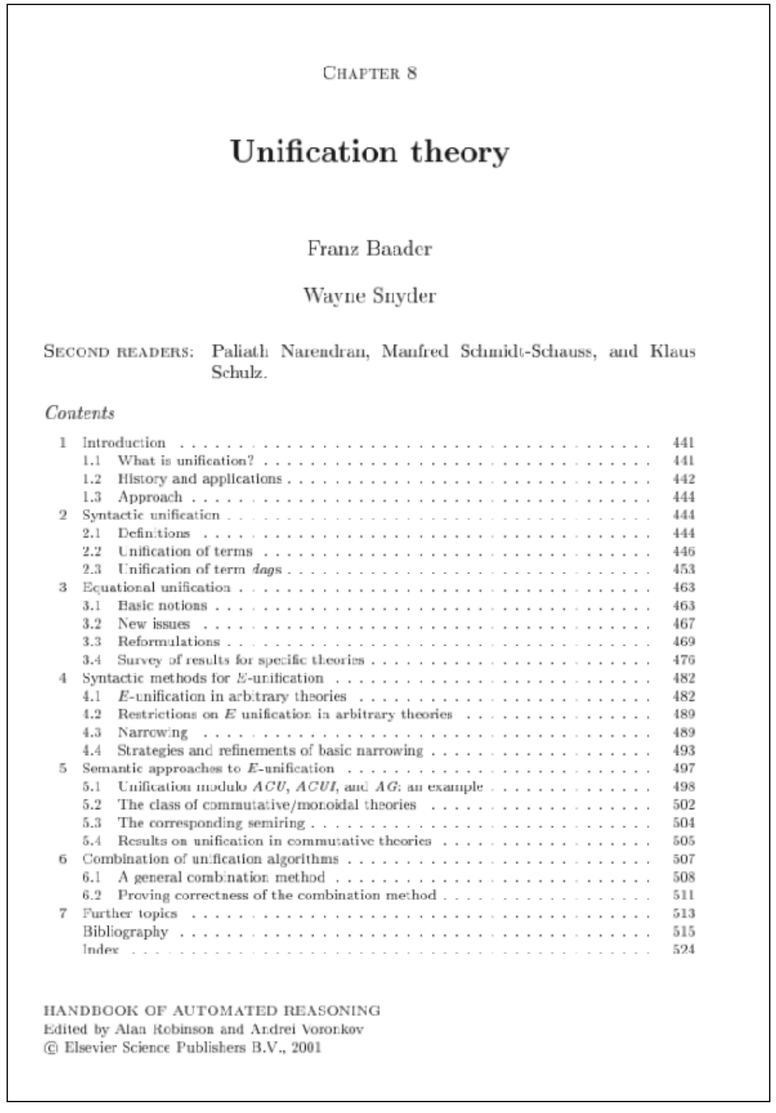
IN4303 Compiler Construction
TU Delft
September 2017

TU Delft

Reading Material

Unification

Baader, F., and W. Snyder
"Unification Theory”
Ch. 8 of Handbook of Automated Deduction
Springer Verlag, Berlin (2001)



