Constraint Programming

Prof. Dr. Thom Frühwirth

SoSe 2019

Based on:
Essentials of Constraint Programming,
Thom Frühwirth and Slim Abdennadher,
Textbook, Springer Verlag, 2003.

The Holy Grail

> Constraint Programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it.

Eugene C. Freuder, Inaugural issue of the *Constraints Journal*, 1997.

Constraint Reasoning

The Idea

- *Example – Combination Lock:*
  0 1 2 3 4 5 6 7 8 9
  Greater or equal 5.
  Prime number.
- *Declarative problem* representation by variables and constraints:
  $x \in \{0, 1, \ldots, 9\} \ \wedge \ x \geq 5 \ \wedge \ \text{prime}(x)$
- *Constraint propagation and simplification* reduce search space:
  $x \in \{0, 1, \ldots, 9\} \ \wedge \ x \geq 5 \ \rightarrow \ x \in \{5, 6, 7, 8, 9\}$

Constraint Programming

Robust, flexible, maintainable software faster.

- *Declarative modeling by constraints:*
  Description of properties and relationships between partially known objects.
  Correct handling of precise and imprecise, finite and infinite, partial and full information.
- *Automatic constraint reasoning:*
  Propagation of the effects of new information (as constraints).
  Simplification makes implicit information explicit.
- *Solving combinatorial problems efficiently:*
  Easy Combination of constraint solving with search and optimization.

Terminology

Language is first-order logic with equality.

- *Constraint:*
  Conjunction of atomic constraints (predicates)
  E.g., $4X + 3Y = 10 \ \wedge \ 2X - Y = 0$
- *Constraint Problem (Query):*
  A given, initial constraint
- *Constraint Solution (Answer):*
  A valuation for the variables in a given constraint problem that satisfies all constraints of the problem. E.g., $X = 1 \wedge Y = 2$

Constraint Programming – Mortgage

```
% D: Amount of Loan, Debt, Principal
% T: Duration of loan in months
% I: Interest rate per month
% R: Rate of payments per month
% S: Balance of debt after T months

mortgage(D, T, I, R, S) <=>
       T = 0,
       D = S
         ;
       T > 0,
       T1 = T - 1,
       D1 = D + D*I - R,
       mortgage(D1, T1, I, R, S).
```

Constraint Programming – Mortgage (cont)

- `mortgage(100000,360,0.01,1025,S)` yields `S=12625.90`.
- `mortgage(D,360,0.01,1025,0)` yields `D=99648.79`.
- `S=<0, mortgage(100000,T,0.01,1025,S)`
  yields `T=374, S=-807.96`.
- `mortgage(D,360,0.01,R,0)` yields `R=0.0102861198*D`.
- If the interest rate `I` is unknown, the equation `D1 = D + D*I - R` will be non-linear after one recursion step, since `D1`, the new `D`, is not determined either.

Aspects of Constraint Logic Programming

Theoretical

Logical Foundation – First-Order Logic

Conceptual

Sound Modeling

Practical

Efficient Algorithms/Implementations

Combination of different Solvers

Constraint Reasoning and Constraint Programming

A generic framework for

- Modeling
  - with partial information
  - with infinite information
- Reasoning
  - with new information
- Solving
  - combinatorial problems

Constraint Solving

Adaption and combination of existing efficient algorithms from

- Mathematics
  - Operations research
  - Graph theory
  - Algebra
- Computer science
  - Finite automata
  - Automatic proving
- Economics
- Linguistics

Early History of Constraint Programming

60s, 70s Constraint networks in artificial intelligence.

70s Logic programming (Prolog).

80s Constraint logic programming.

80s Concurrent logic programming.

90s Concurrent constraint programming.

90s Commercial applications.

Application Domains

- Modeling
- Executable Specifications
- Solving combinatorial problems
  - Scheduling, Planning, Timetabling
  - Configuration, Layout, Placement, Design
  - Analysis: Simulation, Verification, Diagnosis

  of software, hardware and industrial processes.

Application Domains (cont)

- Artificial Intelligence
  - Machine Vision
  - Natural Language Understanding
  - Temporal and Spatial Reasoning
  - Theorem Proving
  - Qualitative Reasoning
  - Robotics
  - Agents
  - Bio-informatics

Applications in Research

- *Computer Science:* Program Analysis, Robotics, Agents
- *Molecular Biology, Biochemistry, Bio-informatics:*
  Protein Folding, Genomic Sequencing
- *Economics:* Scheduling
- *Linguistics:* Parsing
- *Medicine:* Diagnosis Support
- *Physics:* System Modeling
- *Geography:* Geo-Information-Systems

Early Commercial Applications

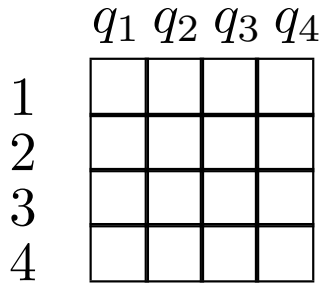
In the late 80s, early 90s:

- *Lufthansa:* Short-term staff planning.
- *Hong Kong Container Harbor:* Resource planning.
- *Renault:* Short-term production planning.
- *Nokia:* Software configuration for mobile phones.
- *Airbus:* Cabin layout.
- *Siemens:* Circuit verification.
- *Caisse d'epargne:* Portfolio management.

In *Decision Support Systems* for Planning, Configuration, for Design, Analysis.

Application – n-Queens Problem

Place n queens $q_1, \ldots, q_n$ on an $n \times n$ chess board, such that they do not attack each other.



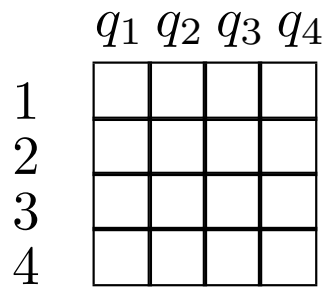
$$q_1, \ldots, q_n \in \{1, \ldots, n\}$$

$$\forall\ i \neq j.\ q_i \neq q_j \wedge |q_i - q_j| \neq |i - j|$$

- no two queens on same row, column or diagonal
  - each row and each column with exactly one queen
  - each diagonal at most one queen
- q_i: row position of the queen in the i-th column

Application – n-Queens Problem (cont)

Place n queens $q_1, \ldots, q_n$ on an $n \times n$ chess board, such that they do not attack each other.



$$q_1, \ldots, q_n \in \{1, \ldots, n\}$$

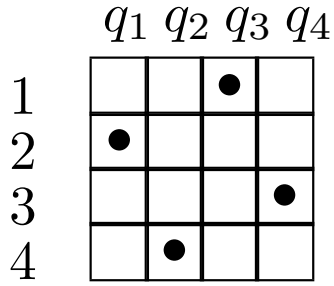
$$\forall\, i \neq j.\ q_i \neq q_j \wedge |q_i - q_j| \neq |i - j|$$

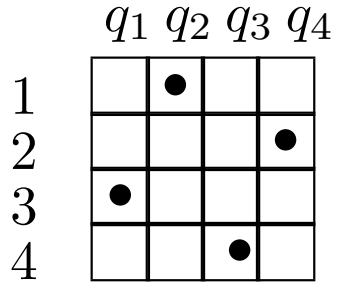
```
solve(N,Qs)       <=> make_domains(N,Qs), queens(Qs), enum(Qs).

queens([Q|Qs])    <=> safe(Q,Qs,1), queens(Qs).

safe(X,[Y|Qs],N) <=> no_attack(X,Y,N), safe(X,Qs,N+1).

no_attack(X,Y,N) <=> X ne Y, X+N ne Y, Y+N ne X.
```

Application – n-Queens Problem (cont 2)

`solve(4,[Q1,Q2,Q3,Q4])`

- `make_domains` produces
  `Q1 in [1,2,3,4], Q2 in [1,2,3,4]`
  `Q3 in [1,2,3,4], Q4 in [1,2,3,4]`
- `safe` adds `noattack` producing `ne` constraints
- `enum` called for labeling
- `[Q1,Q2,Q3,Q4] = [2,4,1,3]`, `[Q1,Q2,Q3,Q4] = [3,1,4,2]`





References

- *Essentials of Constraint Programming*
  Series: Cognitive Technologies
  Thom Frühwirth, Slim Abdennadher
  2003, Springer

- *Constraint-Programmierung*
  Lehrbuch
  Thom Frühwirth, Slim Abdennadher
  1997, Springer

- *Constraint Handling Rules*
  Lehrbuch
  Thom Frühwirth
  2009, Cambridge University Press

Foundations from Logic

> Good, too, Logic, of course; in itself, but not in fine weather.

Arthur Hugh Clough, 1819-1861

> Die Logik muß für sich selber sorgen.

Ludwig Wittgenstein, 1889-1951

First-Order Logic

Syntax – Language

- Alphabet
- Well-formed Expressions

Semantics – Meaning

- Interpretation
- Logical Consequence

Calculi – Derivation

- Inference Rules
- Transition Systems

Syntax of First-Order Logic

Alphabet

- $\mathcal{P}$: predicate symbols: $p, q, r, \ldots$
- $\mathcal{F}$: function symbols: $a, b, c, \ldots, f, g, h, \ldots$
- $\mathcal{V}$: countably infinite set of variables: $X, Y, Z, \ldots$
- logic symbols:
  - truth symbols: $\bot$ (false), $\top$ (true)
  - logical connectives: $\neg$, $\wedge$, $\vee$, $\rightarrow$
  - quantors: $\forall$, $\exists$
  - syntactic symbols: "(","(", ","

Signature $\Sigma = (\mathcal{P}, \mathcal{F})$ of a first-order language

- $\mathcal{P}$: finite set of *predicate symbols*, each with *arity* $n \in \mathbb{N}$
- $\mathcal{F}$: finite set of *function symbols*, each with *arity* $n \in \mathbb{N}$

Naming conventions:

- *nullary, unary, binary, ternary* for arities $0, 1, 2, 3$
- *constants*: nullary function symbols
- *propositions*: nullary predicate symbols

Well-Formed Expressions

Term

Set of *terms* $\mathcal{T}(\Sigma, \mathcal{V})$:

- a *variable* from $\mathcal{V}$, or
- a *function term* $f(t_1, \ldots, t_n)$, where f is an n-ary function symbol from Σ and the *arguments* $t_1, \ldots, t_n$ are terms ($n \geq 0$).

Examples ($a/0$, $f/1$, $g/2$):

- X
- a
- $f(X)$
- $g(f(X), g(Y, f(a)))$

Well-Formed Formula

Set of *(well-formed) formulae* $\mathcal{F}(\Sigma, \mathcal{V}) = \{A, B, C, \ldots, F, G, \ldots\}$:

- *atomic formula* (*atom*) $p(t_1, \ldots, t_n)$, where p is an n-ary predicate symbol from Σ and the *arguments* $t_1, \ldots, t_n$ are terms, or $\bot$, or $\top$, or
- *negation* $\neg F$ of a formula F, or
- *conjunction* $(F \wedge F')$, *disjunction* $(F \vee F')$, or *implication* $(F \to F')$ between formulae F and F', or
- *universally quantified formula* $\forall X F$, or *existentially quantified formula* $\exists X F$ (X variable, F formula).

Example – Terms and Formulae

$\mathcal{P} = \{\texttt{mortal/1}\}$, $\mathcal{F} = \{\texttt{socrates/0}, \texttt{father/1}\}$, $\mathcal{V} = \{X, \dots\}$

- *Terms*:
  X, $\texttt{socrates}$, $\texttt{father}(\texttt{socrates})$, $\texttt{father}(\texttt{father}(\texttt{socrates}))$,
  but not: $\texttt{father}(X, \texttt{socrates})$
- *Atomic Formulae*:
  $\texttt{mortal}(X)$, $\texttt{mortal}(\texttt{socrates})$, $\texttt{mortal}(\texttt{father}(\texttt{socrates}))$
  but not: $\texttt{mortal}(\texttt{mortal}(\texttt{socrates}))$
- *Non-Atomic Formulae*:
  $\texttt{mortal}(\texttt{socrates}) \wedge \texttt{mortal}(\texttt{father}(\texttt{socrates}))$
  $\forall X \texttt{mortal}(X) \rightarrow \texttt{mortal}(\texttt{father}(X))$

Free Variables

- quantified formula $\forall X F$ or $\exists X F$ *binds* variable X within *scope* F
- variables not bound are called *free*

Example – Free Variables

Give the set of free variables for each braced part.

$$\underbrace{p(X) \ \wedge \ \overbrace{\exists X p(X)}}$$

$$\overbrace{\underbrace{\left(\forall X \, \underbrace{p(X,Y)}\right)}_{} \ \vee \ \underbrace{q(X)}}$$

$$\underbrace{}$$

Universal and Existential Closure of F

- *universal closure* $\forall F$ of F:

$$\forall X_1 \forall X_2 \ldots \forall X_n F$$

- *existential closure* $\exists F$ of F:

$$\exists X_1 \exists X_2 \ldots \exists X_n F$$

where $X_1, X_2, \ldots, X_n$ are all free variables of F

Naming Conventions:

- *closed formula* or *sentence*: does not contain free variables
- *theory*: set of sentences
- *ground term or formula*: does not contain any variables

Semantics of First-Order Logic

Interpretation I of Σ

- *universe* U: a non-empty set
- $I(f) : U^n \to U$: function for every n-ary function symbol f of Σ
- $I(p) \subseteq U^n$: relation for every n-ary predicate symbol p of Σ

Variable Valuation for $\mathcal{V}$ w.r.t. I

- $\eta : \mathcal{V} \to U$: for every variable X of $\mathcal{V}$ into the universe U of I

(Σ signature of a first-order language, $\mathcal{V}$ set of variables)

Interpretation of Terms

$\eta^I : \mathcal{T}(\Sigma, V) \to U$: for every term

$$\eta^I(X) := \eta(X) \text{ for a variable } X$$

$$\eta^I(f(t_1, \ldots, t_n)) := I(f)(\eta^I(t_1), \ldots, \eta^I(t_n))$$

for an n-ary function symbol f and terms $t_1, \ldots, t_n$

(Σ signature, I interpretation with universe U, $\eta : \mathcal{V} \to U$ variable valuation)

Interpretation of Formulae – Preliminaries

Given a variable valuation $\eta : \mathcal{V} \to U$. The function $\eta[Y \mapsto u]$ is

$$\eta[Y \mapsto u](X) := \begin{cases} \eta(X) & \text{if } X \neq Y, \\ u & \text{if } X = Y. \end{cases}$$

($\mathcal{V}$ set of variables, X and Y variables in $\mathcal{V}$, U universe, u in U)

Interpretation of Formulae

I interpretation of Σ, η variable valuation *satisfy* F written $I, \eta \models F$:

- $I, \eta \models \top$ and $I, \eta \not\models \bot$
- $I, \eta \models p(t_1, \ldots, t_n)$ iff $(\eta^I(t_1), \ldots, \eta^I(t_n)) \in I(p)$
- $I, \eta \models \neg F$ iff $I, \eta \not\models F$
- $I, \eta \models F \wedge F'$ iff $I, \eta \models F$ and $I, \eta \models F'$
- $I, \eta \models F \vee F'$ iff $I, \eta \models F$ or $I, \eta \models F'$
- $I, \eta \models F \rightarrow F'$ iff $I, \eta \not\models F$ or $I, \eta \models F'$
- $I, \eta \models \forall X F$ iff $I, \eta[X \mapsto u] \models F$ for all $u \in U$
- $I, \eta \models \exists X F$ iff $I, \eta[X \mapsto u] \models F$ for some $u \in U$

Example – Interpretation

$\boxed{\forall X(p(X,a,b) \rightarrow q(b,X))}$

.	$I_1(.)$	$I_2(.)$
U	real things	natural numbers
a	"food"	5
b	"Fritz the cat"	10
p	"_ gives _ to _"	$\_ + \_ > \_$
q	"_ loves _"	$\_ < \_$

I_1: "Fritz the cat loves everybody who gives food to him."

I_2: "10 is less than any X if $X + 5 > 10$."
(counterexample: $X = 6$)

Model of F, Validity

- I *model of* F or I *satisfies* F, written $I \models F$:
  $I, \eta \models F$ for every variable valuation η
- I *model* of theory T: I model of each formula in T

Sentence S is

- *valid*: satisfied by every interpretation, i.e., $I \models S$ for every I
- *satisfiable*: satisfied by some interpretation, i.e., $I \models S$ for some I
- *unsatisfiable*: not satisfied by any interpretation, i.e., $I \not\models S$ for every I
- *falsifiable*: not satisfied by some interpretation, i.e., $I \not\models S$ for some I

(Σ signature, I interpretation)

Example – Validity, Satisfiability, Falsifiable, Unsatisfiability (1)

	valid	satisfiable	falsifiable	unsatisfiable
$A \vee \neg A$				
$A \wedge \neg A$				
$A \rightarrow \neg A$				
$A \rightarrow (B \rightarrow A)$				
$A \rightarrow (A \rightarrow B)$				
$A \leftrightarrow \neg A$				

(A, B formulae)

Logical Consequence

- A sentence/theory T_1 is a *logical consequence* of a sentence/theory T_2, written $T_2 \models T_1$, if every model of T_2 is also a model of T_1, i.e. $I \models T_2$ implies $I \models T_1$.
- Two sentences or theories are *equivalent* ($\models\!\!\models$) if they are logical consequences of each other.
- $\models$ is undecidable [Church]

Example:
$\neg(A \wedge B) \models\!\!\models \neg A \vee \neg B$ (de Morgan)

Example – Tautology Laws (1)

Dual laws hold for $\wedge$ and $\vee$ exchanged.

- $A \models\!\!\!\models \neg\neg A$ (double negation)
- $\neg(A \wedge B) \models\!\!\!\models \neg A \vee \neg B$ (de Morgan)
- $A \wedge A \models\!\!\!\models A$ (idempotence)
- $A \wedge (A \vee B) \models\!\!\!\models A$ (absorption)
- $A \wedge B \models\!\!\!\models B \wedge A$ (commutativity)
- $A \wedge (B \wedge C) \models\!\!\!\models (A \wedge B) \wedge C$ (associativity)
- $A \wedge (B \vee C) \models\!\!\!\models (A \wedge B) \vee (A \wedge C)$ (distributivity)

Example – Tautology Laws (2)

- $A \to B \models\!\dashv \neg A \vee B$ (implication)
- $A \to B \models\!\dashv \neg B \to \neg A$ (contraposition)
- $(A \to (B \to C)) \models\!\dashv (A \wedge B) \to C$
- $\neg\forall X\, A \models\!\dashv \exists X \neg A$
- $\neg\exists X\, A \models\!\dashv \forall X \neg A$
- $\forall X (A \wedge B) \models\!\dashv \forall X\, A \wedge \forall X\, B$
- $\exists X (A \vee B) \models\!\dashv \exists X\, A \vee \exists X\, B$
- $\forall X\, B \models\!\dashv B \models\!\dashv \exists X\, B$ (with X not free in B)

Example – Logical Consequence

F	G	$F \models G$ or $F \not\models G$
A	$A \vee B$	
A	$A \wedge B$	
A, B	$A \vee B$	
A, B	$A \wedge B$	
$A \wedge B$	A	
$A \vee B$	A	
$A, (A \rightarrow B)$	B	

Note: I is a model of $\{A, B\}$, iff $I \models A$ and $I \models B$.

Example – Logical Consequence, Validity, Unsatisfiability

The following statements are equivalent:

1. $F_1, \ldots, F_k \models G$
   (G is a logical consequence of $F_1, \ldots, F_k$)
2. $\left(\bigwedge_{i=1}^{k} F_i\right) \rightarrow G$ is valid.
3. $\left(\bigwedge_{i=1}^{k} F_i\right) \wedge \neg G$ is unsatisfiable.

Note: $F \not\models G$ does *not* imply $F \models \neg G$.