CHAPTER 8

Unification theory

Franz Baader

Wayne Snyder

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

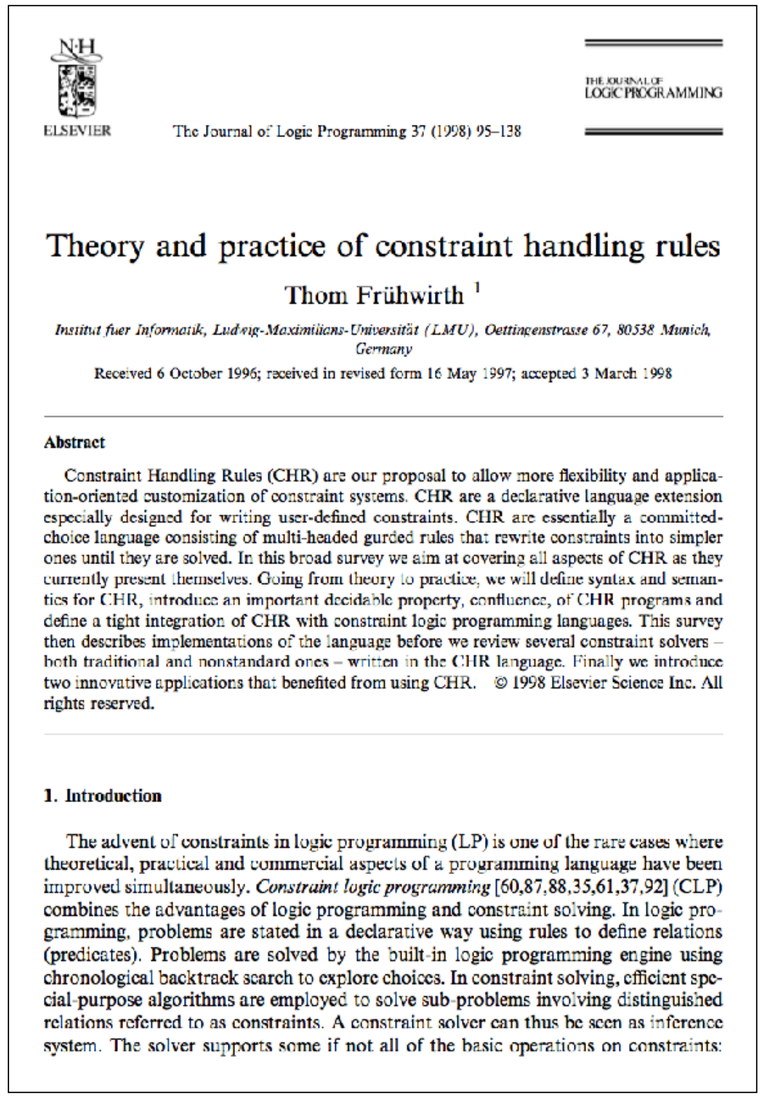
Contents

HANDBOOK OF AUTOMATED REASONING
Edited by Alan Robinson and Andrei Voronkov

http://www.cs.bu.edu/~snyder/publications/UnifChapter.pdf

Constraint Handling Rules

Thom Frühwirth
“Theory and practice of constraint handling rules”
In The Journal of Logic Programming
Volume 37, Issues 1–3, 1998, Pages 95-138



ELSEVIER The Journal of Logic Programming 37 (1998) 95–138

THE JOURNAL OF LOGIC PROGRAMMING

Theory and practice of constraint handling rules

Thom Frühwirth [1]

Institut fuer Informatik, Ludwig-Maximilians-Universität (LMU), Oettingenstrasse 67, 80538 Munich, Germany

Received 6 October 1996; received in revised form 16 May 1997; accepted 3 March 1998

Abstract

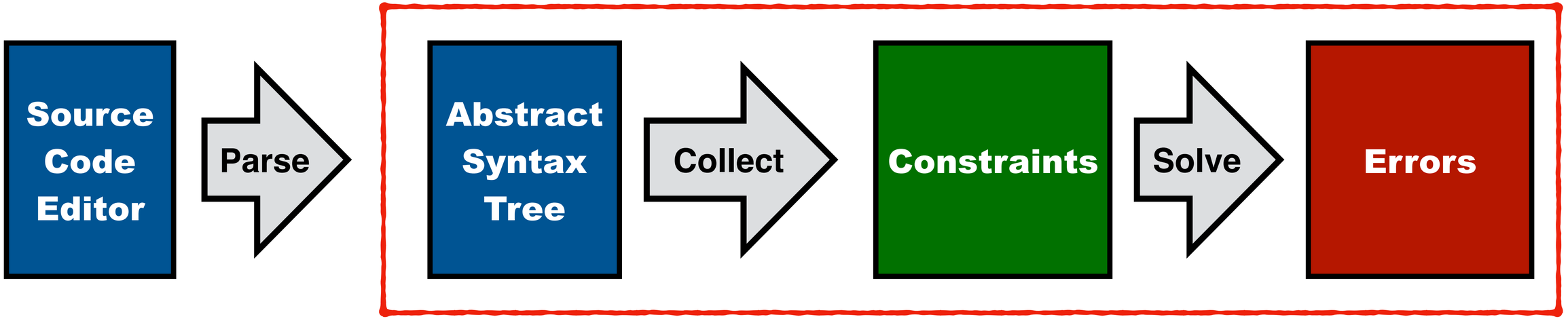
Constraint Handling Rules (CHR) are our proposal to allow more flexibility and application-oriented customization of constraint systems. CHR are a declarative language extension especially designed for writing user-defined constraints. CHR are essentially a committed-choice language consisting of multi-headed gurded rules that rewrite constraints into simpler ones until they are solved. In this broad survey we aim at covering all aspects of CHR as they currently present themselves. Going from theory to practice, we will define syntax and semantics for CHR, introduce an important decidable property, confluence, of CHR programs and define a tight integration of CHR with constraint logic programming languages. This survey then describes implementations of the language before we review several constraint solvers – both traditional and nonstandard ones – written in the CHR language. Finally we introduce two innovative applications that benefited from using CHR.

1. Introduction

The advent of constraints in logic programming (LP) is one of the rare cases where theoretical, practical and commercial aspects of a programming language have been improved simultaneously. *Constraint logic programming* [60,87,88,35,61,37,92] (CLP) combines the advantages of logic programming and constraint solving. In logic programming, problems are stated in a declarative way using rules to define relations (predicates). Problems are solved by the built-in logic programming engine using chronological backtrack search to explore choices. In constraint solving, efficient special-purpose algorithms are employed to solve sub-problems involving distinguished relations referred to as constraints. A constraint solver can thus be seen as inference system. The solver supports some if not all of the basic operations on constraints:

http://www.sciencedirect.com/science/article/pii/S0743106698100055

Type Checking with Constraints



language specific

language independent

Type Checking with Constraints

What are type checkers for?