Logic and Calculus

logic: formal language for expressions

- *syntax*: "spelling rules" for expressions
- *semantics*: meaning of expressions (logical consequence $\models$)
- *calculus*: set of given formulae and syntactic rules for manipulation of formulae to perform proofs that can be mechanized

Resolution Calculus

[Robinson 1965]

- calculus that can be easily implemented
- based on Herbrand interpretation
- uses clausal normal form and unification
- used as execution mechanism of (constraint) logic programming

Clausal Normal Form

- *literal*: atom (*positive* literal) or negation of atom (*negative* literal)
- *complementary* literals: positive literal L and its negation $\neg L$
- *clause (in disjunctive normal form)*: formula of the form $\bigvee_{i=1}^{n} L_i$ where L_i are literals.
  - *empty clause (empty disjunction)*: $n = 0$: $\bot$

Clauses and Literals (cont)

- *implication form* of the clause:

$$F = \underbrace{\bigwedge_{j=1}^{n} B_j}_{body} \rightarrow \underbrace{\bigvee_{k=1}^{m} H_k}_{head}$$

for

$$F = \bigvee_{i=1}^{n+m} L_i \text{ with } L_i = \begin{cases} \neg B_i & \text{for } i = 1, \ldots, n \\ H_{i-n} & \text{for } i = n+1, \ldots, n+m \end{cases}$$

for atoms B_j and H_k

- *closed clause*: sentence $\forall x_1, \ldots, x_n C$ with C clause
- *clausal form* of theory: consists of closed clauses

Normalization steps

Any theory can be transformed into clausal form as follows

- Convert every formula into an equivalent formula in *negation normal form* using tautology laws that move negation inwards.
- Perform *Skolemization* in order to eliminate all existential quantifiers by replacing existentially quantified variables with function terms.
- Move conjunctions and universal quantifiers outwards by applying the distributive tautology laws.

Herbrand Theorem

Recall: The formula A is valid if $I, \eta \models A$ for every interpretation I and every valuation η.

Jacques Herbrand (1908-1931) discovered that there is a *universal* domain together with a *universal* interpretation, s.t. that any *universally* valid formula (in clausal normal form) is valid in *any* interpretation. Therefore, only interpretations in the *Herbrand universe* need to be checked (provided the Herbrand universe is infinite).

Herbrand Interpretation

- *Herbrand universe*: set $\mathcal{T}(\Sigma, \emptyset)$ of ground terms
- for every n-ary function symbol f of Σ, the assigned function $I(f)$ maps a tuple $(t_1, \ldots, t_n)$ of ground terms to the ground term "$f(t_1, \ldots, t_n)$".

Herbrand model of sentence/theory: Herbrand interpretation satisfying sentence/theory.

Herbrand base for signature Σ: set of ground atoms in $\mathcal{F}(\Sigma, \emptyset)$, i.e., $\{p(t_1, \ldots, t_n) \mid p \text{ is an } n\text{-ary predicate symbol of } \Sigma \text{ and } t_1, \ldots, t_n \in \mathcal{T}(\Sigma, \emptyset)\}$.

Example – Herbrand Interpretation

For the formula $\forall X \forall Y.p(X, a) \wedge q(X, f(Y))$ the (infinite, as there is a constant and a function symbol) Herbrand domain is $\{a, f(a), f(f(a)), f(f(f(a))), \dots\}$.

For the formula $F \equiv \exists X \exists Y.p(X, a) \wedge \neg p(Y, a)$ the Herbrand universe is $\{a\}$ and F is unsatisfiable in the Herbrand universe as $p(a, a) \wedge \neg p(a, a)$ is false, i.e., there is no Herbrand model.

However, if we add another element b we have $p(a, a) \wedge \neg p(b, a)$. So F is satisfiable for any interpretation whose universe has cardinality greater than 1.

Unification

Substitution to a Term

- function $\sigma : \mathcal{V} \to \mathcal{T}(\Sigma, \mathcal{V}')$
- *finite*: $\mathcal{V}$ finite, written as $\{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$ with distinct variables X_i and terms t_i
- *identity substitution* $\epsilon = \emptyset$
- written as postfix operator, application from left to right in composition
- $\sigma : \mathcal{T}(\Sigma, \mathcal{V}) \to \mathcal{T}(\Sigma, \mathcal{V}')$
  implicit homomorphic extension, i.e.,
  $f(t_1, \ldots, t_n)\sigma := f(t_1\sigma, \ldots, t_n\sigma)$.

Example:
$\sigma = \{X \mapsto 2, Y \mapsto 5\}$: $(X * (Y + 1))\sigma = 2 * (5 + 1)$

Examples – Substitution to a Formula

- $\sigma = \{X \mapsto Y, Z \mapsto 5\}$: $(X * (Z + 1))\sigma = Y * (5 + 1)$
- $\sigma = \{X \mapsto Y, Y \mapsto Z\}$ (no idempotence)
  $p(X)\sigma = p(Y) \neq p(X)\sigma\sigma = p(Z)$
- $\sigma = \{X \mapsto Y\}, \tau = \{Y \mapsto 2\}$ (no commutativity)
  $(X * (Y + 1))\sigma\tau = (Y * (Y + 1))\tau = (2 * (2 + 1))$
  $(X * (Y + 1))\tau\sigma = (X * (2 + 1))\sigma = (Y * (2 + 1))$

Dealing with Quantifiers:

- $\sigma = \{X \mapsto Y\}$: $(\forall X p(3))\sigma = \forall X' p(3)$
- $\sigma = \{X \mapsto Y\}$: $(\forall X p(X))\sigma = \forall X' p(X')$,
  $(\forall X p(Y))\sigma = \forall X' p(Y)$
- $\sigma = \{Y \mapsto X\}$: $(\forall X p(X))\sigma = \forall X' p(X')$,
  $(\forall X p(Y))\sigma = \forall X' p(X)$

Instance, Variable Renaming, Variants

Logical expressions are terms, formulae, substitutions or tuples thereof.

- e *instance of* e': $e = e'\sigma$
- e' *more general than* e: e is instance of e'
- e and e' *variants* (identical modulo variable renaming):
  $e = e'\sigma$ and $e' = e\tau$ (instances of each other)
  σ and τ are called *variable renamings*

(e, e' logical expressions, σ, τ substitutions)

Unification

- *unifier* for e_1, e_2: $e_1\sigma = e_2\sigma$
- e_1, e_2 *unifiable*: unifier exists
- *most general unifier (mgu)* σ for e_1, e_2: every unifier τ for e_1, e_2 is instance of σ, i.e., $\tau = \sigma\rho$ for some ρ

(e_1, e_2 expressions, σ, τ, ρ substitutions)

Most General Unifier by Hand

Given two logical expressions to unify. Return mgu or failure.

- Start with empty subsitution ϵ. Scan expressions simultaneously from left to right according to their structure.
- Check the syntactic equivalence of the symbols encountered: Repeat until no more expression to process
  - *identical function/predicate symbols with the same arity*: continue with arguments
  - *different symbols*: halt with failure
  - *identical variables*: continue
  - *one expression is a variable*:
    - variable occurs in other expression: halt with failure
    - add corresponding substitution
    - apply the new substitution to the logical expressions

Example – Most General Unifier (mgu)

to unify	current substitution, remarks
$p(X, f(a))$ *vs.* $p(a, f(X))$	ϵ, start
$p/2$ *vs.* $p/2$	identical symbols, continue w. arguments
X *vs.* a	$\{X \mapsto a\}$, substitution added
$f(X)\{X \mapsto a\} = f(a)$	apply new subsitution
$f(a)$ *vs.* $f(a)$	identical symbols, continue w. arguments
a *vs.* a	identical symbols, no more expressions

So mgu is $\{X \mapsto a\}$.

What about $p(X, f(a), Y) = p(a, f(X))$?
What about $p(X, f(b)) = p(a, f(X))$?

Examples – Most General Unifier

s	t	
f	g	failure
a	a	ϵ
X	a	$\{X \mapsto a\}$
X	Y	$\{X \mapsto Y\}$, but also $\{Y \mapsto X\}$
$f(a, X)$	$f(Y, b)$	$\{Y \mapsto a, X \mapsto b\}$
$f(g(a, X), Y)$	$f(c, X)$	failure
$f(g(a, X), h(c))$	$f(g(a, b), Y)$	$\{X \mapsto b, Y \mapsto h(c)\}$
$f(g(a, X), h(Y))$	$f(g(a, b), Y)$	failure

Inference Rules

- Given a set of formulae

$$\frac{F_1, \ldots, F_n}{F}$$

  - *premises* $F_1, \ldots, F_n$ with *conclusion* F
  - if premises are given, conclusion is added to the formulae
  - this is called a *derivation step*
  - *derivation*: sequence of derivation steps with conclusion taken as premises for next step

- rule application usually nondeterministic

Resolution Calculus – Inference Rules

Works by contradiction: Theory united with negated consequence must be unsatisfiable ("derive empty clause").

Axiom

$\bot$ empty clause (i.e. the elementary contradiction)

Resolution Step

$$\frac{R \vee A \qquad R' \vee \neg A'}{(R \vee R')\sigma}$$

σ is a most general unifier for the atoms A and A'

Factoring

$$\frac{R \vee L \vee L'}{(R \vee L)\sigma}$$

σ is a most general unifier for the literals L and L'

$R \vee A$ and $R' \vee \neg A'$ have different variables (otherwise rename them apart)

Resolution – Remarks

- *resolution rule*:
  - two clauses C and C' instantiated s.t. literal from C and literal from C' complementary
  - two instantiated clauses are combined into a new clause
  - *resolvent* added
- *factoring rule*:
  - clause C instantiated, s.t. two literals become equal
  - remove one literal
  - *factor* added

Example – Resolution Calculus

Resolution:

$$\frac{p(a,X) \vee q(X) \qquad \neg p(a,b) \vee r(X)}{(q(X) \vee r(X))\{X \mapsto b\}}$$

Factoring:

$$\frac{p(X) \vee p(b)}{p(X)\{X \mapsto b\}}$$

Refined Calculus

To define *Operational (Procedural) Semantics* of programming languages.

Triple $(\Sigma, \equiv, T)$

- Σ: signature for a first-order logic language
- $\equiv$: *congruence* (equivalence relation) on states
- $T = (S, \mapsto)$: simple state transition system
  states S represent logical expressions over the signature Σ

Simple State Transition System

- $(S, \mapsto)$
  - S set of states
  - $\mapsto$ binary relation over states: *transition relation*
  - *(state) transition* from S_1 to S_2 possible if (some give) condition holds, written $S_1 \mapsto S_2$
- distinguished subsets of S: *initial* and *final states*
- $S_1 \mapsto S_2 \mapsto \ldots \mapsto S_n$ *derivation (computation)*
- $S \mapsto^* S'$ reflexive-transitive closure of $\mapsto$

Reduction is synonym for transition

Transition Rules

IF	*Condition*
THEN	$S \mapsto S'$.

- *(state) transition (reduction, derivation step, computation)* from S to S' possible if the condition *Condition* holds
- straightforward to concretize a calculus ("$\vdash$") into a state transition system ("$\mapsto$")

Congruence

- states which are considered equivalent for purpose of computation
- *congruence* instead of modeling with additional transition rules
- formally congruence is equivalence relation:

  (Reflexivity) $A \equiv A$

  (Symmetry) If $A \equiv B$ then $B \equiv A$

  (Transitivity) If $A \equiv B$ and $B \equiv C$ then $A \equiv C$

Example:
$(X = 3) \equiv (3 = X)$

Congruence

Commutativity:	$G_1 \wedge G_2$	$\equiv$	$G_2 \wedge G_1$
Associativity:	$G_1 \wedge (G_2 \wedge G_3)$	$\equiv$	$(G_1 \wedge G_2) \wedge G_3$
Identity:	$G \wedge \top$	$\equiv$	G
Absorption:	$G \wedge \bot$	$\equiv$	$\bot$

derived from tautology laws of predicate logic

Constraint Programming Languages

Preliminaries – Syntax and Semantics

Syntax

- *extended Backus-Naur form* (EBNF) grammar

$$\mathit{Name}:\ \ G, H ::= A \mid B,\ \ \mathit{Condition}$$

  - *capital letters*: syntactical entities
  - *symbols G and H defined*: with name *Name*.
  - *Condition* holds: G and H can be of the form A or B

Semantics

Operational (Procedural) Semantics

State transition system (refined calculus)

Declarative Semantics

Logical reading (meaning) of program as theory (set of formulae of first-order logic)

Operational vs. Declarative Semantics

- Soundness
- Completeness

Logic Programming

> A logic program is a set of axioms, or rules, defining relationships between objects. A computation of a logic program is a deduction of consequences of the program. A program defines a set of consequences, which is its meaning. The art of logic programming is constructing concise and elegant programs that have desired meaning.

Sterling and Shapiro: The Art of Prolog, Page 1.

LP Syntax

Only clauses with at most one positive literal (*Horn clauses*).

- *goal*: G
  - *empty goal* $\top$ (top) or $\bot$ (bottom), or
  - atom, or
  - conjunction of goals
- *(definite) clause*: $A \leftarrow G$
  - *head* A: atom
  - *body* G: goal
- Naming conventions
  - *query*: clause of form $\bot \leftarrow G$, abbreviated to G
  - *fact*: clause of form $A \leftarrow \top$, abbreviated to A
  - *rule*: otherwise
- *(logic) program*: finite set of definite clauses

LP Calculus – Syntax Summary EBNF Grammar

Atom:	A, B	$::=$	$p(t_1, \ldots, t_n),\ t_i$ terms$, n \geq 0$
Goal:	G, H	$::=$	$\top \mid \bot \mid A \mid G \wedge H$
Clause:	K	$::=$	$A \leftarrow G$
Program:	P	$::=$	$K_1 \ldots K_m,\ m \geq 0$

LP Calculus – State Transition System

- *state* $<G, \theta>$
  - G: goal
  - θ: substitution
- *initial state* $<G, \epsilon>$
- *successful final state* $<\top, \theta>$
- *failed final state* $<\bot, \epsilon>$

Derivations, Goals

Derivation

- *successful*: its final state is successful
- *failed*: its final state is failed
- *infinite*: if there are an infinite sequence of states and transitions $S_1 \mapsto S_2 \mapsto S_3 \mapsto \ldots$

Goal G

- *successful*: has a successful derivation starting with $<G, \epsilon>$
- *finitely failed*: has only failed derivations starting with $<G, \epsilon>$

Logical Reading, Answer

- *logical reading* of $<H, \theta>$: $\exists \bar{X}(H\theta)$
  - *where* $<G, \epsilon> \mapsto^* <H, \theta>$
  - $\bar{X}$: variables which occur in $H\theta$ but not in G
- *(computed) answer of a goal* G:
  substitution θ with successful derivation $<G, \epsilon> \mapsto^* <\top, \theta>$

G is also called *initial goal* or *query*

Operational Semantics

LP Transition Rules

Unfold

If	$(B \leftarrow H)$ is a fresh variant of a clause in P
and	β is the most general unifier of B and $A\theta$
then	$<A \wedge G, \theta> \mapsto <H \wedge G, \theta\beta>$

Failure

If	there is no clause $(B \leftarrow H)$ in P
with	a unifier of B and $A\theta$
then	$<A \wedge G, \theta> \mapsto <\bot, \epsilon>$

Non-determinism

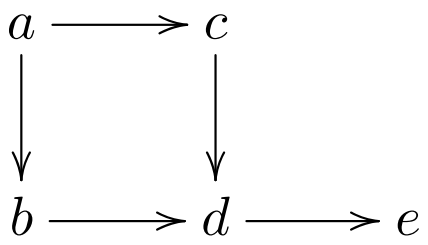
The **Unfold** transition exhibits two kinds of non-determinism.

- *don't-care non-determinism*:
  - any atom in $A \wedge G$ can be chosen as the atom A according to the congruence defined on states
  - affects length of derivation (infinite in the worst case)
- *don't-know non-determinism*:
  - any clause $(B \leftarrow H)$ in P for which B and $A\theta$ are unifiable can be chosen
  - determines the computed answer of derivation

SLD Resolution
(Selective Linear Definite Clause Resolution)

- *selection strategy*: use textual order of clauses and atoms in a program
- *(chronological) backtracking (backtrack search)*
- left-to-right, depth-first exploration of the search tree
- efficient implementation using a stack-based approach
- can get trapped in infinite derivations
  (but breadth-first search too inefficient)

Example - Accessibility in Directed Acyclic Graph

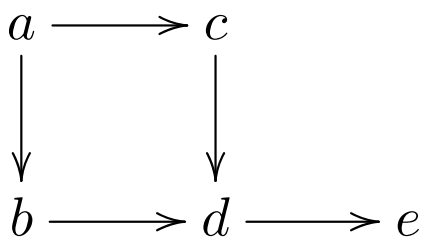


$$\texttt{edge(a,b)} \leftarrow \top \qquad (e1)$$
$$\texttt{edge(a,c)} \leftarrow \top \qquad (e2)$$
$$\texttt{edge(b,d)} \leftarrow \top \qquad (e3)$$
$$\texttt{edge(c,d)} \leftarrow \top \qquad (e4)$$
$$\texttt{edge(d,e)} \leftarrow \top \qquad (e5)$$

$$\texttt{path(Start,End)} \leftarrow \texttt{edge(Start,End)} \qquad (p1)$$
$$\texttt{path(Start,End)} \leftarrow \texttt{edge(Start,Node)} \wedge \texttt{path(Node,End)} \qquad (p2)$$

Example - Accessibility in DAG (cont)

$$\texttt{path(Start,End)} \leftarrow \texttt{edge(Start,End)} \quad (p1)$$
$$\texttt{path(Start,End)} \leftarrow \texttt{edge(Start,Node)} \wedge \texttt{path(Node,End)} \quad (p2)$$



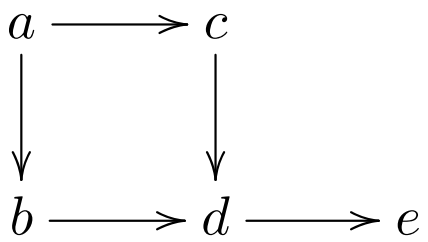
With first rule *p1* for `path` selected:

$$\begin{array}{ll} & <\texttt{path}(\texttt{b},\texttt{Y}),\varepsilon> \\ \mapsto_{\textbf{Unfold}\ (p1)} & <\texttt{edge}(\texttt{S},\texttt{E}),\{\texttt{S}\leftarrowtail \texttt{b},\texttt{E}\leftarrowtail \texttt{Y}\}> \\ \mapsto_{\textbf{Unfold}\ (e3)} & <\top,\{\texttt{S}\leftarrowtail \texttt{b},\texttt{E}\leftarrowtail \texttt{d},\texttt{Y}\leftarrowtail \texttt{d}\}> \end{array}$$

Answer is $\{\texttt{S}\leftarrowtail \texttt{b},\texttt{E}\leftarrowtail \texttt{d},\texttt{Y}\leftarrowtail \texttt{d}\}$.

Example - Accessibility in DAG (cont)

$$\texttt{path(Start,End)} \leftarrow \texttt{edge(Start,End)} \quad (p1)$$
$$\texttt{path(Start,End)} \leftarrow \texttt{edge(Start,Node)} \wedge \texttt{path(Node,End)} \quad (p2)$$



With second rule *p2* for `path` selected:

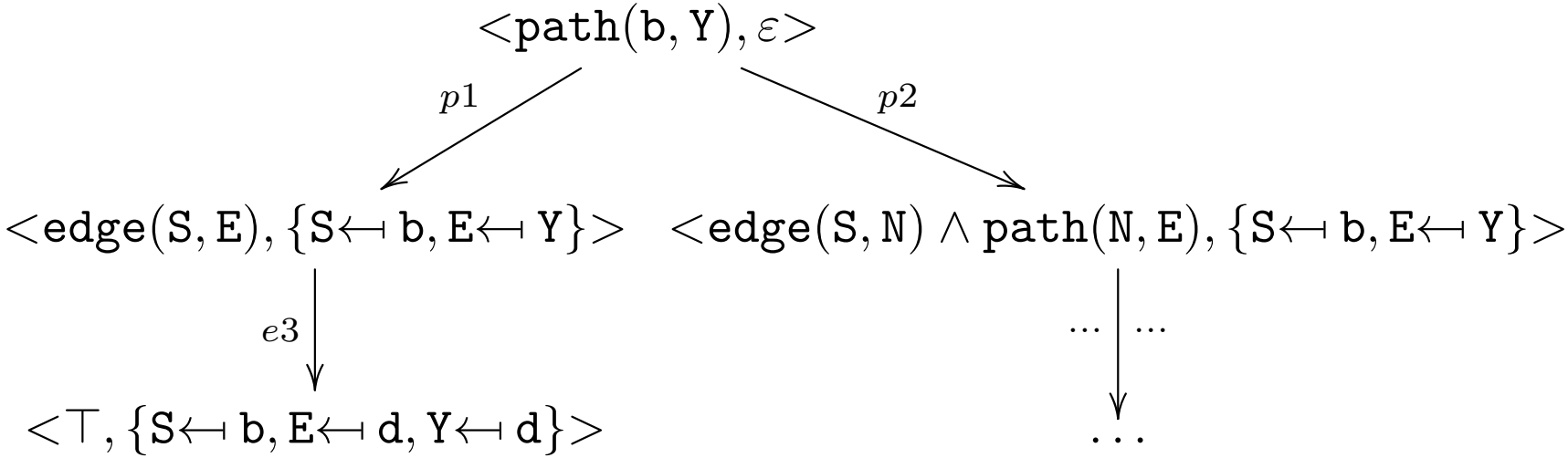
$$\begin{aligned}
& <\texttt{path(b, Y)}, \varepsilon> \\
\mapsto_{\textbf{Unfold}\ (p2)} & <\texttt{edge(S, N)} \wedge \texttt{path(N, E)}, \{\texttt{S} \leftarrowtail \texttt{b}, \texttt{E} \leftarrowtail \texttt{Y}\}> \\
\mapsto_{\textbf{Unfold}\ (e3)} & <\texttt{path(N, E)}, \{\texttt{S} \leftarrowtail \texttt{b}, \texttt{E} \leftarrowtail \texttt{Y}, \texttt{N} \leftarrowtail \texttt{d}\}> \\
\mapsto_{\textbf{Unfold}\ (p1)} & <\texttt{edge(N, E)}, \{\texttt{S} \leftarrowtail \texttt{b}, \texttt{E} \leftarrowtail \texttt{Y}, \texttt{N} \leftarrowtail \texttt{d}\}> \\
\mapsto_{\textbf{Unfold}\ (e5)} & <\top, \{\texttt{S} \leftarrowtail \texttt{b}, \texttt{E} \leftarrowtail \texttt{e}, \texttt{N} \leftarrowtail \texttt{d}, \texttt{Y} \leftarrowtail \texttt{e}\}>
\end{aligned}$$

Example - Accessibility in DAG (cont)

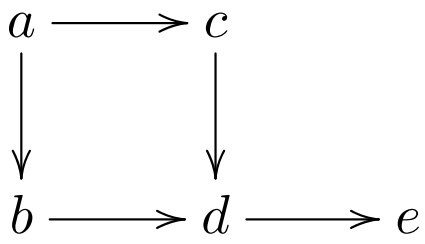
$\texttt{path(Start,End)} \leftarrow \texttt{edge(Start,End)} \quad (p1)$

$\texttt{path(Start,End)} \leftarrow \texttt{edge(Start,Node)} \wedge \texttt{path(Node,End)} \quad (p2)$

Partial search tree:



Example - Accessibility in DAG (cont)



With the first rule:

$$\begin{array}{ll} & <\texttt{path}(\texttt{f},\texttt{g}),\varepsilon> \\ \mapsto_{\textbf{Unfold}\ (p1)} & <\texttt{edge}(\texttt{S},\texttt{E}),\{\texttt{S}\leftarrowtail\texttt{f},\texttt{E}\leftarrowtail\texttt{g}\}> \\ \mapsto_{\textbf{Failure}} & <\bot,\varepsilon> \end{array}$$

With the second rule:

$$\begin{array}{ll} & <\texttt{path}(\texttt{f},\texttt{g}),\varepsilon> \\ \mapsto_{\textbf{Unfold}\ (p2)} & <\texttt{edge}(\texttt{S},\texttt{N})\wedge\texttt{path}(\texttt{N},\texttt{E}),\{\texttt{S}\leftarrowtail\texttt{f},\texttt{E}\leftarrowtail\texttt{g}\}> \\ \mapsto_{\textbf{Failure}} & <\bot,\varepsilon> \end{array}$$

Declarative Semantics

- *implication* ($G \rightarrow A$): Horn clause
- *logical reading of a program* P: universal closure of the conjunction of the clauses of P, written $P^{\rightarrow}$
- only positive information can be derived
- "complete" $P^{\rightarrow}$:
  - keep *necessary* conditions (implications)
  - add corresponding *sufficient* conditions (implications in the other direction)

Example:
In DAG with nodes a, b, c, d, e:
$P^{\rightarrow} \not\models$ `path(f,g)`, $P^{\rightarrow} \not\models \neg$`path(f,g)`
In completed logical reading, $\neg$`path(f,g)` is a logical consequence.

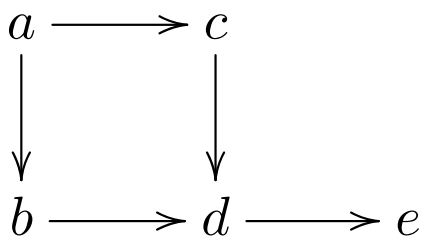
Completion of P: $P^{\leftrightarrow}$

For each p/n in P			add to $P^{\leftrightarrow}$ the formula			
$p(\bar{t}_1)$	$\leftarrow$	G_1	$\forall \bar{X}\ (p(\bar{X})$	$\leftrightarrow$	$\exists \bar{Y}_1\ (\bar{t}_1 \doteq \bar{X} \wedge G_1)$	$\vee$
$\vdots$		$\vdots$			$\vdots$	$\vee$
$p(\bar{t}_m)$	$\leftarrow$	$G_m,$			$\exists \bar{Y}_m\ (\bar{t}_m \doteq \bar{X} \wedge G_m)),$	

- $\bar{X}$: pairwise distinct fresh variables
- $\bar{t}_i$: terms
- $\bar{Y}_i$: variables occurring in G_i and t_i

Clark's completion of P: $P^{\leftrightarrow} \cup CET$

Example – Logical Reading



For the `edge/2` predicates

$$\texttt{edge(a,b)} \leftarrow \top$$
$$\vdots$$
$$\texttt{edge(d,e)} \leftarrow \top$$

add to $P^{\leftrightarrow}$ the formula

$$\forall X_1 X_2 \quad (\texttt{edge}(X_1, X_2) \leftrightarrow$$
$$(\texttt{a} \doteq X_1, \texttt{b} \doteq X_2) \quad \vee$$
$$\vdots \quad \vee$$
$$(\texttt{d} \doteq X_1, \texttt{e} \doteq X_2))$$

Example – Logical Reading (2)

For the `path/2` predicates

```
path(Start,End) ←
edge(Start,End)

path(Start,End) ←
edge(Start,Node) ∧
path(Node,End)
```

add to $P^{\leftrightarrow}$ the formula

$$\forall X_1 X_2 (\texttt{path}(X_1, X_2) \leftrightarrow$$
$$\exists Y_{11} Y_{12}\ (Y_{11} \doteq X_1, Y_{12} \doteq X_2$$
$$\wedge\ \texttt{edge}(Y_{11}, Y_{12}))$$
$$\vee$$
$$\exists Y_{21} Y_{22} Y_{23}\ (Y_{21} \doteq X_1, Y_{22} \doteq X_2$$
$$\wedge\ \texttt{edge}(Y_{21}, Y_{23})$$
$$\wedge\ \texttt{path}(Y_{23}, Y_{22}))$$

Clark's Equality Theory (CET)

Universal closure of the formulae (axiom scheme)

Reflexivity	$(\top \rightarrow X \doteq X)$
Symmetry	$(X \doteq Y \rightarrow Y \doteq X)$
Transitivity	$(X \doteq Y \wedge Y \doteq Z \rightarrow X \doteq Z)$
Compatibility	$(X_1 \doteq Y_1 \wedge \ldots \wedge X_n \doteq Y_n \rightarrow f(X_1, \ldots, X_n) \doteq f(Y_1, \ldots, Y_n))$
Decomposition	$(f(X_1, \ldots, X_n) \doteq f(Y_1, \ldots, Y_n) \rightarrow X_1 \doteq Y_1 \wedge \ldots \wedge X_n \doteq Y_n)$
Contradiction (Clash)	$(f(X_1, \ldots, X_n) \doteq g(Y_1, \ldots, Y_m) \rightarrow \bot)$ if $f \neq g$ or $n \neq m$
Acyclicity	$(X \doteq t \rightarrow \bot)$ if t is function term and X appears in t

(Σ: signature with infinitely many function symbols, including at least one constant)

Acyclicity Examples

$X \doteq X$ is unifiable but *not*:

- $X \doteq f(X)$
- $X \doteq p(A, f(X, a))$
- $X \doteq Y \wedge X \doteq f(Y)$

Unifiable in CET

Terms s and t are unifiable if and only if

$$CET \models \exists (t \doteq s).$$

Soundness and Completeness

Successful derivations

- **Soundness:**
  If θ is a computed answer of G, then $P^{\leftrightarrow} \cup CET \models \forall G\theta$.
- **Completeness:**
  If $P^{\leftrightarrow} \cup CET \models \forall G\theta$, then a computed answer σ of G exists, such that $\theta = \sigma\beta$.

(There may also be failed or infinite derivations.)

(P logic program, $P^{\leftrightarrow} \cup CET$ Clark's completion, G goal, θ substitution)

Failed Derivations

- **Fair Derivation:**
  Either fails or each atom appearing in the derivation is selected after finitely many reductions.
- **Soundness and Completeness:**
  Any fair derivation starting with $<G, \epsilon>$ fails finitely if and only if

$$P^{\leftrightarrow} \cup CET \models \neg\exists G.$$

Remark: SLD resolution *not* fair. Derivation may be infinite.

(P logic program, G goal)

Constraint Logic Programming (CLP)

- *Constraint Satisfaction Problems (CSP)*
  - artificial intelligence (1970s)
  - e.g. $X{\in}\{1,2\} \wedge Y{\in}\{1,2\} \wedge Z{\in}\{1,2\} \wedge X{=}Y \wedge X{\neq}Z \wedge Y{>}Z$
- *Constraint Logic Programming (CLP)*
  - developed in the mid-1980s
  - combination of two declarative paradigms: constraint solving and logic programming
  - together more expressive, flexible, efficient
  - *LP languages*: arbitrary predicates, non-deterministic (search)
  - *constraint solvers*: special predicates, deterministic (efficient)

Early history of constraint-based and logic programming

1963	I. Sutherland, Sketchpad, graphic system for geometric drawing
1970	U. Montanari, Pisa, Constraint networks
1970	R.E. Fikes, REF-ARF, language for integer linear equations
1972	A. Colmerauer, U. Marseille, and R. Kowalski, IC London, **Prolog**
1977	A.K. Mackworth, Constraint networks algorithms
1978	J.-L. Lauriere, Alice, language for combinatorial problems
1979	A. Borning, Thinglab, interactive graphics
1980	G.L. Steele, Constraints, first constraint-based language, in LISP
1982	A. Colmerauer, Prolog II, U. Marseille, equality constraints
1984	Eclipse Prolog, ECRC Munich, later IC-PARC London
1985	**SICStus Prolog**, Swedish Institute of Computer Science (SICS)
1987	**SWI Prolog**, J. Wielemaker, U. Amsterdam

Early history of constraint-based programming (1987/1988)

1987	H. Ait-Kaci, U. Austin, Life, equality constraints
1987	J. Jaffar and J.L. Lassez, **CLP(X) - Scheme**, Monash U. Melbourne
1987	J. Jaffar, **CLP($\Re$)**, Monash U. Melbourne, linear polynomials
1988	P. v. Hentenryck, **CHIP**, ECRC Munich, finite domains, Booleans
1988	P. Voda, Trilogy, Vancouver, integer arithmetics
1988	W. Older, BNR-Prolog, Bell-Northern Research Ottawa, intervals
1988	A. Aiba, CAL, ICOT Tokyo, non-linear equation systems
1988	W. Leler, Bertrand, term rewriting for defining constraints
1988	A. Colmerauer, Prolog III, U. Marseille, list constraints and more

Constraints

Constraints are special predicates of general interest (e.g. arithmetic).

- signature augmented with *constraint symbols*
- consistent first-order constraint theory (CT) describes constraints
- constraints include *true* and *false* as well as syntactic equality $\doteq$ (by including CET into CT)
- constraints handled by predefined, given built-in constraint solver

CLP Syntax

- *atom*: $p(t_1, \ldots, t_n)$, with n-ary predicate symbol p/n
- *atomic constraint*: $c(t_1, \ldots, t_n)$, with n-ary constraint symbol c/n
- *constraint*:
  - atomic constraint, or conjunction of constraints
- *goal*:
  - $\top$ (top), or $\bot$ (bottom), or
  - atom, or an atomic constraint, or
  - conjunction of goals
- *(CL) clause*: $A \leftarrow G$, with atom A (*head*) and goal G (*body*)
- *CL program*: finite set of CL clauses

CLP Syntax Summary – EBNF Grammar

Atom:	A, B	$::=$	$p(t_1, \ldots, t_n),\ n \geq 0$
Constraint:	C, D	$::=$	$c(t_1, \ldots, t_n) \mid C \wedge D,\ n \geq 0$
Goal:	G, H	$::=$	$\top \mid \bot \mid A \mid C \mid G \wedge H$
CL Clause:	K	$::=$	$A \leftarrow G$
CL Program:	P	$::=$	$K_1 \ldots K_m,\ m \geq 0$

CLP Example – Min

The minimum of X and Y is Z.

$$\texttt{min(X,Y,Z)} \leftarrow \texttt{X}\leq\texttt{Y} \wedge \texttt{X}\doteq\texttt{Z} \quad (c1)$$
$$\texttt{min(X,Y,Z)} \leftarrow \texttt{Y}\leq\texttt{X} \wedge \texttt{Y}\doteq\texttt{Z} \quad (c2)$$

Constraints with usual meaning

- $\leq$ total order
- $\doteq$ syntactic equality

CLP State Transition System

- *state* $<G, C>$: G goal (store), C constraint (store)
- *initial state*: $<G, true>$
- *successful final state*: $<\top, C>$ and C is different from *false*
- *failed final state*: $<G, false>$
- *successful and failed derivations and goals*: as in LP calculus

CLP Derivations, Goals

Derivation is

- *successful*: its final state is successful
- *failed*: its final state is failed
- *infinite*: if there are an infinite sequence of states and transitions $S_1 \mapsto S_2 \mapsto S_3 \mapsto \ldots$

Goal G is

- *successful*: it has a successful derivation starting with $<G, true>$
- *finitely failed*: it has only failed derivations starting with $<G, true>$

CLP Operational Semantics

Unfold

If	$(B \leftarrow H)$ is a fresh variant of a clause in P
and	$CT \models \exists\,((B \doteq A) \wedge C)$
then	$<A \wedge G, C> \mapsto <H \wedge G, (B \doteq A) \wedge C>$

Failure

If	there is no clause $(B \leftarrow H)$ in P
with	$CT \models \exists\,((B \doteq A) \wedge C)$
then	$<A \wedge G, C> \mapsto <\bot, \mathit{false}>$

Solve

If	$CT \models \forall\,((C \wedge D_1) \leftrightarrow D_2)$
then	$<C \wedge G, D_1> \mapsto <G, D_2>$

CLP Unfold – Comparison with LP

Unfold	
If	$(B \leftarrow H)$ is a fresh variant of a clause in P
and	$CT \models \exists ((B \doteq A) \wedge C)$
then	$<A \wedge G, C> \mapsto <H \wedge G, (B \doteq A) \wedge C>$

Generalization of logic programming (LP)

- most general unifier (mgu substitution) in LP replaced in CLP by equality constraint between B and A in context of constraint store C, add equality constraint to store C

($(B \doteq A)$: shorthand for equating arguments of B and A pairwise)

CLP Solve

Solve

If $CT \models \forall\ ((C \wedge D_1) \leftrightarrow D_2)$

then $< C \wedge G, D_1 > \mapsto < G, D_2 >$

- form of simplification depends on constraint system and its constraint solver
- try to simplify inconsistent constraints to *false*
- failed final state can be reached via **Solve**

CLP State Transition System (vs. LP)

- *LP*: accumulate and compose substitutions
  CLP: accumulate and simplify constraints
- like substitutions, constraints never removed from constraint store (information increases monotonically during derivations)
- like in LP, two degrees of non-determinism in the calculus (selecting the goal and selecting the clause)
- like in LP, search trees (mostly SLD resolution)

CLP as Extension to LP

- derivation in LP can be expressed as CLP derivations:
  - *LP*: substitution $\{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$
  - *CLP*: equality constraints: $X_1 \doteq t_1 \wedge \ldots \wedge X_n \doteq t_n$
- CLP generalizes form of answers.
  - *LP answer*: substitution
  - *CLP answer*: constraint

CLP Example – Min

The minimum of X and Y is Z.

$$\texttt{min(X,Y,Z)} \leftarrow \texttt{X}\leq\texttt{Y} \wedge \texttt{X}\doteq\texttt{Z} \quad (c1)$$
$$\texttt{min(X,Y,Z)} \leftarrow \texttt{Y}\leq\texttt{X} \wedge \texttt{Y}\doteq\texttt{Z} \quad (c2)$$

Constraints with usual meaning

- $\leq$ total order
- $\doteq$ syntactic equality

CLP Example – Goal `min(1,2,C)`

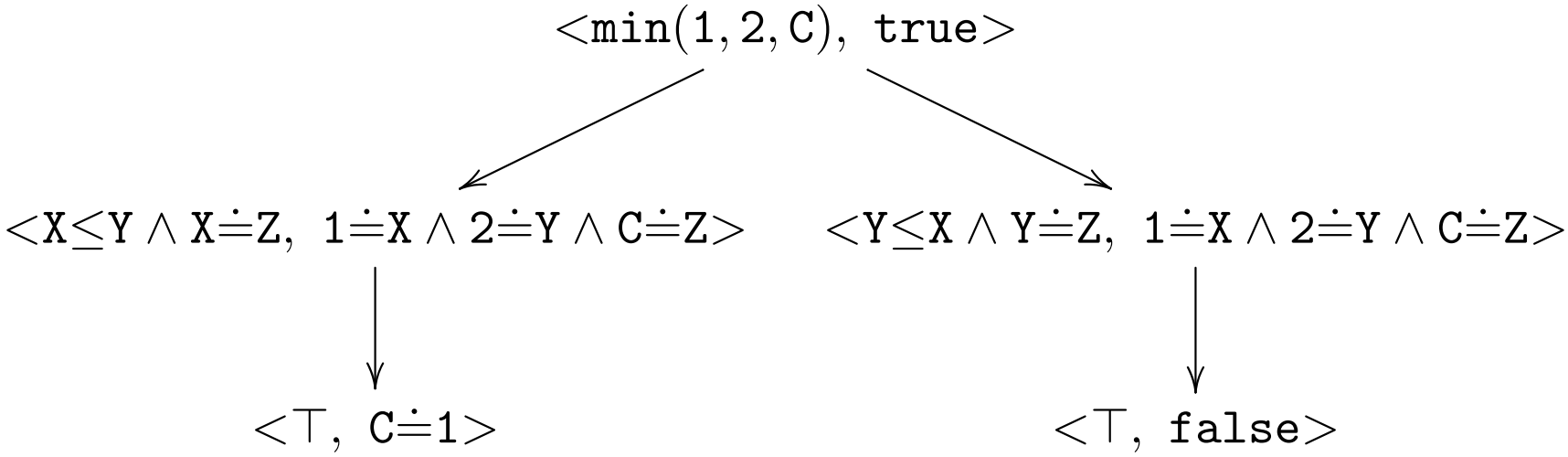
$$\texttt{min(X,Y,Z)} \leftarrow \texttt{X} \leq \texttt{Y} \wedge \texttt{X} \doteq \texttt{Z} \quad (c1)$$
$$\texttt{min(X,Y,Z)} \leftarrow \texttt{Y} \leq \texttt{X} \wedge \texttt{Y} \doteq \texttt{Z} \quad (c2)$$

$$\begin{array}{ll} & \texttt{<min}(1,2,\texttt{C}),\ \texttt{true>} \\ \mapsto_{\textbf{Unfold}\ (c1)} & \langle \texttt{X} \leq \texttt{Y} \wedge \texttt{X} \doteq \texttt{Z},\quad 1 \doteq \texttt{X} \wedge 2 \doteq \texttt{Y} \wedge \texttt{C} \doteq \texttt{Z} \rangle \\ \mapsto_{\textbf{Solve}} & \langle \top,\ \texttt{C} \doteq 1 \rangle \quad \text{(restricted to variables of initial state)} \end{array}$$

Using clause $(c2)$ leads to inconsistent constraint store
$2 \leq 1 \wedge 2 \doteq \texttt{C}$, the derivation fails.