- Resolve names and compute or check types
- Report error messages
- Provide a representation of name and type information

Constraints separate concerns

- Declarative specification
- Separate solver algorithm

What is the meaning of constraints?

- Until now, we have an intuitive understanding
- How can we be precise about the meaning of constraints?

How can we implement a constraint solver?

- Which techniques are available?
- When is a solver correct?

Term Equality & Unification

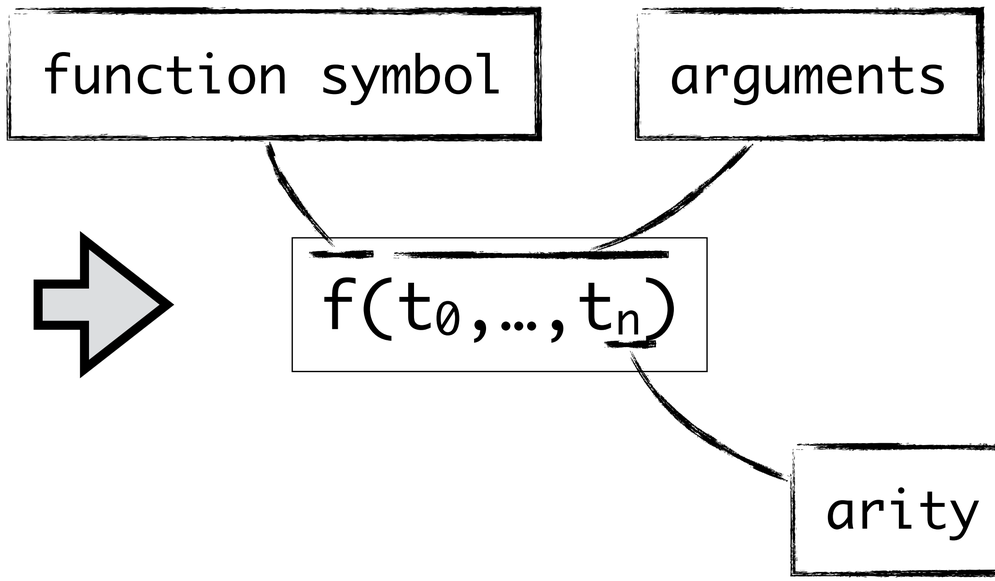
TU Delft

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

- $f = g$ (including arity)
- $t_i == u_i$ for every i

Variables and Substitution

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

variable

substitution

f(g(),a)

s = { a -> f(g(),b), b -> h() }

domain

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

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()))

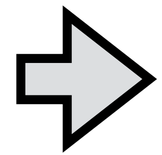
f(a,h()) == g(h(),b) ⇒ no unifier

f(b,b) == b ⇒ b -> f(b,b) ← not idempotent

Most General Unifiers

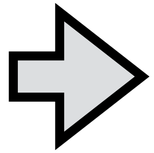
```
terms         t, u
functions     f, g, h
variables     a, b, c
substitution  s
```

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

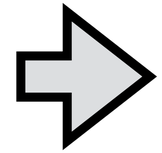


```
f(g(),g()) == f(g(),g())
```

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



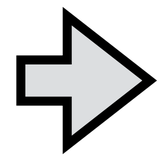
```
a -> b
c -> b
```



```
f(b,b) == f(b,b)
```

most general unifiers

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

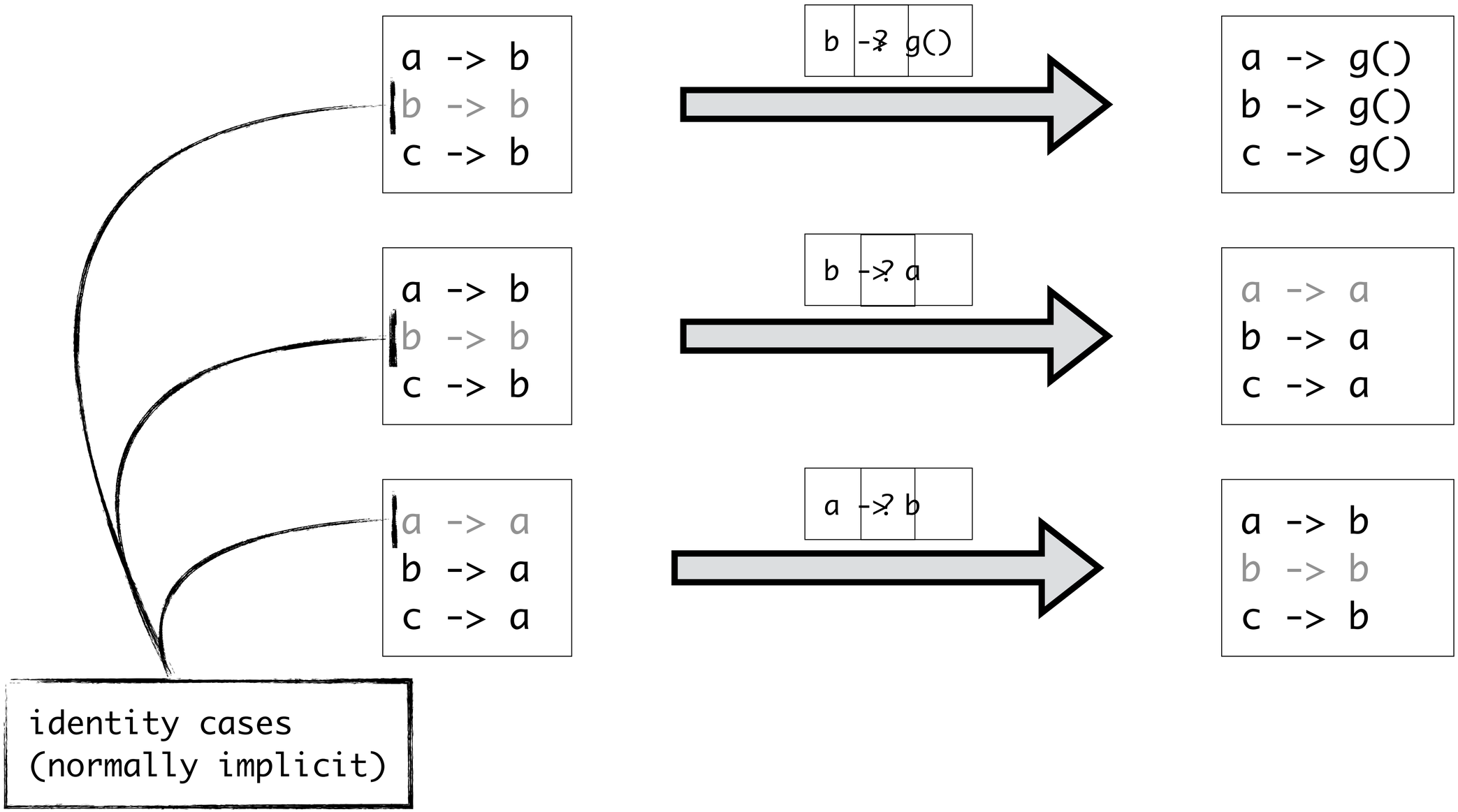


```
f(a,a) == f(a,a)
```

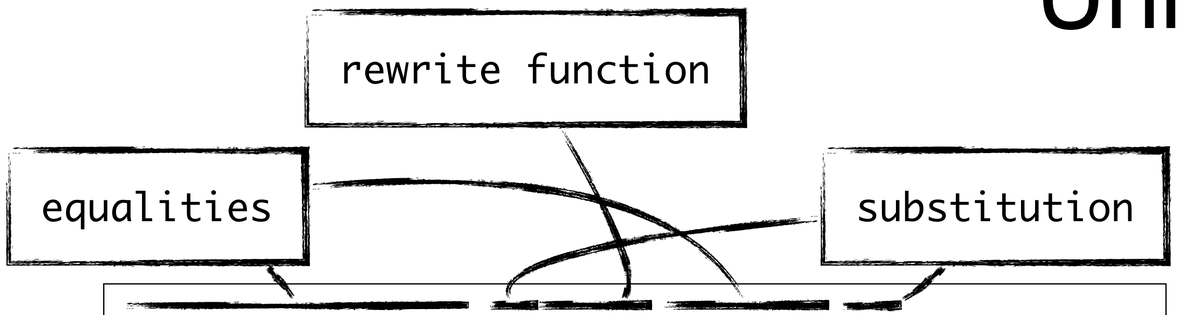
Most General Unifiers

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

every unifier is an instance of a most general unifier



Unification




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

$\{ t == u \} \cup C; s \longrightarrow C' \cup C; s'$

$\{ t == t \} \cup C; s \longrightarrow C; s$

$\{ f(t_0,\ldots,t_n) == f(u_0,\ldots,u_n) \} \cup C; s \longrightarrow \{ t_0 == u_0, \ldots, t_n == u_n \} \cup C; s$

$\{ f(t_0,\ldots,t_n) == g(u_0,\ldots,u_m) \} \cup C; s \longrightarrow FAIL$ if $f \neq g$

$\{ a == t \} \cup C; s \longrightarrow C\{a \to t\}; s\{a \to t\}$ if $a \notin vars(t)$

$\{ a == t \} \cup C; s \longrightarrow FAIL$ if $a \in vars(t)$

$\{ t == a \} \cup C; s \longrightarrow \{ a == t \} \cup C; s$

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()}

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

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

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

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

fully applied

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

triangular

Constraint Semantics

What gives constraints meaning?