CLP Example – Search Tree for `min(1,2,C)`

$$\texttt{min(X,Y,Z)} \leftarrow \texttt{X} \leq \texttt{Y} \wedge \texttt{X} \doteq \texttt{Z} \quad (c1)$$
$$\texttt{min(X,Y,Z)} \leftarrow \texttt{Y} \leq \texttt{X} \wedge \texttt{Y} \doteq \texttt{Z} \quad (c2)$$



CLP Example – Min (More Derivations)

$$\texttt{min(X,Y,Z)} \leftarrow \texttt{X}\leq\texttt{Y} \wedge \texttt{X}\doteq\texttt{Z} \quad (c1)$$
$$\texttt{min(X,Y,Z)} \leftarrow \texttt{Y}\leq\texttt{X} \wedge \texttt{Y}\doteq\texttt{Z} \quad (c2)$$

- Goal `min(A,2,1)`:

  $$\begin{array}{ll} & \texttt{<min(A,2,1), true>} \\ \mapsto_{\textbf{Unfold}\ (c1)} & \langle \texttt{X}\leq\texttt{Y} \wedge \texttt{X}\doteq\texttt{Z},\ \ \texttt{A}\doteq\texttt{X} \wedge \texttt{2}\doteq\texttt{Y} \wedge \texttt{1}\doteq\texttt{Z}\rangle \\ \mapsto_{\textbf{Solve}} & \langle \top,\ \ \texttt{A}\doteq\texttt{1}\rangle \end{array}$$

  but fails with $(c2)$.

- `min(A,2,2)` has answer $\texttt{A}\doteq\texttt{2}$ for $(c1)$, and $2 \leq A$ for $(c2)$

- `min(A,2,3)` fails

CLP Example – Min (More Derivations 2)

$$\texttt{min(X,Y,Z)} \leftarrow \texttt{X}\leq\texttt{Y} \wedge \texttt{X}\doteq\texttt{Z} \quad (c1)$$
$$\texttt{min(X,Y,Z)} \leftarrow \texttt{Y}\leq\texttt{X} \wedge \texttt{Y}\doteq\texttt{Z} \quad (c2)$$

- `min(A,A,B)` using $(c1)$ (same answer with $(c2)$)

$$\begin{aligned} & <\texttt{min(A,A,B)},\ \texttt{true}> \\ \mapsto_{\textbf{Unfold}\ (c1)} & \langle \texttt{X}\leq\texttt{Y} \wedge \texttt{X}\doteq\texttt{Z},\quad \texttt{A}\doteq\texttt{X} \wedge \texttt{A}\doteq\texttt{Y} \wedge \texttt{B}\doteq\texttt{Z}\rangle \\ \mapsto_{\textbf{Solve}} & \langle \top,\ \texttt{A}\doteq\texttt{B}\rangle \end{aligned}$$

- General goal $\texttt{min(A,B,C)} \wedge \texttt{A}\leq\texttt{B}$
  - using $(c1)$: answer $\texttt{A}\doteq\texttt{C} \wedge \texttt{A}\leq\texttt{B}$
  - using $(c2)$: answer $\texttt{A}\doteq\texttt{C} \wedge \texttt{A}\doteq\texttt{B}$ (more specific)
- `min(A,B,C)` ?

CLP Answer Constraint

- *logical reading of a state* $<H, C>$: $\exists \bar{X}(H \wedge C)$
  - $<G, \mathit{true}> \mapsto^* <H, C>$
  - $\bar{X}$: variables which occur in H or C but not in G
- *answer (constraint) of a goal* G:
  logical reading of final state of derivation starting with $<G, \mathit{true}>$

Answer constraints of successful and failed final states:

- $\langle \top, C \rangle$ is $\exists \bar{X}(\top \wedge C)$, i.e. $\exists \bar{X} C$
- $\langle G, \mathit{false} \rangle$ is $\exists \bar{X}(G \wedge \mathit{false})$, i.e. false

CLP Example – Min (Logical Reading)

$$\texttt{min(X,Y,Z)} \leftarrow \texttt{X}\leq\texttt{Y} \wedge \texttt{X}\doteq\texttt{Z} \quad (c1)$$
$$\texttt{min(X,Y,Z)} \leftarrow \texttt{Y}\leq\texttt{X} \wedge \texttt{Y}\doteq\texttt{Z} \quad (c2)$$

$$\forall X_1 X_2 X_3 \big((\texttt{min}(X_1, X_2, X_3) \leftrightarrow (\exists Y_{11} Y_{12} Y_{13}\ Y_{11}\doteq X_1, Y_{12}\doteq X_2, Y_{13}\doteq X_3 \wedge Y_{11} \leq Y_{12} \wedge Y_{11}\doteq Y_{13}) \vee (\exists Y_{21} Y_{22} Y_{23}\ Y_{21}\doteq X_1, Y_{22}\doteq X_2, Y_{23}\doteq X_3 \wedge Y_{22} \leq Y_{21} \wedge Y_{22}\doteq Y_{23})) \big)$$

In short:

$$\forall X_1 X_2 X_3 \big(\texttt{min}(X_1, X_2, X_3) \leftrightarrow (X_1 \leq X_2 \wedge X_1 \doteq X_3) \vee (X_2 \leq X_1 \wedge X_2 \doteq X_3) \big)$$

CLP Declarative Semantics of P

Logical Reading: Union of $P^{\leftrightarrow}$ with a *constraint theory CT* (including CET)

CLP Successful derivations

- **Soundness:**
  If G has successful derivation with answer constraint C, then $P^{\leftrightarrow} \cup CT \models \forall(C \rightarrow G)$.
- **Completeness:**
  If $P^{\leftrightarrow} \cup CT \models \forall(C \rightarrow G)$ and C is satisfiable in CT, then there are successful derivations for G with answer constraints $C_1, \ldots, C_n$ s.t. $CT \models \forall(C \rightarrow (C_1 \vee \ldots \vee C_n))$.

(P CL program, G goal)

CLP Example – Completeness

P	$P^{\leftrightarrow}$
$\mathrm{p(X,Y)} \leftarrow \mathrm{X} \leq \mathrm{Y}$	$\forall X \forall Y\, p(X,Y) \leftrightarrow (X \leq Y \vee Y \leq X)$
$\mathrm{p(X,Y)} \leftarrow \mathrm{X} \geq \mathrm{Y}$	(CT total order $\leq$)

Completeness:
As $P^{\leftrightarrow} \cup CT \models \forall(\mathtt{true} \rightarrow \mathtt{p(X,Y)})$ there are successful derivations for the goal `p(X,Y)`. The answer constraints $\mathtt{X} \leq \mathtt{Y}$ and $\mathtt{X} \geq \mathtt{Y}$ of `p(X,Y)` satisfy $CT \models \forall(\mathtt{true} \rightarrow \mathtt{X} \leq \mathtt{Y} \vee \mathtt{X} \geq \mathtt{Y})$.

But: Each answer on its own is not sufficient:
$CT \not\models \forall(\mathtt{true} \rightarrow \mathtt{X} \leq \mathtt{Y})$ and $CT \not\models \forall(\mathtt{true} \rightarrow \mathtt{X} \geq \mathtt{Y})$.

CLP Failed derivations

Soundness and Completeness:

$P^{\leftrightarrow} \cup CT \models \neg\exists G$ if and only if

each fair derivation starting with $<G, \mathit{true}>$ fails finitely

(P CL program, G goal)

CLP Monotonicity (Stability Property)

Ensures correctness of computation step in any larger context.

If

- $<G, C> \mapsto <G', C'>$
- $C' \wedge D$ satisfiable

then also $<G \wedge H, C \wedge D> \mapsto <G' \wedge H, C' \wedge D>$.

Computation can be performed in any larger context or it will fail.

(D constraint, H goal)

CLP vs. LP – Don't Know Nondeterminism

- *generate-and-test in LP*: impractical, facts used in passive manner only
- *constrain-and-generate in CLP*: use facts in active manner to reduce the search space (constraints)

CLP Constrain/Generate vs. LP Generate/Test

Crypto-arithmetic Puzzle – Send More Money

$$\begin{array}{cccccc} & & S & E & N & D \\ + & & M & O & R & E \\ \hline = & M & O & N & E & Y \end{array}$$

Replace distinct letters by distinct digits,
numbers have no leading zeros.

LP Example – Send More Money (Generate/Test)

```
send([S,E,N,D,M,O,R,Y]) :-
     gen_domains([S,E,N,D,M,O,R,Y],0..9),
     labeling([],[S,E,N,D,M,O,R,Y]),
     S #\= 0, M #\= 0,
     all_distinct([S,E,N,D,M,O,R,Y]),
                  1000*S + 100*E + 10*N + D
     +            1000*M + 100*O + 10*R + E
     #= 10000*M + 1000*O + 100*N + 10*E + Y.
```

95,671,082 choices to find the solution

CLP Example – Send More Money (Constrain/Generate)

```
:- use_module(library(clpfd)).

send([S,E,N,D,M,O,R,Y]) :-
     gen_domains([S,E,N,D,M,O,R,Y],0..9),
     S #\= 0, M #\= 0,
     all_distinct([S,E,N,D,M,O,R,Y]),
                  1000*S + 100*E + 10*N + D
     +            1000*M + 100*O + 10*R + E
     #= 10000*M + 1000*O + 100*N + 10*E + Y,
     labeling([],[S,E,N,D,M,O,R,Y]).
```

CLP Example – Send More Money (Constrain/Generate 2)

send without Labeling

```
:- send([S,E,N,D,M,O,R,Y]).
M = 1, O = 0, S = 9,
E in 4..7,
N in 5..8,
D in 2..8,
R in 2..8,
Y in 2..8 ?
```

$$\begin{array}{cccccc} & & S & E & N & D \\ + & & M & O & R & E \\ \hline = & M & O & N & E & Y \end{array}$$

CLP Example – Send More Money (Constrain/Generate 3)

`send` without Labeling

```
:- send([9,4,N,D,M,O,R,Y]).
no
```

Propagation determines `N = 5`, `R = 8`, but fails as `D` has no possible value.

But second choice

```
:- send([9,5,N,D,M,O,R,Y]).
D = 7, M = 1, N = 6,
O = 0, R = 8, Y = 2
yes
```

already computes solution.

$$\begin{array}{cccccc} & & S & E & N & D \\ + & & M & O & R & E \\ \hline = & M & O & N & E & Y \end{array}$$

CLP Example – Send More Money (Solution)

$$
\begin{array}{cccccc}
 & & S & E & N & D \\
 & & 9 & 5 & 6 & 7 \\
 & & M & O & R & E \\
+ & & 1 & 0 & 8 & 5 \\
\hline
 & M & O & N & E & Y \\
= & 1 & 0 & 6 & 5 & 2
\end{array}
$$

Concurrent Distributed Programming

- *processes (agents)*:
  executed concurrently and interact with each other
- *communicate* and *synchronize*: sending and receiving messages
- *distributed systems*: network of processes
- can be intentionally non-terminating (e.g., operating systems, control)

Concurrent Constraint Logic Programming (CCLP)

- integrates ideas from concurrent LP and CLP
- communication: common constraint store (blackboard)
- processes: predicates
- (partial) messages: constraints
- communication channels: variables
- running processes: goals that place and check constraints on shared variables

CCLP Ask and Tell

- *tell*: *producer* adds/places constraint to the constraint store
- *ask*: *consumer* checks entailment (implication) of constraints from the store (but does not remove any constraint)

Example:

Goal		Constraint Store
tell	$\mathtt{X} \leq \mathtt{Y}$	$\mathtt{X} \leq \mathtt{Y}$
tell	$\mathtt{Y} \leq \mathtt{Z}$	$\mathtt{X} \leq \mathtt{Y} \wedge \mathtt{Y} \leq \mathtt{Z}$
ask	$\mathtt{X} \leq \mathtt{Z}$	$\mathtt{X} \leq \mathtt{Y} \wedge \mathtt{Y} \leq \mathtt{Z}$
ask	$\mathtt{Y} \leq \mathtt{X}$	$\mathtt{X} \leq \mathtt{Y} \wedge \mathtt{Y} \leq \mathtt{Z}$
tell	$\mathtt{Z} \leq \mathtt{X}$	$\mathtt{X} = \mathtt{Y} \wedge \mathtt{Y} = \mathtt{Z}$
ask	$\mathtt{Y} \leq \mathtt{X}$	$\mathtt{X} = \mathtt{Y} \wedge \mathtt{Y} = \mathtt{Z}$
ask	$\mathtt{X} > \mathtt{Z}$	$\mathtt{X} = \mathtt{Y} \wedge \mathtt{Y} = \mathtt{Z}$

CCLP Consequences of Concurrency

- decisions and resulting actions cannot be undone anymore
- search as in CLP languages and failure should be avoided
- *don't-know non-determinism* is replaced by *don't-care non-determinism*
- *committed choice*: just one arbitrary of the applicable clauses is chosen
- loss in expressiveness, gain in efficiency

CCLP Early History

1981	K. Clark and S. Gregory, Relational Language for Parallel Prog.
1982-94	Japanese Fifth-Generation Computing Project, KL1
1983	E. Shapiro, Concurrent Prolog, FCP (Flat Concurrent Prolog)
1983	K. Clark and S. Gregory, Parlog (Parallel Prolog)
1985	K. Ueda, GHC (Guarded Horn Clauses)
1987	M. Maher, ALPS language class
1989	V. Saraswat, CC language framework (Concurrent constraints)
1990	S. Haridi, AKL (Andorra Kernel Language)
1991	M. Hermengildo, CIAO (Parallel Multi-Paradigm Prolog Extension)
1992	G. Smolka, OZ/Mozart (integrates functions, objects, and constraints)

CCLP Syntax

- *(CCL) clause*: $A \leftarrow D \mid G$
  - *head* A: atom
  - *guard* D: constraint
  - *body* G: goal
- trivial guard "*true* |" may be omitted
- *CCL program*: finite set of CCL clauses

CCLP Syntax - Summary

Atom:	A, B	::=	$p(t_1, \ldots, t_n), \quad n \geq 0$
Constraint:	C, D	::=	$c(t_1, \ldots, t_n) \mid C \wedge D, \, n \geq 0$
Goal:	G, H	::=	$\top \mid \bot \mid A \mid C \mid G \wedge H$
CCL Clause:	K	::=	$A \leftarrow C \vert G$
CCL Program:	P	::=	$K_1 \ldots K_m, \; m \geq 0$

CCLP Transition Rules

Unfold

If $(B \leftarrow D \mid H)$ is a fresh variant of a clause in P with variables $\bar{X}$

and $CT \models \forall\, (C \rightarrow \exists \bar{X}((B \doteq A) \wedge D))$

then $< A \wedge G, C > \mapsto < H \wedge G, (B \doteq A) \wedge D \wedge C >$

Solve

If $CT \models \forall\, ((C \wedge D_1) \leftrightarrow D_2)$

then $< C \wedge G, D_1 > \mapsto < G, D_2 >$

CCLP Unfold

If	$(B \leftarrow D \mid H)$ is a fresh variant
	of a clause in P with variables $\bar{X}$
and	$CT \models \forall\,(C \rightarrow \exists \bar{X}((B \doteq A) \wedge D))$
then	$<A \wedge G, C> \mapsto <H \wedge G, (B \doteq A) \wedge D \wedge C>$

- *entailment test (Ask)*: checks implication of guard D by store C, i.e., $CT \models \forall\,(C \rightarrow D)$
- clause with head B and guard D is *applicable* to A in the context of constraints C, if $CT \models \forall(C \rightarrow \exists \bar{X}((B \doteq A) \wedge D))$
- *clause application*: A removed, body H added to goal store, equation $B \doteq A$ and guard D added to constraint store (tell)
- committed choice of a clause, cannot be undone
- implicit concurrency (atom selection)

CCLP Unfold – Matching/One-sided Unification

Clause Applicability Condition in **Unfold** transition rule

$$CT \models \forall(C \rightarrow \exists \bar{X}((B \doteq A) \wedge D))$$

Given the constraints of C, try to solve the constraints $(B \doteq A \wedge D)$ without further constraining (touching) any variable in A and C

- first check that A *matches* B
  - A is an instance of B
  - only allowed to instantiate variables from the clause $\bar{X}$ of B but not variables of A
- then check the guard D under this matching

$((B \doteq A)$: shorthand for equating arguments of B and $A)$

CCLP Example – Matching

$$CT \models \forall(C \rightarrow \exists\bar{X}((B \doteq A) \wedge D))$$

Matching $B \doteq A, C = \mathit{true}, D = \mathit{true}$

- $\exists X(p(X) \doteq p(a))$
- $\forall Y \exists X(p(X) \doteq p(Y))$

but not

- $\forall Y(p(a) \doteq p(Y))$

More Examples

- $CT \models \forall Y(Y \doteq a \rightarrow \exists X(p(X) \doteq p(Y)) \wedge X \doteq a)$
- $CT \models \forall Y(Y \doteq a \rightarrow (p(a) \doteq p(Y)))$
- $CT \not\models \forall Y, Z(Z \doteq a \rightarrow (p(a) \doteq p(Y)))$

CCLP Deadlock

Successful and failed derivations and goals as in CLP. But new final state *deadlocked*.

$<G, C>$ with G different from $\top$ and C different from *false* and no more transitions are possible

- consequence of having no **Failure** transition
- usually programming errors

CCLP Example – Flipping Coins

```
flip(Side) ← Side≐face1
flip(Side) ← Side≐face2
```

- *CLP:*
  `flip(Coin)` with two answers
- *CCLP:*
  output not determined: depending on selected clause
  e.g., `flip(face1)` can either be failed or successful
  (But implementations will usually fix the clause choice.)

CCLP Example – Min

$$\texttt{min(X,Y,Z)} \leftarrow \texttt{X}\leq\texttt{Y} \mid \texttt{X}\doteq\texttt{Z} \qquad (c1)$$
$$\texttt{min(X,Y,Z)} \leftarrow \texttt{Y}\leq\texttt{X} \mid \texttt{Y}\doteq\texttt{Z} \qquad (c2)$$

Guard variables correspond to generalized *input parameters* (per rule).

- For goal `min(1,2,C)` rule $c1$ is applicable since

  $$CT \models \forall(\mathit{true} \rightarrow \exists\, \texttt{X},\texttt{Y},\texttt{Z}((\texttt{1}\doteq\texttt{X} \wedge \texttt{2}\doteq\texttt{Y} \wedge \texttt{C}\doteq\texttt{Z}) \wedge \texttt{X} \leq \texttt{Y}))$$

  Derivation:

  $$\begin{array}{ll} & <\texttt{min}(\texttt{1},\texttt{2},\texttt{C}),\mathit{true}> \\ \mapsto_{\textbf{Unfold}\ (c1)} & \langle \texttt{X}\doteq\texttt{Z},\ \texttt{1}\doteq\texttt{X}\wedge\texttt{2}\doteq\texttt{Y}\wedge\texttt{C}\doteq\texttt{Z}\rangle \\ \mapsto_{\textbf{Solve}} & \langle \top,\ \texttt{C}\doteq\texttt{1}\rangle \end{array}$$

- Rule $c2$ is not applicable, the guard is not implied.

CCLP Example – Min (cont)

$$\texttt{min(X,Y,Z)} \leftarrow \texttt{X}\leq\texttt{Y} \mid \texttt{X}\doteq\texttt{Z} \qquad (c1)$$
$$\texttt{min(X,Y,Z)} \leftarrow \texttt{Y}\leq\texttt{X} \mid \texttt{Y}\doteq\texttt{Z} \qquad (c2)$$

- $<\texttt{min}(\texttt{A},2,1), \mathit{true}>$, `min(A,2,2)`, and `min(A,2,3)` deadlocked (completely solved in CLP, but with both CL clauses)
- $\texttt{min}(\texttt{A},\texttt{B},\texttt{C}) \wedge \texttt{A}\leq\texttt{B}$ leads to $\texttt{A}\leq\texttt{B} \wedge \texttt{A}\doteq\texttt{C}$ by selecting the first clause (two answers in CLP, second one was $\texttt{A}\doteq\texttt{B} \wedge \texttt{A}\doteq\texttt{C}$)
- What about $<\texttt{min}(\texttt{A},\texttt{A},\texttt{C}), \mathit{true}>$?