A constraint problem

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

- What is the meaning of these constraints?
- In other words, what is a valid solution?
- Can we describe this independent of an algorithm?

Constraint satisfaction

- Formally described by the semantics
- Includes the substitution
- Describes for every constraint when it is satisfied
- Can depend on more than just the substitution
  - Scope graph
  - Sub-type relation
- Written as $s \models C$

Semantics of (a Subset of) NaBL2 Constraints

Constraint syntax

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

Constraint semantics

```
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
```

Using the Semantics

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

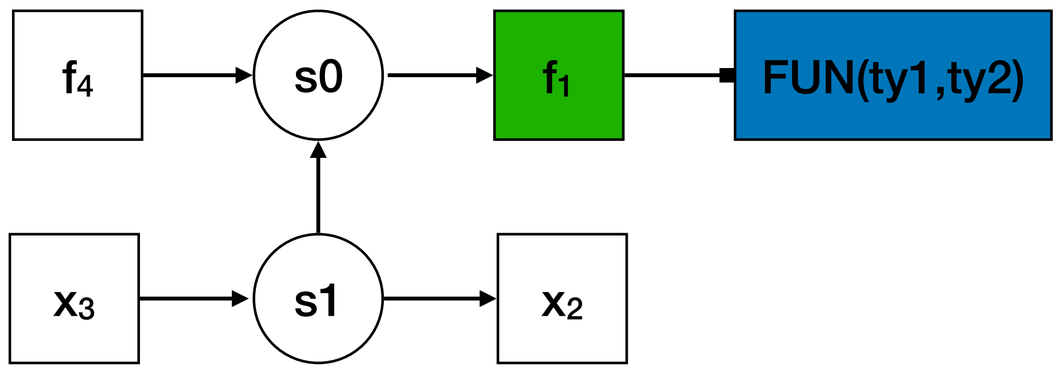
ty1 == INT()
INT() == INT()
Var{x @3} |-> d1
ty2 == INT()
…
Var{f @4} |-> d2
ty3 == FUN(ty4,ty5)
ty4 == INT()
…

```
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

How do we …

- Check that constraints are satisfied?
- Find a substitution `s`?
- Algorithm `solve(G,C) = s`

Soundness

- If solver returns `s`, then `G,s` ⊨ `C`

Completeness

- If an `s` exists such that `G,s` ⊨ `C`, then the solver returns it
- If no such `s` exists, the solver fails

Principality

- The solver finds the most general `s`

Solving Constraints

Constraint Solvers

How can a constraint solver be implemented?

- Case: unification algorithm
  - Rewrite or eliminate constraints
  - Build a solution
  - Use guards to control which rules are chosen
- Other aspects
  - Infer information (if A <: B, and B <: C, then A <: C)
  - Search between alternatives (choose overloaded method)

Common architecture

- Write solver as a set of rules that process constraints
- Use built-in procedures for special cases (unification, name resolution)

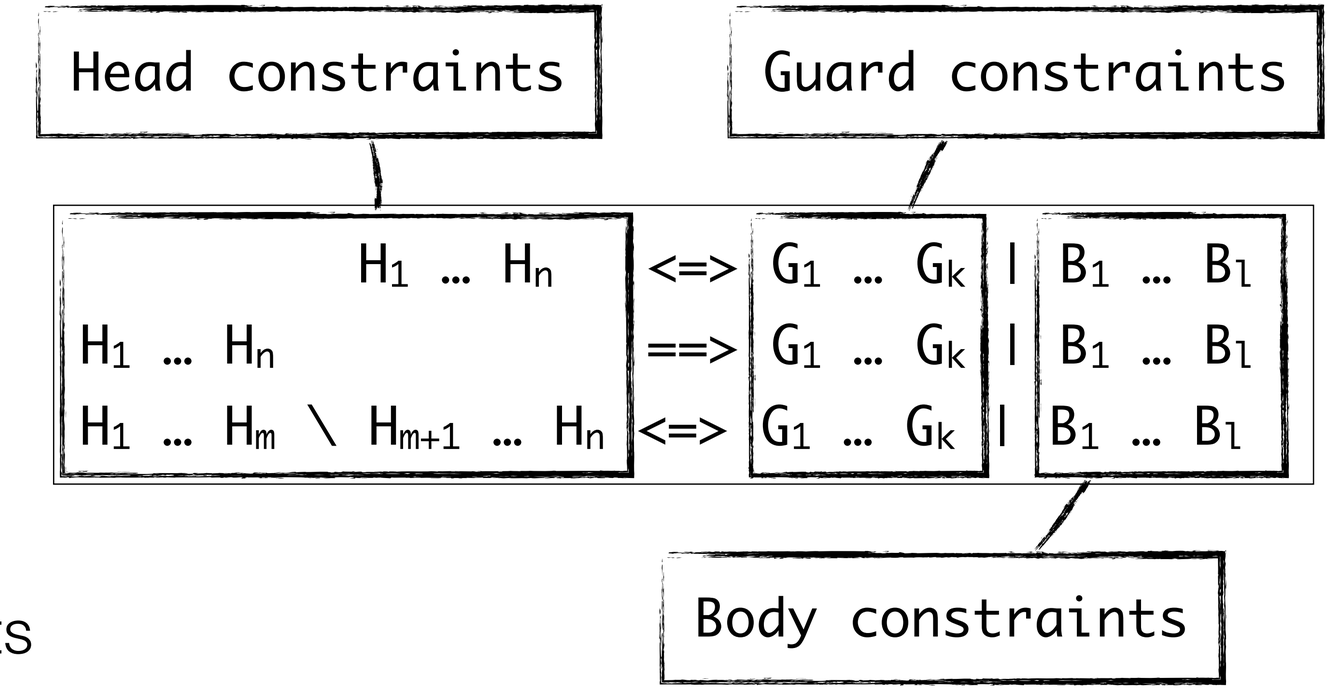
Constraint Handling Rules

General formalism to write constraint solvers

- Generic representation of constraints
- Committed-choice (no search)
- Two types of constraints
  - User-defined CHR constraints: sub(A,B), and(X,Y,Z)
  - Built-in constraints: A == B

Three types of rules

- simplification
  - Rewrite constraints
- propagation
  - Introduce new constraints
- ‘simpagation’
  - Rewrite & introduce constraints



Semantics of Sub-typing Constraints

Constraint syntax