CCLP Example – Hamming

Hamming's Problem: compute ordered ascending sequence of all numbers whose only prime factors are 2, 3, or 5:

$1 \diamond 2 \diamond 3 \diamond 4 \diamond 5 \diamond 6 \diamond 8 \diamond 9 \diamond 10 \diamond 12 \diamond 15 \diamond 16 \diamond 18 \diamond 20 \diamond 24 \diamond 25 \diamond ...$

- `hamming(S)` with infinite sequence `S`
- pretend (infinite!) sequence `S` is already known
  (actually only start 1 is known)
- `mults` processes multiply the numbers in `S` with 2, 3, and 5
- `merge` processes combine results to (ordered, duplicate-free) sequence `S`

CCLP Example – Hamming (cont)

```
hamming(S) ←
  S≐1◇S1 ∧
  mults(S,2,S2) ∧ mults(S,3,S3) ∧ mults(S,5,S5) ∧
  merge(S2,S3,S23) ∧ merge(S5,S23,S1)

mults(X◇S,N,XSN) ← XSN≐X*N◇SN ∧ mults(S,N,SN)

merge(X◇In1,Y◇In2,XYOut) ← X=Y |
  XYOut≐X◇Out ∧ merge(In1,In2,Out)
merge(X◇In1,Y◇In2,XYOut) ← X<Y |
  XYOut≐X◇Out ∧ merge(In1,Y◇In2,Out)
merge(X◇In1,Y◇In2,XYOut) ← X>Y |
  XYOut≐Y◇Out ∧ merge(X◇In1,In2,Out)
```

CCLP Example – Hamming (cont)

- no base cases in recursion as sequences are infinite
- concurrent-process network, processes can be executed in parallel
- `mults` and `merge` processes synchronize themselves
- processes communicate via the shared sequence variables

CCLP Example – Hamming (cont)

Goal `hamming(S)`

`mults(1◇S1,2,S2)` with **Unfold** and **Solve** leads to
`S2≐2◇S2N ∧ mults(S1,2,S2N).`

Overall, the three `mults` processes yield
`S2≐2◇S2N ∧ S3≐3◇S3N ∧ S5≐5◇S5N ∧ mults(S1,2,S2N) ∧`
`mults(S1,3,S3N) ∧ mults(S1,5,S5N) ∧ merge(S2,S3,S23) ∧`
`merge(S5,S23,S1).`

`merge(S2,S3,S23)` can unfold with the second clause
`S23≐2◇S23N ∧ merge(S2N,S3,S23N).`

`merge(S5,S23,S1)` yields
`S1≐2◇SN ∧ merge(S5,S23N,SN).`

CCLP Declarative Semantics

- declarative semantics of CCL programs analogous to that of CL programs
- symbol "|" interpreted as conjunction:
  $A \leftarrow C \mid G$ corresponds to $A \leftarrow C \wedge G$
- Clark's Completion of a CCL program is as in CLP
- discrepancy between operational and declarative semantics, as CCLP goals have only one derivation (compared to CLP goals)

Soundness of successful derivations

- If G has a successful derivation with an answer constraint C, then $P^{\leftrightarrow} \cup CT \models \forall(C \rightarrow G)$.
- Analogous to soundness for CL programs.

(P CCL program, G goal)

Completeness, Flip Coin Example

Analogous to CLP completeness?

$\texttt{flip(Side)} \leftarrow \texttt{Side} \doteq \texttt{face1}$

$\texttt{flip(Side)} \leftarrow \texttt{Side} \doteq \texttt{face2}$

Declarative semantics $P^{\leftrightarrow}$:

$\texttt{flip(Coin)} \Leftrightarrow \texttt{Coin} \doteq \texttt{face1} \vee \texttt{Coin} \doteq \texttt{face2}$

- $P^{\leftrightarrow} \cup CT \models (\texttt{Coin} \doteq \texttt{face1} \rightarrow \texttt{flip(Coin)})$
- If $\texttt{flip(Coin)}$ has successful derivation with answer constraint C' then $CT \models \forall(\texttt{Coin} \doteq \texttt{face1} \rightarrow C')$?
- Wrong if $C' = \texttt{Coin} \doteq \texttt{face2}$

Completeness (and soundness for failed derivations) only for *deterministic* programs and *fair* derivations without deadlocks. (Not discussed further here.)

Constraint Handling Rules (CHR)

CHR: concurrent committed-choice constraint language

- originally designed for writing constraint solvers
  - multi-headed rules that transform constraints into simpler ones until they are solved
- used now for reasoning in computational logic
- as general-purpose concurrent constraint language
- as flexible production rule system with constraints

CHR Syntax

- generalization of CCLP calculus by non-atomic heads and two types of rules

Simplification rule:	$E \Leftrightarrow C \mid G$
Propagation Rule:	$E \Rightarrow C \mid G$

CHR Syntax (2)

Built-in Constraint:	C, D	::=	$c(t_1, \ldots, t_n) \mid C \wedge D, n \geq 0$
CHR Constraint:	E, F	::=	$e(t_1, \ldots, t_n) \mid E \wedge F, n \geq 0$
Goal:	G, H	::=	$\top \mid \bot \mid C \mid E \mid G \wedge H$
CHR Rule:	R	::=	$E \Leftrightarrow C \mid G \mid E \Rightarrow C \mid G$
CHR Program:	P	::=	$R_1 \ldots R_m, m \geq 0$

CHR constraints instead of (C)CLP atoms.

- *(built-in) constraint symbols* handled by predefined given constraint solvers
- *(user-defined) CHR constraint symbols* defined by a CHR program

CHR Operational Semantics

- *state* $<G, C>$: G goal (store), C (built-in) constraint (store)
- *initial state* $<G, true>$
- *successful final state* $<E, C>$:
  no transition is applicable, C is different from *false*
- *failed final state* $<G, false>$

CCLP deadlocked states vs. CHR successful states

- CHR constraints that could not be further simplified do not imply consistency of logical reading

CHR Transition Rules

- **Simplify** (extends **Unfold** of CCLP)
- **Propagate** (similar **Simplify**)
- **Solve** (as in CCLP)

CHR Transition Rule – Simplify

Simplify

If	$(F \Leftrightarrow D \mid H)$ is a fresh variant of
	a rule in P with variables $\bar{X}$
and	$CT \models \forall\, (C \rightarrow \exists \bar{X} (F \doteq E \wedge D))$
then	$<E \wedge G, C> \mapsto <H \wedge G, (F \doteq E) \wedge D \wedge C>$

- head F
  - *CCLP:* atom
  - *CHR:* user-defined CHR constraint
- $F \doteq E$: pairwise matching of the conjuncts of F and E

CHR Transition Rule – Propagate

Propagate

If	$(F \Rightarrow D \mid H)$ is a fresh variant of
	a rule in P with variables $\bar{X}$
and	$CT \models \forall\,(C \rightarrow \exists \bar{X}(F \doteq E \wedge D))$
then	$<E \wedge G, C> \mapsto <E \wedge H \wedge G, (F \doteq E) \wedge D \wedge C>$

- like **Simplify** except that it keeps constraints E
- never apply a rule second time to same constraints
  - ensures termination

CHR Transition Rule – Solve

Solve

If $CT \models \forall((C \wedge D_1) \leftrightarrow D_2)$

then $<C \wedge G, D_1> \mapsto <G, D_2>$

- as in CLP and CCLP

CHR Transition Rules – Summary

Simplify

If $(F \Leftrightarrow D \mid H)$ fresh variant of a rule with variables $\bar{X}$

and $CT \models \forall\,(C \rightarrow \exists \bar{X}(F \doteq E \wedge D))$

then $<E \wedge G, C> \mapsto <H \wedge G, (F \doteq E) \wedge D \wedge C>$

Propagate

If $(F \Rightarrow D \mid H)$ fresh variant of a rule with variables $\bar{X}$

and $CT \models \forall\,(C \rightarrow \exists \bar{X}(F \doteq E \wedge D))$

then $<E \wedge G, C> \mapsto <E \wedge H \wedge G, (F \doteq E) \wedge D \wedge C>$

Solve

If $CT \models \forall((C \wedge D_1) \leftrightarrow D_2)$

then $<C \wedge G, D_1> \mapsto <G, D_2>$

CHR Example – Partial Order Relation `leq`

`reflexivity @ X leq Y` $\Leftrightarrow$ `X`$\doteq$`Y | true` $(r1)$

`antisymmetry @ X leq Y` $\wedge$ `Y leq X` $\Leftrightarrow$ `X`$\doteq$`Y` $(r2)$

`transitivity @ X leq Y` $\wedge$ `Y leq Z` $\Rightarrow$ `X leq Z` $(r3)$

`idempotency @ X leq Y` $\wedge$ `X leq Y` $\Leftrightarrow$ `X leq Y` $(r4)$

(*true* and $\doteq$: given built-in constraints)

CHR Example - Partial Order Relation `leq` (2)

	$<\underline{\texttt{A leq B}} \wedge \underline{\texttt{C leq A}} \wedge \texttt{B leq C}, \texttt{true}>$
$\mapsto$**Propagate** $(r3)$	$<\texttt{A leq B} \wedge \texttt{C leq A} \wedge \underline{\texttt{B leq C}} \wedge \underline{\texttt{C leq B}}, \texttt{true}>$
$\mapsto$**Simplify** $(r2)$	$<\texttt{A leq B} \wedge \texttt{C leq A} \wedge \underline{\texttt{B}\doteq\texttt{C}}, \texttt{true}>$
$\mapsto$**Solve**	$<\underline{\texttt{A leq B}} \wedge \underline{\texttt{C leq A}}, \texttt{B}\doteq\texttt{C}>$
$\mapsto$**Simplify** $(r2)$	$<\underline{\texttt{A}\doteq\texttt{B}}, \texttt{B}\doteq\texttt{C}>$
$\mapsto$**Solve**	$<\top, \texttt{A}\doteq\texttt{B} \wedge \texttt{B}\doteq\texttt{C}>$

CHR program:

$\texttt{X leq Y} \Leftrightarrow \texttt{X}\doteq\texttt{Y} \mid \texttt{true} \quad (r1)$

$\texttt{X leq Y} \wedge \texttt{Y leq X} \Leftrightarrow \texttt{X}\doteq\texttt{Y} \quad (r2)$

$\texttt{X leq Y} \wedge \texttt{Y leq Z} \Rightarrow \texttt{X leq Z} \quad (r3)$

$\texttt{X leq Y} \wedge \texttt{X leq Y} \Leftrightarrow \texttt{X leq Y} \quad (r4)$

CHR Example – Min

$$\texttt{min(X,Y,Z)} \Leftrightarrow \texttt{X}\leq\texttt{Y} \mid \texttt{Z}\doteq\texttt{X} \qquad (r1)$$
$$\texttt{min(X,Y,Z)} \Leftrightarrow \texttt{Y}\leq\texttt{X} \mid \texttt{Z}\doteq\texttt{Y} \qquad (r2)$$
$$\texttt{min(X,Y,Z)} \Leftrightarrow \texttt{Z}<\texttt{X} \mid \texttt{Y}\doteq\texttt{Z} \qquad (r3)$$
$$\texttt{min(X,Y,Z)} \Leftrightarrow \texttt{Z}<\texttt{Y} \mid \texttt{X}\doteq\texttt{Z} \qquad (r4)$$
$$\texttt{min(X,Y,Z)} \Rightarrow \texttt{Z}\leq\texttt{X} \wedge \texttt{Z}\leq\texttt{Y} \qquad (r5)$$

($\doteq$, $\leq$ and $<$ built-in constraint symbols)

CHR Example – Min (2)

$$\begin{array}{ll} & <\texttt{min}(1,2,\texttt{M}),\texttt{true}> \\ \mapsto_{\textbf{Simplify}\ (r1)} & <\texttt{M}\doteq 1,\texttt{true}> \\ \mapsto_{\textbf{Solve}} & <\top,\texttt{M}\doteq 1> \end{array}$$

CHR program:

$$\begin{array}{lll} \texttt{min(X,Y,Z)} \Leftrightarrow \texttt{X}\leq\texttt{Y} \mid \texttt{Z}\doteq\texttt{X} & (r1) \\ \texttt{min(X,Y,Z)} \Leftrightarrow \texttt{Y}\leq\texttt{X} \mid \texttt{Z}\doteq\texttt{Y} & (r2) \\ \texttt{min(X,Y,Z)} \Leftrightarrow \texttt{Z}<\texttt{X} \mid \texttt{Y}\doteq\texttt{Z} & (r3) \\ \texttt{min(X,Y,Z)} \Leftrightarrow \texttt{Z}<\texttt{Y} \mid \texttt{X}\doteq\texttt{Z} & (r4) \\ \texttt{min(X,Y,Z)} \Rightarrow \texttt{Z}\leq\texttt{X} \wedge \texttt{Z}\leq\texttt{Y} & (r5) \end{array}$$

CHR Example – Min (3)

$$
\begin{array}{ll}
 & <\mathtt{min}(\mathtt{A},\mathtt{B},\mathtt{M}) \wedge \mathtt{A} \leq \mathtt{B}, true> \\
\mapsto_{\textbf{Solve}} & <\mathtt{min}(\mathtt{A},\mathtt{B},\mathtt{M}), \mathtt{A} \leq \mathtt{B}> \\
\mapsto_{\textbf{Simplify}\ (r1)} & <\mathtt{M} \doteq \mathtt{A}, \mathtt{A} \leq \mathtt{B}> \\
\mapsto_{\textbf{Solve}} & <\top, \mathtt{M} \doteq \mathtt{A} \wedge \mathtt{A} \leq \mathtt{B}>
\end{array}
$$

CHR program:

$$
\begin{array}{lll}
\mathtt{min(X,Y,Z)} \Leftrightarrow \mathtt{X} \leq \mathtt{Y} \mid \mathtt{Z} \doteq \mathtt{X} & & (r1) \\
\mathtt{min(X,Y,Z)} \Leftrightarrow \mathtt{Y} \leq \mathtt{X} \mid \mathtt{Z} \doteq \mathtt{Y} & & (r2) \\
\mathtt{min(X,Y,Z)} \Leftrightarrow \mathtt{Z} < \mathtt{X} \mid \mathtt{Y} \doteq \mathtt{Z} & & (r3) \\
\mathtt{min(X,Y,Z)} \Leftrightarrow \mathtt{Z} < \mathtt{Y} \mid \mathtt{X} \doteq \mathtt{Z} & & (r4) \\
\mathtt{min(X,Y,Z)} \Rightarrow \mathtt{Z} \leq \mathtt{X} \ \wedge \ \mathtt{Z} \leq \mathtt{Y} & & (r5)
\end{array}
$$

CHR Example – Min (4)

$$
\begin{array}{ll}
 & <\texttt{min}(\texttt{A},2,2),\texttt{true}> \\
\mapsto_{\textbf{Propagate}\ (r5)} & <\texttt{min}(\texttt{A},2,2) \wedge 2 \leq \texttt{A} \wedge 2 \leq 2,\texttt{true}> \\
\mapsto_{\textbf{Solve}} & <\texttt{min}(\texttt{A},2,2),2 \leq \texttt{A}> \\
\mapsto_{\textbf{Simplify}\ (r2)} & <2 \doteq 2, 2 \leq \texttt{A}> \\
\mapsto_{\textbf{Solve}} & <\top, 2 \leq \texttt{A}>
\end{array}
$$

(Deadlock in CCLP.)

CHR program:

$$
\begin{array}{ll}
\texttt{min(X,Y,Z)} \Leftrightarrow \texttt{X} \leq \texttt{Y} \mid \texttt{Z} \doteq \texttt{X} & (r1) \\
\texttt{min(X,Y,Z)} \Leftrightarrow \texttt{Y} \leq \texttt{X} \mid \texttt{Z} \doteq \texttt{Y} & (r2) \\
\texttt{min(X,Y,Z)} \Leftrightarrow \texttt{Z} < \texttt{X} \mid \texttt{Y} \doteq \texttt{Z} & (r3) \\
\texttt{min(X,Y,Z)} \Leftrightarrow \texttt{Z} < \texttt{Y} \mid \texttt{X} \doteq \texttt{Z} & (r4) \\
\texttt{min(X,Y,Z)} \Rightarrow \texttt{Z} \leq \texttt{X} \wedge \texttt{Z} \leq \texttt{Y} & (r5)
\end{array}
$$

CHR Example – Min (5)

$$
\begin{array}{ll}
 & <\texttt{min}(\texttt{A},\texttt{B},\texttt{M}),\texttt{A}\dot{=}\texttt{M}> \\
\mapsto_{\textbf{Propagate}\ (r5)} & <\texttt{min}(\texttt{A},\texttt{B},\texttt{M}) \wedge \texttt{M} \leq \texttt{A} \wedge \texttt{M} \leq \texttt{B},\texttt{A}\dot{=}\texttt{M}> \\
\mapsto_{\textbf{Solve}} & <\texttt{min}(\texttt{A},\texttt{B},\texttt{M}),\texttt{M} \leq \texttt{B} \wedge \texttt{A}\dot{=}\texttt{M}> \\
\mapsto_{\textbf{Simplify}\ (r1)} & <\texttt{A}\dot{=}\texttt{M},\texttt{A} \leq \texttt{B} \wedge \texttt{M} \leq \texttt{B} \wedge \texttt{A}\dot{=}\texttt{M}> \\
\mapsto_{\textbf{Solve}} & <\top,\texttt{M} \leq \texttt{B} \wedge \texttt{A}\dot{=}\texttt{M}>
\end{array}
$$

(Deadlock in CCLP.)

CHR program:

$$
\begin{array}{ll}
\texttt{min(X,Y,Z)} \Leftrightarrow \texttt{X}\leq\texttt{Y} \mid \texttt{Z}\dot{=}\texttt{X} & (r1) \\
\texttt{min(X,Y,Z)} \Leftrightarrow \texttt{Y}\leq\texttt{X} \mid \texttt{Z}\dot{=}\texttt{Y} & (r2) \\
\texttt{min(X,Y,Z)} \Leftrightarrow \texttt{Z}<\texttt{X} \mid \texttt{Y}\dot{=}\texttt{Z} & (r3) \\
\texttt{min(X,Y,Z)} \Leftrightarrow \texttt{Z}<\texttt{Y} \mid \texttt{X}\dot{=}\texttt{Z} & (r4) \\
\texttt{min(X,Y,Z)} \Rightarrow \texttt{Z}\leq\texttt{X} \wedge \texttt{Z}\leq\texttt{Y} & (r5)
\end{array}
$$

CHR Example – Min (6)

- `min(A,2,1)` leads to `A`$\doteq$`1` via $(r4)$
- `min(A,2,3)` leads to failure via $(r5)$

(Deadlock in CCLP.)

CHR program:

$$\texttt{min(X,Y,Z)} \Leftrightarrow \texttt{X}\leq\texttt{Y} \mid \texttt{Z}\doteq\texttt{X} \qquad (r1)$$
$$\texttt{min(X,Y,Z)} \Leftrightarrow \texttt{Y}\leq\texttt{X} \mid \texttt{Z}\doteq\texttt{Y} \qquad (r2)$$
$$\texttt{min(X,Y,Z)} \Leftrightarrow \texttt{Z}<\texttt{X} \mid \texttt{Y}\doteq\texttt{Z} \qquad (r3)$$
$$\texttt{min(X,Y,Z)} \Leftrightarrow \texttt{Z}<\texttt{Y} \mid \texttt{X}\doteq\texttt{Z} \qquad (r4)$$
$$\texttt{min(X,Y,Z)} \Rightarrow \texttt{Z}\leq\texttt{X} \wedge \texttt{Z}\leq\texttt{Y} \qquad (r5)$$

CHR Monotonicity

If

- $<G, C> \mapsto <G', C'>$
- D constraint
- H goal
- $CT \models \exists(C \wedge D)$

then also $<G \wedge H, C \wedge D> \mapsto <G' \wedge H, C' \wedge D>$.

CHR Parallelism

If

- $<G, C> \mapsto <G', C'>$
- $<H, D> \mapsto <H', D'>$
- $CT \models \exists(C \wedge D)$

then also $<G \wedge H, C \wedge D> \mapsto <G' \wedge H', C' \wedge D'>$.

Generalisation of Monotonicity.

CHR Declarative Semantics

- Clark's completion not used
- each CHR rule alone gives rise to a formula
- "strong" declarative semantics as CHR is concerned with
  - solving constraints (that always admit a logical reading)
  - not with defining arbitrary processes

(If CHR is used as a general-purpose programming language, another declarative semantics based on linear logic can be more useful.)