```
C = …
  | t <? t        // sub-type check
```

Constraint semantics

```
ST,G,s ⊨ t <? u        if s(t) is a sub-type of s(u) in ST

                       ST properties:
                       - T is a sub-type of T (reflexivity)
                       - if A sub-type of B, and B sub-type of C,
                         then A sub-type of C (transitivity)
                       - if A sub-type of B, and B sub-type of A,
                         then A = B (anti-symmetry)
```

CHR Rules for Sub-typing Constraints

```
reflexivity    @ sub(A,B)              <=> A = B | true.
transitivity   @ sub(A,B), sub(B,C) ==>          sub(A,C).
anti-symmetry  @ sub(A,B), sub(B,A) <=>          A == B.
```

sub(“X”, “Y”)
sub(“Y”, “Z”)

sub(“X”, “Z”)

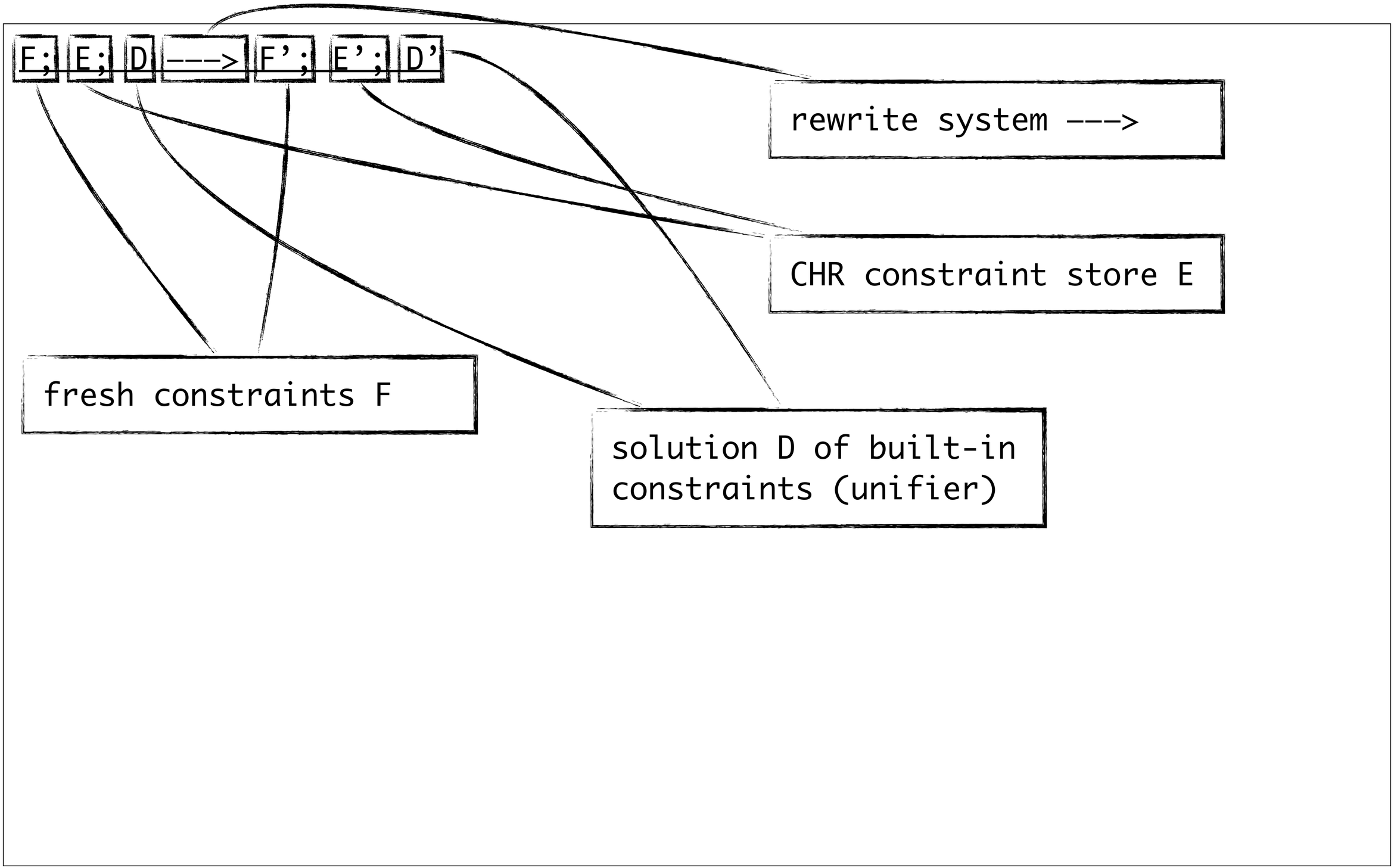
sub(“X”, “X”)

true

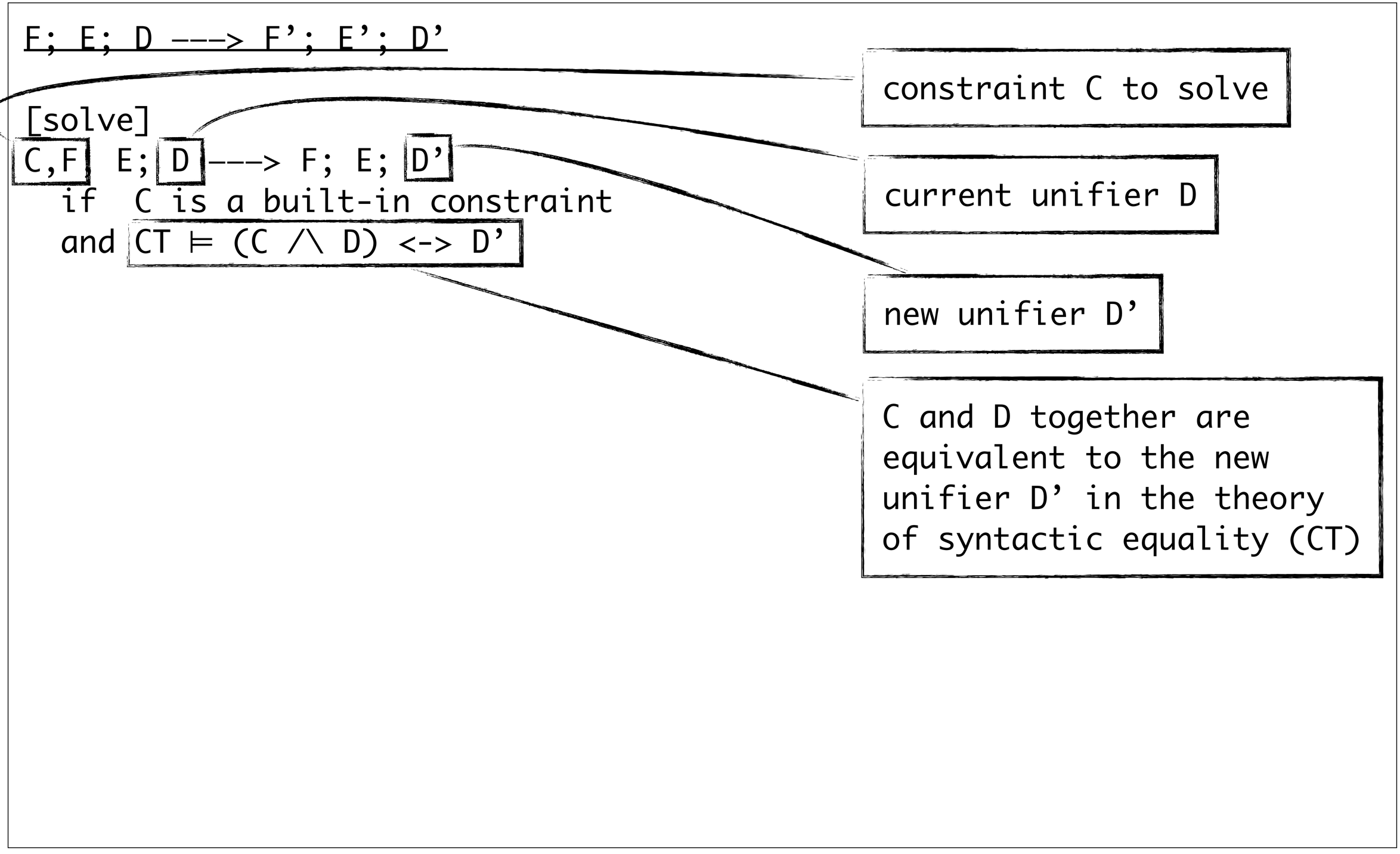
sub(“X”, “Y”)
sub(“Y”, “X”)

“X” == “Y”
inconsistent

Evaluation of CHR



Evaluation of CHR



Evaluation of CHR

E; E; D ---> F'; E'; D'

[solve]
C,F; E; D ---> F; E; D'
  if C is a built-in constraint
  and CT ⊨ (C /\ D) <-> D'

[introduce]
H,F; E; D ---> F; H,E; D'
  if H is a CHR constraint

CHR constraint H to solve

current constraint store E

constraint store updated with H

Evaluation of CHR

F; E; D ---> F'; E'; D'

[solve]
C,F; E; D ---> F; E; D'
  if C is a built-in constraint
  and CT ⊨ (C /\ D) <-> D'

[introduce]
H,F; E; D ---> F; H,E; D'
  if H is a CHR constraint

[simplify]
F; H',E; D ---> B,F; E; (H == H'),D'
  if a rule "H <=> G | B" exists
  and CT ⊨ D -> ∃x.(H = H', G)

matched constraints H'

body constraints are introduced

the simplification rule to apply

guard G must be satisfied given the current unifier D

rule head must match

Evaluation of CHR