CHR Logical Reading

	CHR-Rule	Logical Reading
Simplify	$E \Leftrightarrow C \mid G$	$\forall \bar{X}\ (C \rightarrow (E \leftrightarrow \exists \bar{Y}\ G))$
Propagate	$E \Rightarrow C \mid G$	$\forall \bar{X}\ (C \rightarrow (E \rightarrow \exists \bar{Y}\ G))$

- *logical reading of CHR program P*: union of
  - conjunction of logical readings of its rules $\mathcal{P}$
  - *constraint theory CT* defining built-in constraint symbols
- *logical reading of state and definition of answer constraints*: same as in (C)CLP

($\bar{X}$: variables in E and C, $\bar{Y}$: variables that appear only in G)

CHR Soundness and Completeness

Equivalence of States

- *computable constraint of* G: logical reading of a state which appears in a derivation of G
- for all computable constraints C_1 and C_2 of G:

$$\mathcal{P} \cup CT \models \forall\ (C_1 \leftrightarrow C_2)$$

- no distinction between successful and failed derivations

(P: CHR program, G: goal)

CHR Soundness

If

- G has a derivation with answer constraint C

then $\mathcal{P} \cup CT \models \forall\ (C \leftrightarrow G)$.

(P: CHR program, G: goal)

CHR Completeness

If

- G with at least one finite derivation
- $\mathcal{P} \cup CT \models \forall\ (C \leftrightarrow G)$

then

- G has a derivation with answer constraint C'
- $\mathcal{P} \cup CT \models \forall\ (C \leftrightarrow C')$.

(P: CHR program, G: goal)

CHR Example – Completeness

CHR program P:

$$\mathtt{p} \Leftrightarrow \mathtt{p}$$

CHR Completeness does not hold if G has no finite derivation

- goal G: $\mathtt{p}$
- $\mathcal{P} \cup CT \models \mathtt{p} \leftrightarrow \mathtt{p}$
- but G has only one infinite derivation

CHR Soundness and Completeness for Failed Derivations

If goal G has a finitely failed derivation, then $\mathcal{P} \cup CT \models \neg\exists G$.

But the converse does not hold, e.g.

$$\texttt{p} \Leftrightarrow \texttt{q}$$
$$\texttt{p} \Leftrightarrow \texttt{false}$$

- $\mathcal{P} \cup CT \models \neg\texttt{q}$
- q has no finitely failed derivation, but a successful one

CHR Confluence

The answer of a query is always the same, no matter which of the applicable rules are applied.

- *confluent*: for all states S, S_1, and S_2
  If $S \mapsto^* S_1$ and $S \mapsto^* S_2$ then S_1 and S_2 are joinable.
- *joinable*: if there exist states T_1 and T_2 such that $S_1 \mapsto^* T_1$ and $S_2 \mapsto^* T_2$ and T_1 and T_2 are variants.

In general infinitely many states to test.

CHR Decidable Confluence with Terminating Programs

- *terminating CHR program*: there are no infinite derivations

A terminating CHR program is confluent if and only if all its critical states are joinable.

- S: critical ancestor state built from overlap of heads and guards of two rules R_1 and R_2
- $S \mapsto_{R_1} S_1$, $S \mapsto_{R_2} S_2$
- check joinability
  - $S_1 \mapsto^* T_1, S_2 \mapsto^* T_2$
  - T_1, T_2 are variants?

CHR Example – Confluence Flip Coin

$$\texttt{flip(Coin)} \Leftrightarrow \mathit{true} \mid \texttt{Coin} \doteq \texttt{face1},$$
$$\texttt{flip(Coin)} \Leftrightarrow \mathit{true} \mid \texttt{Coin} \doteq \texttt{face2}$$

The source state

$$<\texttt{flip(Coin)}, \mathit{true}>$$

yields the following target states (forming a critical pair):

$$<\texttt{Coin}\doteq\texttt{face1}, \top> \text{ versus } <\texttt{Coin}\doteq\texttt{face2}, \top>$$

These two states are not *joinable*, hence the program is not confluent.

CHR Example – Confluence Min

$$\texttt{min(X,Y,Z)} \Leftrightarrow \texttt{X} \leq \texttt{Y} \mid \texttt{X} \doteq \texttt{Z},$$
$$\texttt{min(X,Y,Z)} \Leftrightarrow \texttt{Y} \leq \texttt{X} \mid \texttt{Y} \doteq \texttt{Z}$$

Overlapping the two rules yields the source state

$$<\texttt{min}(\texttt{X},\texttt{Y},\texttt{Z}), X \leq Y \wedge Y \leq X>$$

and the critical pair

$$<X \doteq Z, X \leq Y \wedge Y \leq X> \text{ versus } <Y \doteq Z, X \leq Y \wedge Y \leq X>$$

Both states of the critical pair can yield

$$<\top, X \doteq Y \wedge Y \doteq Z>$$

Hence the program is confluent.

CHR Soundness and Completeness of Failed Derivations

If

- P: *terminating* and *confluent* CHR program
- G: goal with at least one answer constraint consisting of only built-in constraints

then

$\mathcal{P} \cup CT \models \neg\exists G$ if and only if each finite derivation starting with $<G, true>$ fails.

CHR$^\vee$: Adding Disjunction

- to subsume the expressive power of CLP:
  disjunctions on the right-hand sides of CHR rules
- entire application in a uniform language

Split

$$<(H_1 \vee H_2) \wedge G, C> \mapsto <H_1 \wedge G, C> \mid <H_2 \wedge G, C>$$

- additional transition **Split** for CHR$^\vee$, to deal with disjunction $\vee$

Example:

`min(`X`,`Y`,`$Z`$`) <=> true |` $((X \leq Y \wedge X \doteq Z) \vee (Y \leq X \wedge Y \doteq Z))$

Constraint System $(\Sigma, \mathcal{D}, CT, \mathcal{C})$

- *signature* Σ: contains constraint symbols, at least
  - *true*/0 and *false*/0, and
  - syntactic equality $\doteq/2$
- *domain (universe)* $\mathcal{D}$ together with interpretation of function and constraint symbols in Σ
- *constraint theory* CT *over* Σ:
  - non-empty
  - consistent
- *allowed constraints* $\mathcal{C}$:
  - all constraints from Σ
  - closed under existential quantification (or: variable renaming) and conjunction

Example – Constraint System E (Syntactic Equality)

- signature Σ: contains
  - countably infinitely many function symbols, including at least one constant
  - constraint symbols $\mathit{true}/0$, $\mathit{false}/0$, and $\doteq/2$
- domain $\mathcal{D}$: Herbrand universe
- constraint theory CT: CET (Clark's Equality Theory)
- allowed constraints

$$\mathcal{C} ::= \mathit{true} \quad | \quad \mathit{false} \mid s \doteq t \mid \exists \bar{x}\, \mathcal{C} \mid \mathcal{C} \wedge \mathcal{C}$$

(s, t: terms over Σ)

Constraint Theory

Completeness

For every constraint C, either $CT \models C$ or $CT \models \neg C$.

Satisfaction-completeness

For every allowed constraint $C \in \mathcal{C}$, either $CT \models \exists C$ or $CT \models \neg\exists C$.

Example:

- CET is complete
- If there were only finitely many function symbols in Σ
  - not complete
  - but satisfaction-complete

Constraint Solvers

- implement algorithms for solving allowed constraints in accordance with constraint theory
- collect the constraints that arrive *incrementally* from one or more running programs
- put constraints into the constraint store
- test satisfiability, simplify, and if possible solve constraints

Capabilities of Constraint Solvers

As required by constraint programming languages (solve transition and transition preconditions).

Reasoning Services (in order of importance)

- Satisfiability (Consistency) test
- Simplification
- Variable determination
- Variable projection/elimination
- Entailment test

Satisfiability (Consistency) test

The solver returns *false* if C is inconsistent: $CT \models \neg\exists C$.

- solver implements a *decision procedure* for satisfiability of allowed constraints
- *syntactic equality* and *linear polynomial equations* admit efficient satisfaction-complete algorithm
- *Boolean* satisfiability is NP-complete (there is no efficient algorithm), same for *finite domains*

Example:
$X{>}X$ is inconsistent, $X{>}Y$ is consistent

Simplification

The solver tries to transform a given constraint C into a logically equivalent, but simpler constraint D (solve transition):
$CT \models \forall(C \leftrightarrow D)$.

- simpler constraint can be handled more efficiently when new constraints arrive
- improve presentation of answer constraint
- what simpler exactly means depends on the constraint system (often in the eye of the beholder)
- finding most simple representation can be substantially harder than solving

Example:
$X{\leq}2 \wedge X{\leq}4$ is simplified into $X{\leq}2$
$2{*}X{=}6$ into $X{=}3$

Variable Determination

Detect that a variable X occurring in a constraint C can only take a unique value: $CT \models \forall(C \rightarrow X{=}v)$, where v is a value.

- special case of simplification
- important for representing answer constraints as solutions that give values to variables
- supports a simple way of communication between different constraint solvers via shared variables by exchanging values for those variables

Example:
$X{\leq}2 \wedge 2{\leq}X$ implies $X{=}2$
$X^2{=}X \wedge X < 1$ implies $X{=}0$

Variable Projection/Elimination

Eliminate a variable X by projecting a constraint C onto all other variables: $CT \models \exists X C \leftrightarrow D$, where D does not contain X.

- keep constraint store small
- simplify answer constraint by eliminating local variables
- but in some cases not possible, significant increase in size/number for constraints,

Examples – Variable projection/elimination

- Projection of $\exists Y(X{<}Y \wedge Y{<}Z)$ onto X and Z results in $X{<}Z$ over the reals and $X{+}1{<}Z$ over the integers
- Elimination of Y in
  $\exists Y(X_1{<}Y \wedge \ldots \wedge X_m{<}Y \wedge Y{<}Z_1 \wedge \ldots \wedge Y{<}Z_n)$
  yields $n{*}m$ constraints of the form $X_i{<}Z_j$ for
  $1 \leq i \leq m, 1 \leq j \leq n$
- In the constraint system E, in the formula $\exists Y(X{=}f(Y))$, the variable Y cannot be eliminated

Entailment test

Check whether a constraint C implies D: $CT \models \forall(C \rightarrow D)$?

- required for guard checks in concurrent constraint languages like CCLP and CHR
- (incomplete) entailment test can be implemented as follows:
  if $C \wedge D$ simplifies to C, then $C \rightarrow D$, or:
  if $C \wedge \neg D$ is inconsistent, then $C \rightarrow D$.

Example – Entailment test

$X<Y$ entails $X\leq Y$, but not vice versa.

- $X<Y \rightarrow X\leq Y$ entailed
  - $\texttt{solve}(X < Y \wedge X \leq Y) = (X < Y)$
  - $\texttt{solve}(X < Y \wedge X > Y) = \mathit{false}$
- $X\leq Y \rightarrow X < Y$ not entailed
  - $\texttt{solve}(X \leq Y \wedge X < Y) = (X < Y)$
  - $\texttt{solve}(X \leq Y \wedge X \geq Y) = (X = Y)$

Reasoning Services – Implementation

- variations of simplification that maintain a *normal form* of the constraints
- implement simplification efficiently – average time complexity should be a polynomial of low degree
- prefers incomplete implementation if complete one would take exponential time

Desirable Properties of Constraint Solvers

Constraint solver modeled as a function `solve` over allowed constraints.

- *Correct*: If $solve(C) = D$, then $CT \models \forall(C \leftrightarrow D)$
- *Satisfaction-complete*: If $CT \models \neg\exists C$, then $solve(C) = false$
- *Incremental*: $solve(solve(C) \wedge D) = solve(C \wedge D)$
  - ideally, the incremental computation $solve(solve(C) \wedge D)$ should not be more costly than $solve(C \wedge D)$
- *Independence of variable naming*: C and D are variants of each other, then $solve(C)$ and $solve(D)$ are variants

Properties of Constraint Solvers (2)

- *Congruence respecting*:

Associative	$solve((C \wedge D) \wedge E) = solve(C \wedge (D \wedge E))$
Identity	$solve(C \wedge true) = solve(C)$
Commutative	$solve(C \wedge D) = solve(D \wedge C)$

- *Idempotent*: $solve(solve(C)) = solve(C)$
  - *solve* computes a fixpoint
  - no gain in simplifying simplified constraints again
- *Canonical*: If $CT \models \forall(C \leftrightarrow D)$, then $solve(C) = solve(D)$
  - very strong condition, implying all previous ones

Terminating and confluent CHR program is idempotent, congruence respecting, and incremental.

Properties of Constraint Solvers (3)

Dependency on Variable Order

Algorithms based on variable elimination (e.g., solving linear polynomial equations) often rely on an order of variables. (Typical variable orders: chronological or alphabetical.)

- different variable orders may result in different simplified constraints, e.g., `X=Y` vs. `Y=X`
- properties that may not hold anymore
  - Independence of variable naming
  - Congruence respecting
  - Incremental
  - Canonical

Principles of Constraint-Solving Algorithms

Variety of algorithms, mostly adapted from artificial intelligence, graph theory, and operations research.

Two viewpoints

- variable-elimination (or: equation solving)
  - typically satisfaction-complete
- local-consistency (or: local-propagation)
  - interleaved with search to achieve completeness

Variable Elimination - Normal Form

Allowed constraints are *equations*.

- $e_1 = e_2$
  - *l.h.s. (left-hand side)* e_1
  - *r.h.s. (right-hand side)* e_2
- *Normal form*: $X{=}e$
  X variable, e expression of some specific syntactic form

Variable Elimination - Solved Form

Solved (normal) form or *solution*: logically equivalent form that

- determines variables (gives values to variables)
- is unique (if possible)

usually

- conjunction of syntactic equality constraints $X{=}v$
- X only l.h.s. occurrence of the variable

Examples:

- $X{=}Y \ \wedge \ Y{=}Z$ and $X{=}Z \ \wedge \ Z{=}Y$ in solved form
- $X{=}Y \ \wedge \ X{=}Z$ not in solved form, X occurs twice on l.h.s.
- Gaussian elimination for solving linear polynomial equations $X{=}7{-}Y \ \wedge \ X{=}3{+}Y$ leads to $X{=}5 \ \wedge \ Y{=}2$

Variable Elimination Algorithms

Eliminate multiple occurrences of variables.

- repeatedly choose an equation $X{=}e$
- replace all other occurrences of X by e
- simplify s.t. normal form is maintained
- termination result may rely on order of variables and expressions
- usually not confluent, i.e. solved form not unique, e.g.
  $X{=}7{-}Y{-}2Z \ \wedge \ X{=}3{+}Y$ may lead to
  $X{=}3{+}Y \ \wedge \ Y{=}2{-}Z$ or
  $X{=}5{-}Z \ \wedge \ Y{=}2{-}Z$

Local Consistency (Local Propagation)

Small fixed-size sub-problems of the initial problem considered repeatedly until a fixpoint is reached.

- sub-problems are simplified and new implied (redundant) constraints are computed (propagated) from them
- constraints are added hoping that they cause simplification

Examples:

- $X>Y \ \wedge \ Y>Z \ \wedge \ Z>X$
  The first two constraints imply $X>Z$ and together with $Z>X$ imply *false*, i.e., inconsistency is detected (satisfaction complete).
- $X \neq Y \wedge Y \neq Z \wedge X \neq Z$ no propagation for sub-problems of two constraints but inconsistent if only same two values possible for each variable.

Local Consistency – Flat Normal Form

Variables are the only arguments of functions.

- often required by local-consistency methods
- *flattening*: performing the opposite of variable elimination
  - replace each non-variable sub-expression by a new variable
  - equate this new variable with the sub-expression
- uniform treatment of allowed constraints but introduction of auxiliary variables
- consistency methods sensitive to representation of constraints, but no efficient way to find an optimal representation

Examples:

- $2X+Y>5$ flattened into $W>F \ \wedge \ T+Y=W \ \wedge \ 2X=T \ \wedge \ F=5$
- $X^2>3Y$ flattened into
  $L>R \ \wedge \ L=X^T \ \wedge \ T=2 \ \wedge \ R=DY \wedge D=3$

Local Consistency (Local Propagation) - Complexity and History

Polynomial number of small sub-problems.

- sub-problems in polynomial time then overall algorithm in polynomial time

Classical consistency algorithms first explored for *constraint networks* in artificial intelligence research in the late 1960s. Originally, the algorithms involved unary and binary constraints over finite sets of values only.

- *arc consistency* and *path consistency*: main classical algorithms

Search

Local-consistency methods combined with search to achieve satisfaction-completeness, i.e., *global consistency*.

- *branching*: introduce branches in *search tree*
- *case analysis*: *case splitting* by introducing *choices*
- exponential complexity to combinatorial and other NP-complete constraint problems, because dependencies between choices are not and cannot be fully taken care of
- interleaved with constraint solving (repeat until solution is found)
  - perform local propagation
  - constraint is simplified together with the existing constraints
  - search step performed, it adds a new constraint

Search Routines/Procedures

Labeling procedure or *enumeration procedure (routine)*: try possible values for a variable $X{=}v_1 \vee \ldots \vee X{=}v_n$

- labeling procedure will use heuristics to choose the next variable and value for labeling
- *variable ordering*: chosen sequence of variables
  first-fail principle: choose the most constrained variable first; will often lead to failure quickly, thus *pruning* the search tree early
- *value ordering*: next value for labeling a variable must be chosen

Constraint Systems – CHR$^\vee$

- CHR$^\vee$ for implementation
  - Prolog style disjunction in the body of rules for search
  - CHR rule is never applied a second time to (syntactically) same conjunction of constraints
- in implementations efficient is achieved by
  - using simpagation rules
  - relying on textual rule application order
  - compiler options
  - additional constraints in the guards of propagation rules

Constraint Systems – CHR$^\vee$

Mapping abstract syntax into concrete syntax

logical symbols	program code symbols
$\Leftrightarrow$	`<=>`
$\Rightarrow$	`==>`
$\wedge$	`,`
$\vee$	`;`
$=$	`=` (built-in syntactical equality of Prolog) or `eq` (CHR constraint)
$<, \leq, >, \geq, \neq$	`<, =<, >, >=, \=` (built-in arithmetic constraints) `lt, le, gt, ge, ne` (CHR constraints)

Boolean Algebra B (Propositional Logic)

Constraint System B

Domain

Truth values $\top$ and $\bot$

Signature

- Function symbols
  - Constants: truth values 0 and 1
  - Boolean operations:
    - Unary connective $\neg$
    - Binary connectives $\sqcap, \sqcup, \oplus, \rightarrow, \leftrightarrow$ (do not confuse with $\wedge$, $\vee$)
- Constraint symbols
  - Nullary symbols *true*, *false*
  - Binary symbol $=$ (for Boolean expressions)

Constraint System B (2)

Constraint theory

Instances of $X = Z$, $\neg X{=}Z$, and $X \odot Y{=}Z$ according to the following truth table, where $\odot \in \{\sqcap, \sqcup, \oplus, \rightarrow, \leftrightarrow\}$.

X	Y	$\neg X$	$X \sqcap Y$	$X \sqcup Y$	$X \oplus Y$	$X \rightarrow Y$	$X \leftrightarrow Y$
0	0	1	0	0	0	1	1
0	1	1	0	1	1	1	0
1	0	0	0	1	1	0	0
1	1	0	1	1	0	1	1

- translation into formulae: $0 \sqcup 0 = 0 \ldots$
- theory decidable and complete

Constraint System B (3)

Allowed atomic constraints

$$C ::= \mathit{true} \mid \mathit{false} \mid X = Z \mid \neg X = Z \mid X \odot Y = Z$$

(X, Y, Z: variables or truth values)

- allowed atomic constraints are in *flat normal form*: each constraint contains at most one logical connective
- $\neg X = Z$, $X \odot Y{=}Z$: *inputs* X, Y and *output* Z
- no independence of negated constraints, e.g. $(\mathit{true} \wedge \neg(X = 0) \wedge \neg(X = 1))$ (equality over finite signature)

Example:
$(X \sqcap Y) \sqcup Z{=}\neg W$ flattened into
$(U \sqcup Z{=}V) \wedge (X \sqcap Y{=}U) \wedge (\neg W{=}V)$

B – Local-Propagation Constraint Solver

Rules for $X \sqcap Y = Z$ (Boolean equality $=$) in relational form (built-in syntactic equality `=`):

```
a1 @ and(X,Y,Z) <=> X=0 | Z=0.
a2 @ and(X,Y,Z) <=> Y=0 | Z=0.
a3 @ and(X,Y,Z) <=> X=1 | Y=Z.
a4 @ and(X,Y,Z) <=> Y=1 | X=Z.
a5 @ and(X,Y,Z) <=> X=Y | Y=Z.
a6 @ and(X,Y,Z) <=> Z=1 | X=1,Y=1.
```

Analogous for other logical connectives.

- *value/constant propagation* (`a1`, `a2`, `a6`)
- propagate equalities between variables (`a3-a5`)

B – Local-Propagation Constraint Solver (2)

Query: `and(X,Y,Z), X = 0`

Answer: `X = 0, Z = 0`

Query: `and(X,Y,Z), neg(Y,Z), X=1`

Answer: `false`

Query: `and(X,Y,0), and(X,Z,V), and(Y,V,W)`

Answer: `and(X,Y,0), and(X,Z,V), and(Y,V,W)`

(but `W=1` yields `false`, therefore `W=0`)

B – Local-Propagation Constraint Solver (3)

- determining satisfiability is *NP-complete*
- solver is *terminating* and *confluent*, e.g. overlap `and(0,0,Z)`
- *complexity*: slightly worse than $O(c^2)$
  - at most c derivation steps
  - in each derivation step, check (each of the) $O(c)$ constraints
  - checking the applicability of one constraint against one rule in quasi-constant time
  - rule application in quasi-constant time (syntactical equality in quasi-constant time using classical *union-find algorithm*)
- complexity linear if only value propagation

(c atomic Boolean constraints in query)

B Example – Complexity

`and(X,Y1,Y2), and(X,Y2,Y3), ..., and(X,YC,YCC), X=Y1`

- c Boolean constraints
- `X=Y1` leads to `Y1=Y2` hence `X=Y2`
- after step `I`:
  - `X=Y1=Y2=...=YI`,
  - check all constraints with variables `X, Y1, Y2, ..., YI`
- hence all constraints must be checked
- complexity $O(c^2)$ due to equality between variables

B – Search

- incomplete (polynomial time cannot solve exponential problems), `and(X,Y,Z), and(X,Y,W), neg(Z,W)` inconsistency not detected
  `and(X,Y,0), and(X,Z,V), and(Y,V,W)` not detected `W=1`
- interleave constraint solving and search for efficency
  (in CHR: search part at end of query)
- *labeling procedure* `enum` in CHR$^\vee$

```
enum([])    <=> true.
enum([X|L]) <=> bool(X), enum(L).

bool(X)     <=> (X=0 ; X=1).
```

Example:

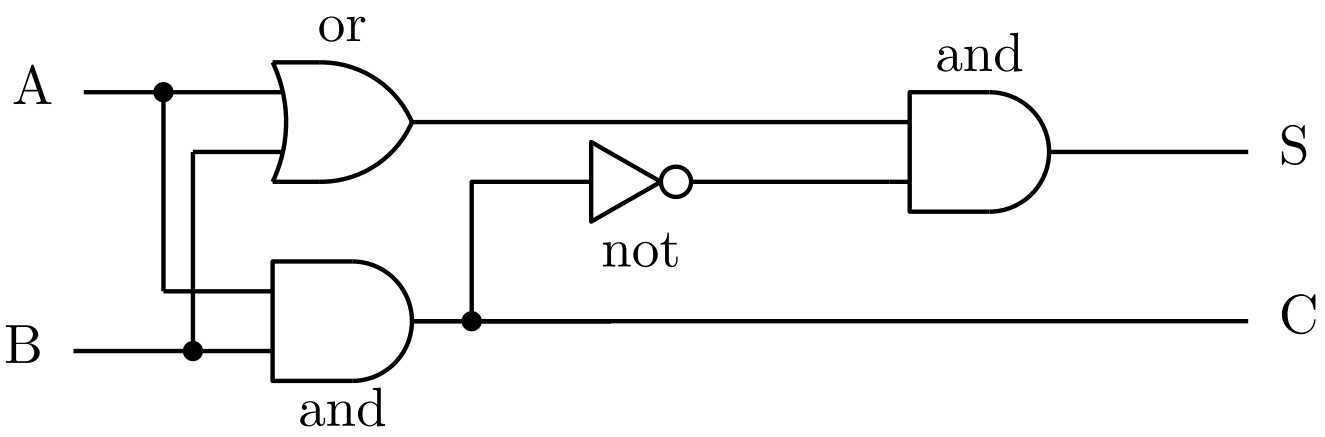
```
and(X,Y,Z), and(X,Y,W), neq(Z,W), enum([X,Y,Z,W])
```

B – Search Heuristic

Heuristics to improve the labeling (static or dynamic).

- *variable ordering*:
  - choose the variable that occurs most (as input)
  - hope that this will cause most simplification
  - this heuristic is an instance of the *first-fail principle*
- *value ordering*: to improve labeling
  - count the cases in which the values `0` and `1` cause simplification for a variable
  - e.g., choosing `0` for the last argument of `and` is not profitable
  - based on the counts, try one of the values first

B Example – Half-adder

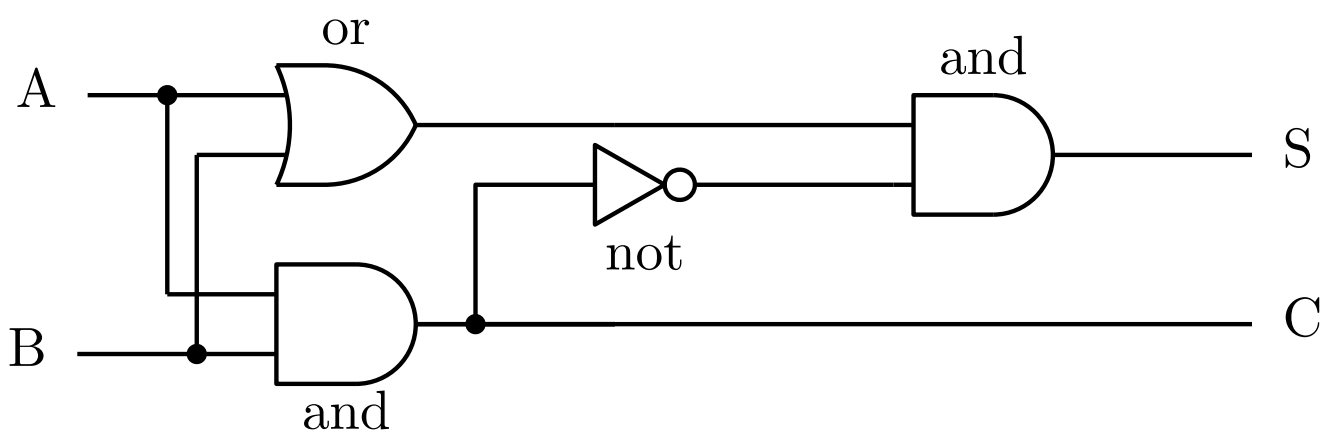


Expected behavior:

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

B Example – Half-adder (2)

```
add(A,B,S,C) <=>
  or(A,B,X1),
  and(A,B,C),
  neg(C,X3),
  and(X1,X3,S).
```



Query:	`add(1,0,S,C)`	Answer:	`S=1,C=0`
Query:	`add(A,B,S,C), C=1`	Answer:	`A=1, B=1, S=0, C=1`
Query:	`add(A,B,S,C), C=0`	Answer:	`or(A,B,S), and(A,B,0)`

B Application – Full-Adder Circuit

- adds three single-digit binary numbers `I1,I2,I3` (`I3`: *carry-in*), to a single number with two digits `O1,O2`, (`O2`: *overflow* or *carry-out*
- circuit consist of *(logical) gates*, which correspond to allowed atomic constraints

```
add(I1,I2,I3,O1,O2) <=>
        and(I1,I2,A1),
        xor(I1,I2,X1),
        and(X1,I3,A2),
        xor(X1,I3,O1),
        or(A1,A2,O2).
```

```
% or use two half adders
add(I1,I2,I3,O1,O2) <=>
        add(I1,I2,X01,X02),
        add(I3,X01,O1,Y02),
        or(X02,Y02,O2).
```

Example:

`add(I1,I2,I3,O1,O2),I3=0,O2=1` reduces to

`I3=0,O2=1,I1=1,I2=1,O1=0`

B Application – Fault Analysis of Full Adder

Each logical gate G_i is associated with a Boolean fault variable F_i

Gate works if it is not faulty: $\neg F_i \rightarrow G_i$

- If $F_i = 1$ then G_i is faulty
- If $F_i = 0$ then G_i is not faulty
- $\neg F_i \leftrightarrow G_i$ is too strict.

$\neg F_1 \rightarrow (I_1 \sqcap I_2 \leftrightarrow A_1)$	And gate G_1
$\neg F_2 \rightarrow (I_1 \oplus I_2 \leftrightarrow X_1)$	Xor gate G_2
$\neg F_3 \rightarrow (X_1 \sqcap I_3 \leftrightarrow A_2)$	And gate G_3
$\neg F_4 \rightarrow (X_1 \oplus I_3 \leftrightarrow O_1)$	Xor gate G_4
$\neg F_5 \rightarrow (A_1 \sqcup A_2 \leftrightarrow O_2)$	Or gate G_5

B – Application Fault Analysis of Full Adder (2)

Hypothesis: At most one gate is faulty:

$$\neg(F_1 \sqcap F_2) \sqcap \neg(F_1 \sqcap F_3) \sqcap \neg(F_1 \sqcap F_4) \sqcap \neg(F_1 \sqcap F_5) \sqcap \neg(F_2 \sqcap F_3) \sqcap$$
$$\neg(F_2 \sqcap F_4) \sqcap \neg(F_2 \sqcap F_5) \sqcap \neg(F_3 \sqcap F_4) \sqcap \neg(F_3 \sqcap F_5) \sqcap \neg(F_4 \sqcap F_5)$$

Inputs: $I_1 = 0, I_2 = 0$ and $I_3 = 1$

Outputs: $O_1 = 0$ and $O_2 = 1$

Solution: $F_1 = 0, F_2 = 1, F_3 = 0, F_4 = 0, F_5 = 0$ (Gate G_2 is faulty)

B – Application Fault Analysis of Full Adder (3)

```
faultanalysis(X,Y,Z,O1,O2,F1,F2,F3,F4,F5) <=>
    and(X,Y,A1),   xor(A1,I1,NF1),  imp(NF1,F1),
    xor(X,Y,XO1),  xor(XO1,I2,NF2), imp(NF2,F2),
    and(I2,Z,A2),  xor(A2,I3,NF3),  imp(NF3,F3),
    xor(Z,I2,XO1), xor(XO1,O1,NF4), imp(NF4,F4),
    or(I1,I3,OR2), xor(OR2,O2,NF5), imp(NF5,F5),

    and(F1,F2,0), and(F1,F3,0), and(F1,F4,0), and(F1,F5,0),
    and(F2,F3,0), and(F2,F4,0), and(F2,F5,0),
    and(F3,F4,0), and(F3,F5,0),
    and(F4,F5,0),

    enum([F1,F2,F3,F4,F5]).
```

($\neg F_i \rightarrow G_i$ has been rewritten to simplify implementation.)

Idea: replace `and`'s for `Fi`'s by single global constraint that counts.

B Example – Boolean Cardinality

`#(L,U,BL,N)` if between `L` and `U` `1`'s in the list `BL`.

- formula for counting true propositions is exponential in the number of involved propositions
- global constraint – admits arbitrary number of variables
- can express all other logical operators of form $\odot C_i = 1$, e.g.
  - negation `#(0,0,[C],1)` ($\rightarrow C = 0$)
  - exclusive-or (xor) `#(1,1,[C1,C2],2)` ($\rightarrow$ `neg(C1,C2)`)
  - at most one `#(0,1,[C1,...,Cn],N)` ($\rightarrow$ fault analysis)
  - conjunction `#(N,N,[C1,...,Cn],N)`
  - disjunction `#(1,N,[C1,...,Cn],N)`

Clauses expressible with disjunction and xor cardinality.

B Example – Boolean Cardinality (2)

```
triv_sat@ #(L,U,BL,N) <=> L=<0,N=<U | true.

pos_sat @ #(L,U,BL,N) <=> L=N | all(1,BL).

neg_sat @ #(L,U,BL,N) <=> U=0 | all(0,BL).

pos_red @ #(L,U,BL,N) <=> delete(1,BL,BL1) |
                          0<U, #(L-1,U-1,BL1,N-1).

neg_red @ #(L,U,BL,N) <=> delete(0,BL,BL1) |
                          L<N, #(L,U,BL1,N-1).
```

B – Consistency Approaches for Solving Bool

- *Generic Consistency Methods (Local Propagation)*
  translate into constraints over finite integer domains
- *Theorem Proving*
  - *SAT problem*: propositional Boolean constraint problem in clausal form
  - transformation into clauses in linear time and vice versa
  - 2-SAT and Horn-SAT is linear time
  - already 3-SAT problem is NP-complete (max. 3 variables per clause)
  - currently most successful algorithms based on randomly flipping truth value of a variable

B – Variable Elimination Approaches for Bool

- *Integer Programming*
  - translate into linear polynomial equations over integers
  - solved by operations research methods like linear programming
  - wide range of methods exist, but often not incremental
- *Boolean Unification*
  - extension of syntactic unification
  - computes a single, most general solution
  - needs auxiliary variables – exponential blow-up
    Example: For $X = Y \sqcap Z$, Z is not a function of X and Y ($X = Y = 0$), but $Z = X \oplus (\neg Y \sqcap V)$ (new variable V).
  - Boolean expressions encoded efficiently as *binary decision diagrams (BDD)*

Theorem Proving – Resolution

High-Level Implementation in CHR

- Syntax: Clause as *ordered* list of signed variables,
  e.g. $\neg x \vee y \vee z$ as `cl([-x,+y,+z])`
- Boolean CSP in CNF: Conjunction of clauses
- Clause: Disjunction of Literals
- Literal: Positive or negative atomic proposition

Theorem Proving – Resolution (2)

Resolution

```
empty_clause @ cl([]) <=> false.
idempotence @ cl(L) \ cl(L) <=> true.

tautology @ cl(L)     <=> in(+X,L),in(-X,L)| true.

res @ cl(L1), cl(L2)  ==> del(+X,L1,L3),del(-X,L2,L4)|
                          merge(L3,L4,L),
                          cl(L).
```

Auxiliary predicates:

`in(A,L)`: Element `A` occurs in list `L`.

`del(A,L1,L2)`: List `L1` without element `A` is `L2`.

`merge(L1,L2,L)`: Ordered union of `L1` and `L2` is `L3`.

B – Projection/Elimination (1)

Constraints in conjunctive normal form (CNF):

$$(Y \sqcap Z = X) \equiv (\neg X \sqcup Y) \sqcap (\neg X \sqcup Z) \sqcap (X \sqcup \neg Y \sqcup \neg Z)$$

Projection onto X and Z:

	$C^0 = \{\neg X \sqcup Z\}$	Y does not appear
Clauses:	$C^+ = \{\neg X \sqcup Y\}$	Y appears positively
	$C^- = \{X \sqcup \neg Y \sqcup \neg Z\}$	Y appears negatively

Resolution between C^+ and C^- (eliminate Y): $\{\neg X \sqcup \neg Z \sqcup X\}$

Result: $(\neg X \sqcup \neg Z \sqcup X) \sqcap (\neg X \sqcup Z) \equiv \neg X \sqcup Z \equiv X \to Z$

Repeated variable elimination can check satisfiability (of course, exponential).

B – Projection/Elimination (2)

Specialized Resolution Rule, simply add `eliminate`

```
eliminate(X),
cl(L1),cl(L2) ==> del(+X,L1,L3),del(-X,L2,L4)|
                  merge(L3,L4,L),
                  cl(L).
```

Variable removal

```
eliminate(X), cl(L) <=> (in(+X,L);in(-X,L)) | true.
eliminate(X) <=> true.
```

`eliminate(X)` at end of query

B – Projection/Elimination (3)

```
?-c([-x,+y]),c([-x,+z]),c([+x,-y,-z]), eliminate(y).
c([-(x),+(z)])

?-c([-x,+y]),c([-x,+z]),c([+x,-y,-z]), eliminate(z).
c([-(x),+(y)])

?-c([-x,+y]),c([-x,+z]),c([+x,-y,-z]),eliminate(x),eliminate(y).
true

?-c([-x,+y]),c([+x]),c([-y]),eliminate(x),eliminate(y).
false
```

Rational Tree (RT)

Finite and (certain) infinite terms.

Possibly infinite tree which has a finite set of subtrees.
Example: the infinite tree $f(f(f(\ldots)))$ only contains itself.

- no *occur-check* in early Prolog implementations for efficiency, unification could loop, e.g., $X \doteq f(X)$
  Prolog II: algorithm for resulting terms
- theory based on constraint system E (for finite trees)
- finite representations: as equality constraint, e.g., $X \doteq f(Y)$
  and as directed (possibly cyclic) graph, with function symbols as nodes and arcs from the term to its arguments (arcs labelled with argument position number)

Constraint System *RT*

Domain

Herbrand universe with Herbrand interpretation

Signature

- Infinitely many function symbols (at least one constant).
- Constraint symbols.
  - Nullary symbols *true*, *false*
  - Binary symbol for syntactic equality $\doteq$

Constraint System *RT* (2)

Constraint theory

Reflexivity	$\forall(true \rightarrow x \doteq x)$
Symmetry	$\forall(x \doteq y \rightarrow y \doteq x)$
Transitivity	$\forall(x \doteq y \wedge y \doteq z \rightarrow x \doteq z)$
Compatibility	$\forall(x_1 \doteq y_1 \wedge \ldots \wedge x_n \doteq y_n \rightarrow f(x_1, \ldots, x_n) \doteq f(y_1, \ldots, y_n))$
Decomposition	$\forall(f(x_1, \ldots, x_n) \doteq f(y_1, \ldots, y_n) \rightarrow x_1 \doteq y_1 \wedge \ldots \wedge x_n \doteq y_n)$
Contradiction (Clash)	$\forall(f(x_1, \ldots, x_n) \doteq g(y_1, \ldots, y_m) \rightarrow false)$ if $f \neq g$ or $n \neq m$

Constraint System *RT* (3)

Constraint theory (cont)

- CET without occur-check (acyclicity)
  - decidable
  - satisfaction-complete
  - but not complete, e.g. $\exists X, Y (X \doteq f(X) \wedge Y \doteq f(Y) \wedge \neg X \doteq Y)$ (complete with Maher's *uniqueness axiom* about implied equalities)

Allowed atomic constraints

$$C ::= \mathit{true} \quad | \quad \mathit{false} \quad | \quad s \doteq t$$

(s, t: terms over Σ)

RT – Solved Normal Form

Conjunction of allowed constraints is *solved (in solved normal form)*:

- *false* or
- $X_1 \doteq t_1 \wedge \ldots \wedge X_n \doteq t_n$ $(n \geq 0)$ (X_i: variables, t_i: terms)
  - $X_1, \ldots, X_n$ pairwise distinct (to avoid clashes)
  - X_i different to t_j if $i < j$ (to avoid cycles)

Each variable occurs at most once on the left and on the right of an equation, but arbitrary often inside terms.