```
F; E; D ---> F'; E'; D'

[solve]
C,F; E; D ---> F; E; D'
  if  C is a built-in constraint
  and CT ⊨ (C /\ D) <-> D'

[introduce]
H,F; E; D ---> F; H,E; D'
  if H is a CHR constraint

[simplify]
F; H',E; D ---> B,F; E; (H == H'),D'
  if  a rule "H <=> G | B" exists
  and CT ⊨ D -> ∃x.(H = H', G)

[propagate]
F; H',E; D ---> B,F; H',E; (H == H'),D'
  if  a rule "H ==> G | B" exists
  and CT ⊨ D -> ∃x.(H = H', G)
```

matched constraints H'

body constraints B are added fresh

matched constraints H' are kept

propagation rule to apply

heads match and guard is satisfied in current unifier D

Evaluating Sub-typing Rules

```
reflexivity     @ sub(A,B)           <=> A = B | true.
transitivity    @ sub(A,B), sub(B,C) ==>         sub(A,C).
anti-symmetry   @ sub(A,B), sub(B,A) <=>         A == B.
```

```
[solve]
C,F; E; D ---> F; E; D'
  if  C is a built-in constraint
  and CT ⊨ (C /\ D) <-> D'

[introduce]
H,F; E; D ---> F; H,E; D'
  if H is a CHR constraint

[simplify]
F; H',E; D ---> B,F; E; (H == H'),D'
  if  a rule "H <=> G | B" exists
  and CT ⊨ D -> ∃x.(H = H', G)

[propagate]
F; H',E; D ---> B,F; H',E; (H == H'),D'
  if  a rule "H ==> G | B" exists
  and CT ⊨ D -> ∃x.(H = H', G)
```

E	E	D	
A == B	-	-	(solve)
-	-	A -> B	

E	E	D	
sub(X,X)	-	-	(intro)
-	sub(X,X)	-	(simpl)
-	-	-	

E	E	D	
sub(X,Y), sub(Y,Z)	-	-	(intro)
sub(Z,X)			
-	sub(X,Y), sub(Y,Z),	-	(prop)
	sub(Z,X)		
-	sub(X,Y), sub(Y,Z),	-	(simpl)
	sub(Z,X), sub(X,Z)		
X == Z	sub(X,Y), sub(Y,Z)	-	(solve)
-	sub(Z,Y), sub(Y,Z)	X -> Z	(…)

Implementation of CHR Semantics

What influences performance?

- Matching rules to multiple constraints in the store
- Unification might enable new matches
- Propagation rules can be applied many times

Techniques to deal with this

- Keep indices on the constraints in the store
- Activate constraints if their variables are unified
- Use constraint identities to see if rules are already applied

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 Handling Rules
  - Simplification and propagation rules
  - Depends on built-in procedures to unify or resolve names
  - Itself described by a rewrite semantics
- 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

Next Week

Error reporting from constraints

- How to relate constraint conflicts back to the program
- How to pick good locations to report errors

Exam-style exercises about

- Scope graphs
- Constraint rules
- Constraint resolution