RT – Solved Normal Form (2)

Examples:

- *not* in solved normal form:
  $f(X, b) \doteq f(a, Y)$,
  $X \doteq t \wedge X \doteq s$,
  $X \doteq Y \wedge Y \doteq X$
- in solved normal form:
  logically equivalent but syntactically different forms:
  $X \doteq Y$ and $Y \doteq X$
  $X \doteq f(X)$ and $X \doteq f(f(X))$
  $X \doteq Y \wedge Y \doteq a$ and $X \doteq a \wedge Y \doteq a$

RT – Variable Elimination Constraint Solver

```
reflexivity   @ X eq X <=> var(X) | true.

orientation   @ T eq X <=> var(X),X<<T | X eq T.

decomposition @ T1 eq T2 <=> nonvar(T1),nonvar(T2) |
                  same_functor(T1,T2),
                  args2list(T1,L1),args2list(T2,L2),
                  same_args(L1,L2).

confrontation @ X eq T1, X eq T2 <=> var(X),X<<T1,T1=<<T2 |
                  X eq T1, T1 eq T2.
```

Satisfaction-complete: produces solved normal form (while not in solved normal form one of the rules is applicable)

RT – Variable Elimination Constraint Solver (2)

Auxiliary constraints

- For termination:
  `s<<t` : total strict order on terms that holds at least
  - if s and t are different variables
  - if depth of `s` is less than depth of `t`
    (*depth*: maximum nesting level of function symbols)
- For term manipulation:
  - `var(T)`, `nonvar(T)`: `T` is variable, function term
  - `same_functor(T1,T2)`: `T1` and `T2` have same function symbol and same arity (functor)
  - `args2list(T1,L1)`: `L1` is ordered list of arguments of term `T1`

(*size*: number of occurrences of variables and function symbols)

RT Example – Equating two Terms

	h(Y,f(a),g(X,a)) eq h(f(U),Y,g(h(Y),U)))
$\mapsto_{\texttt{decomposition}}\mapsto^*$	Y eq f(U), f(a) eq Y, g(X,a) eq g(h(Y),U)
$\mapsto_{\texttt{orientation}}$	Y eq f(U), Y eq f(a), g(X,a) eq g(h(Y),U)
$\mapsto_{\texttt{decomposition}}\mapsto^*$	Y eq f(U), Y eq f(a), X eq h(Y), a eq U
$\mapsto_{\texttt{orientation}}$	Y eq f(U), Y eq f(a), X eq h(Y), U eq a
$\mapsto_{\texttt{confrontation}}$	Y eq f(U), f(U) eq f(a), X eq h(Y), U eq a
$\mapsto_{\texttt{decomposition}}\mapsto^*$	Y eq f(U), U eq a, X eq h(Y), U eq a
$\mapsto_{\texttt{confrontation}}$	Y eq f(U), U eq a, X eq h(Y), a eq a
$\mapsto_{\texttt{decomposition}}\mapsto^*$	Y eq f(U), U eq a, X eq h(Y)

RT Example – Equating two Terms (2)

	X eq f(X), X eq f(f(X))
$\mapsto_{\texttt{confrontation}}$	X eq f(X), f(X) eq f(f(X))
$\mapsto_{\texttt{decomposition}}\mapsto^*$	X eq f(X), X eq f(X)
$\mapsto_{\texttt{confrontation}}$	X eq f(X), f(X) eq f(X)
$\mapsto_{\texttt{decomposition}}\mapsto^*$	X eq f(X), X eq X
$\mapsto_{\texttt{reflexivity}}$	X eq f(X)

Non-termination without the proper order guards in `confrontation`:

	X eq f(X), X eq f(f(X))
$\mapsto_{\texttt{confront/wrong}}$	f(f(X)) eq f(X), X eq f(f(X))
$\mapsto_{\texttt{decomposition}}\mapsto^*$	f(X) eq X, X eq f(f(X))
$\mapsto_{\texttt{orientation}}$	X eq f(X), X eq f(f(X))

RT Solver – Correctness

- logical readings of `reflexivity` and `orientation` are consequences of the corresponding axioms *Reflexivity* and *Symmetry*, respectively.
- `decomposition` is a consequence of the axioms *Compatibility*, *Decomposition*, and *Contradiction (Clash)*
  - `same_functor` fails if there is a clash
- `confrontation` is a consequence of *Transitivity* and *Symmetry*
  - rule chosen over transitivity for efficiency (it does not increase the number of equations)
  - performs limited amount of variable elimination by only considering l.h.s. of equations

RT Solver – Confluence

No critical pairs/overlaps between different rules (disjoint guards), but overlap of confrontation rule with itself

```
confrontation @ X eq T1, X eq T2 <=> var(X),X<<T1,T1=<<T2 |
                    X eq T1, T1 eq T2.
```

Example of critical ancestor state:
`X eq T1, X eq T2, X eq T3,var(X),X<<T1,T1=<<T2,T2=<<T3`

- one possible derivation: $\mapsto$ `X eq T1, T1 eq T2, X eq T3` $\mapsto$ `X eq T1, T1 eq T2, T1 eq T3` (`T1` variable or not?)
- another possible derivation: $\mapsto$ `X eq T1, X eq T2, T2 eq T3` $\mapsto$ `X eq T1, T1 eq T2, T2 eq T3`

In practice, both states will lead to the same final state (since $\mathtt{T}_i$ are known and can be ordered).

Termination and Complexity

Each rule can be applied in constant time (with indexing on left side variables, for `confrontation` rule).

`reflexivity`, `orientation` rule apply at most once to an equation.

`decomposition` produces clash or finite number of equations with terms of smaller size and of smaller depth.

`confrontation` applies at most v times to second equation until left side is a function term (in the worst case). Equation size may double, but depth stays the same. Then only `decomposition` applies, reducing the depth.

Hence complexity is exponential in term depth in the worst case. Unclear if it can happen in practical implementations.

(v: number of different variables in problem)

Flattening

Flat constraints do not contain nested function terms.

Produce flat normal form for allowed constraints

- Opposite of variable elimination, introduce new auxiliary variables for nested terms
- Flattening function [.] transforms atomic equality constraint `eq` into conjunction of *flat* equations:
  $[S \texttt{ eq } T] := [X_0 \texttt{ eq } S] \wedge [X_0 \texttt{ eq } T]$

$$[X \texttt{ eq } T] := \begin{cases} X \texttt{ eq } T & \text{if } T \text{ is a variable} \\ X \texttt{ eq } f(X_1, \ldots, X_n) \wedge \bigwedge_{i=1}^{n} [X_i \texttt{ eq } T_i] & \text{if } T{=}f(T_1, \ldots, T_n) \end{cases}$$

(X variable, S function term, T term, $X_0 \ldots X_n$ new variables)

Flattening increases problem size by a constant factor only.

Flattening for Quadratic Complexity

With flat equations (which have at most one function symbol):

`decomposition` produces clash or equations between variables only.

`confrontation` applies at most v times to second equation until left side is a flat function term (in the worst case). Then only `decomposition` applies. Overall, a function term is replaced by equations for its argument variables.

At most one equation for each function symbol and each variable occurrence in the problem. Each equation costs at most v.

Hence complexity quadratic in problem size in the worst case.

(v: number of different variables in problem)

Flat function terms can be ordered arbitrarily.

Almost-linear time complexity possible with union-find algorithm.

RT Application – Program Analysis

In general, rational trees can represent directed graphs, e.g., automata.

Here we use rational trees to represent a recursive data type and to type check terms with it.

A *list* is defined recursively as

- constant `nil`
- binary list constructor `cons` applied to a term of type `Element` and to a list

Type of lists

```
List eq (nil or cons(Element,List)).
```

(`or`: binary infix operator separating alternatives)

RT Application – Program Analysis (2)

Type checker: `Term of Type` if `Term` is of type `Type`

```
Term of (Type1 or Type2) <=> (Term of Type1 ; Term of Type2).

Term of Type <=> nonvar(Term),nonvar(Type) |
        same_functor(Term,Type),
        args2list(Term,Args),args2list(Type,Types),
        check_args(Args,Types).

Term of Type, Type eq Type1 <=> var(Type) |
        Term of Type1, Type eq Type1.

check_args([],[])                       <=> true.
check_args([Arg|Args],[Type|Types]) <=>
          Arg of Type, check_args(Args,Types).
```

RT Application – Program Analysis (3)

- generate the type or lists with

  ```
  list(List,Element) <=> List eq (nil or cons(Element,List)).
  ```

- `list(NL,(0 or 1 or ...or 9))` lists over single-digit numbers
- `cons(3,cons(0,nil)) of NL` performs a type check:
  `cons(3,cons(0,nil)) of (nil or cons((0 or 1 or ... or 9), NL))`
  - either:
    `cons(3,cons(0,nil)) of nil` (fails)
  - or:

    ```
    cons(3,cons(0,nil)) of cons(0 or 1 or ...  or 9,NL)
    check_args([3,cons(0,nil)],[0 or 1 or ...  or 9,NL])
    3 of 0 or 1 or ...  or 9, cons(0,nil) of NL
    ```

Feature Terms (FTs)

Feature terms allow to model records in logic (originally from linguistics).

Motivation
Prolog Herbrand terms.
Pros: universal, flexible data structure
Cons: position-based access, hard to extend

- *features*: selectors (attributes/name-based access)
- *sort*: type name of record
- can be represented as graphs:
  sorts are nodes, features are arcs
- feature trees can represent rational trees and vice versa

Constraint System *FT*

Signature

- Infinitely many constants
- Constraint symbols:
  - unary sorts s
  - binary features f
  - $\doteq$, *true*, and *false*

Domain

Infinite set of individuals (constants).

Constraint System FT (2)

Constraint theory

Equivalence relation $=$ over variables and constants
(refl., symm., trans.)

$X\ f\ Y \wedge X\ f\ Z \Rightarrow Y = Z$

$a(X) \wedge b(X) \Rightarrow \mathit{false}$ if $a \neq b$

Allowed atomic constraints

$$C \quad ::= \quad \mathit{true} \ \mid \ \mathit{false} \ \mid \ s(X) \ \mid \ X\ f\ Y \ \mid \ X = Y$$

(s: sort, f: feature in infix notation, X, Y: variable or constant)

FT – Solved Normal Form

Conjunction of allowed constraints is *solved (in solved normal form)*:

- *false* (due to different sorts)
- if each variable comes in at most one sort constraint and at most once in the left hand side of an equation or particular feature constraint

FT – Solved Normal Form (2)

Example:
$person(X) \wedge X\ name\ Y \wedge\ Y = leo \wedge X\ livesIn\ Z \wedge munich(Z) \wedge$
$X\ name\ V \wedge V = sepp$

- by CT $V = Y$, replace V by Y
- by CT $Y = leo$ and $Y = sepp$ leads to $false$

Note: $person(X) \wedge X\ name\ Y \wedge Y = leo \wedge X\ livesIn\ Z \wedge munich(Z)$
can be compactly written as
`X:person(name=leo,livesIn=munich)`

FT – Constraint Solver

```
equal   @ X eq Y          <=> X=Y.
feature @ X.F.Y \ X.F.Z <=> Y=Z.
sort    @ X:S1  \ X:S2  <=> S1=S2.
```

- – $=$: `eq`, `=`
  - $s(X)$: `X:s`
  - $X\ f\ Y$: `X.f.Y`
- constraint solver uses built-in unification (`=`) between variables and constants
- `equal`: variable elimination
- `feature`: feature decomposition (functional)
- `sort`: sort intersection

FT – Constraint Solver (2)

FT – Properties of Solver

- Correctness: rules obvious logical consequences of corresponding axioms.
- Termination: obvious, since each rule removes one CHR constraint.
- No Confluence: no overlaps between different rules, but sort and feature rule can overlap with themselves, e.g., `X.F.Y, X.F.Z` yields `X.F.Y, Y=Z` or `X.F.Z, Z=Y`.
- Complexity: number of rule trials and applications linear in number of constraints if indexing is used for sorts and variable-feature pairs. Rule applications take constant time if built-in equality takes constant time (possible with union-find algorithm).

FT – Constraint Solver (3)

Extension: Negation

Added to allowed constraints:
$\neg s(X)$
$\neg X\ f$ (shorthand for $\forall Y \neg(X\ f\ Y)$)

Added to formulas of CT:
$s(X) \wedge \neg s(X) \Rightarrow \mathit{false}$
$X\ f\ Y \wedge \neg X\ f \Rightarrow \mathit{false}$

Added to rules of constraint solver:

```
X:S, not X:S <=> false.
X.F.Z, not X.F <=> false.
```

FT – Constraint Solver (4)

Extension: Arity Constraints

Herbrand-Term $X = g(Y, Z)$ can be written as
$g(X) \wedge X\ 1\ Y \wedge X\ 2\ Z$.

Forbid additional features like $X\ 3\ V$ by explicitly stating allowed features (here $X\{1, 2\}$) (also useful to explicitly disallow definition of new features, e.g. $X\{\mathit{livesIn}, \mathit{name}\}$.)

Added to allowed constraints:
$X\{f_1, \ldots, f_n\}$

(Variant: $S\{f_1, \ldots, f_n\}$, S sort not sufficient with free Herbrand terms)

FT – Constraint Solver (5)

Extension: Arity Constraints (cont)

Added to formulas of CT:
$X\ F \wedge X\ G \Rightarrow \mathit{false}$ if $F \neq G$
$X\ F \wedge X\ f\ Y \Rightarrow \mathit{false}$ if $f \notin F$

Added to rules of constraint solver (`Fs`, `G` ordered lists of features):

```
X in Fs \ X in G <=> Fs=G.
X in Fs, X.F.Y ==> member(F,Fs).
```

Another possible extension:
Ordered sorted feature trees allow hierarchies (Ait-Kaci et al. 93).

FT Example – Linguistics

FT use in knowledge representation in linguistics, e.g. to describe the contents of a sentence.

$$X = \exists Y \left[\begin{array}{l} \text{Person} \\ \text{office} : Y \\ \text{home} : Y \\ \text{name} : \left[\begin{array}{l} \text{first} : \mathit{Leo} \\ \text{last} : \mathit{Smith} \end{array}\right] \end{array}\right]$$

$\mathit{Person}(X) \wedge X\ \mathit{office}\ Y \wedge X\ \mathit{home}\ Y \wedge X\ \mathit{name}\ N \wedge N\ \mathit{first}\ S_1 \wedge S_1 = \mathit{Leo} \wedge N\ \mathit{last}\ S_2 \wedge S_2 = \mathit{Smith}$

Description Logics (DL)

Terminological Reasoning

Rich formalism for knowledge representation (originating in Brachman's KL-ONE, 1980s).

Concepts (unary constraints): properties like `human, car` (not disjunct).

Roles (binary constraints): relations like `child, age` (not functional dependent).

Graph representation: concepts are nodes, roles are arcs.

By default, concepts do not exclude each other.

Rich subset of first order logic to define concepts:

`parent isa human and some child is human`

Multiple inheritance possible. Related to type systems.

DL – Knowledge Representation

Two kinds of knowledge:

Terminological knowledge (T-Box)
General background knowledge of the application domain.
Intensional representation (like database scheme).
Concepts are defined by other concepts and restrictions on roles.
T-Box does not contain variables and individuals.

```
parent isa human and some child is human
```

Assertional knowledge (A-Box)
Concrete problem-specific knowledge.
Extensional representation (like database extension) that is checked against T-Box.
Statements about concepts and roles of individuals (objects).

```
sue:parent, (sue,joe):child
```

Constraint System DL

Domain

Infinite set of individuals (constants).

Signature

Infinitely many constants for individuals, concept and role names.
Function symbols *and, or, not, every, some, is* for building concept terms involving concept and role names.
Constraint symbols: *true*, *false*, *isa* (definitions) and ":" (assertions).

Allowed Constraints

$$C \quad ::= \quad true \quad | \quad false \quad | \quad I : s \quad | \quad (I, J) : r$$

A-Box: $I : s$ membership, $(I, J) : r$ role filler
T-Box: c *isa* s concept definition with concept name c implemented as a rule

(I, J: variables or individuals, r role name, s concept term)

Constraint System DL (2)

Concept Term s

A *concept (name)* c, or one of:

not s, s *and* t, s *or* t, *some* r *is* s, *every* r *is* s,

where s and t are concept terms.

Constraint Theory

$I : \mathit{not}\ S \leftrightarrow \neg(I : S)$

$I : S_1\ \mathit{and}\ S_2 \leftrightarrow I : S_1 \wedge I : S_2$

$I : S_1\ \mathit{or}\ S_2 \leftrightarrow I : S_1 \vee I : S_2$

$I : \mathit{some}\ R\ \mathit{is}\ S \leftrightarrow \exists J((I, J) : R \wedge J : S)$

$I : \mathit{every}\ R\ \mathit{is}\ S \leftrightarrow ((I, J) : R \rightarrow J : S)$ (also holds if no $(I, J) : R$)

$C\ \mathit{isa}\ S \leftrightarrow (I : C \leftrightarrow I : S)$

DL – Solved Normal Form

Conjunction of allowed constraints of form

- *false*
- $I : C_i$, $I : not\ C_j$ where C_i, C_j are primitive (undefined) concepts ($C_i \neq C_j$)
  $I : every\ R\ is\ S$, $(I, J) : R$ (unchanged)

DL – Constraint Solver (1)

Direct implementation of theory: A-Box as constraints, T-Box as rules. (Variant: T-Box as constraints)

Produces solved normal form that shows inconsistency. (Only membership constraints change.)

Split membership constraints:

```
I:S1 and S2 <=> I:S1, I:S2.
I:some R is S <=> (I,J):R, J:S.
I:every R is S, (I,J):R ==> J:S.
label, I:S1 or S2 <=> true | (I:S1 ; I:S2), label. % search
I:C \ I:C <=> true. % remove duplicates
```

DL – Constraint Solver (2)

Negation

```
I:not not S <=> I:S.
I:not (S1 and S2) <=> I:not S1 or not S2.
I:not (S1 or S2) <=> I:not S1 and not S2.
I:not (some R is S) <=> I:every R is not S.
I:not (every R is S) <=> I:some R is not S.
```

Clash Rule

```
I:not S, I:S <=> false.
```

For each concept definition C *isa* S (unfold definitions)

```
I:C <=> I:S.
I:not C <=> I:not S.
```

(T-Box as constraint: `C isa S \ I:C <=> I:S`).

DL – Family Relationships Example T-Box

```
male isa not female.
parent isa human and some child is human.
mother isa parent and female.
proud_parent isa parent and every child is phd.
```

Undefined concepts are called *primitive*:
`female`, `human` and `phd`.

Implicit concept hierarchy (taxonomy) with multiple inheritance:

```
parent<human, mother<parent, mother<female,
proud_parent<parent.
```

Notes:

Cyclic definitions (`female isa not male`)
Completion by `human isa female or male`

DL – Family Relationships Example A-Box

`sue:proud_parent, (sue,joe):child, joe:not phd.`

- Unfold concept definition `proud_parent`

`sue:parent and every child is phd,...`

- Unfold concept definition `parent` and split `and`

`sue:human, sue:some child is human, sue:every child is phd,...`

- Simplify *exists-in restriction* `some`

`sue:human, (sue,X):child, X:human, sue:every child is phd,...`

- Propagate *value restriction* `every`

`X:phd, joe:phd, sue:human, (sue,X):child, X:human, sue:every child is phd, (sue,joe):child, joe:not phd.`

- *Clash* between `joe:phd` and `joe:not phd`

Note: `X` may be `joe`, but need not to be.

DL – Properties

Confluence

Overlaps between `clash` and `duplicate` rules lead to easily joinable critical pairs.

Anytime and Online Algorithm

Computation can be stopped and restarted anytime while getting closer to solved form. Assertions can be added while program runs.

Parallelism and Concurrency

Rule heads consider only one membership assertion at a time, so can be executed in parallel. Program is confluent, so order does not matter. Propagation rules can be applied simultaneously.

DL – Termination

- Only rewritten constraints are memberships assertions
- Propagation rule: generates finite number of smaller membership assertions
  - Concept term: larger than its proper subterms
  - Atomic concept: larger than its defining concept term
  - Primitive concept: smallest
- Concept definitions are finite and acyclic by definition, so order is well-founded

DL – Complexity

- Worst-case time complexity is at least *exponential* due to disjunction in `or`, the propagation rule `every`, and repeated unfolding of same nonprimitive concept.
- For example, consider n T-box definitions in form for $(1 \leq i \leq n)$:
  $\texttt{C}_i$ `isa some r is` $\texttt{a}_i$ `and some r is` $\texttt{b}_i$ `and every r is` $\texttt{C}_{i-1}$
- Role-fillers form binary tree of depth n; each node has a child of concept a and a child of concept b. Tree is generated by exponential number of rule applications.

DL – Some Extensions

Top (universal) and bottom (empty) concepts:

```
X:top <=> true.          X:bot <=> false.
```

everysome quantifiers:

```
parent isa everysome child is human
I:everysome R is S <=> I:every R is S, I:some R is S
```

Role chains (nested roles):

```
grandfather isa father of father
(I,J):A of B <=> (I,K):A, (K,J):B
```

Inverse Roles:

```
(I,J):inv(R) <=> (J,I):R.
```

Transitive Roles:

```
(I,K):R, (K,J):R ==> transitive(R) | (I,J):R.
```

DL – Some Extensions II

Functional roles (features, attributes) e.g. F=age:

```
(I,J):F, (I,K):F ==> feature(F) | J=K.
```

Distinct, disjoint primitive concepts:

```
I:C1, I:C2 ==> distinct(C1), distinct(C2) | C1=C2.
```

Nominals (named individuals, singleton concepts):

```
X:{I1,I2,...,In} ==> (X=I1;X=I2;...;X=In).
```

Concrete domains (constraints from other domains):

```
(I,J):smaller ==> I<J.
(I,J):not_smaller ==> I>=J.
(I,A):flight_from, (I,B):flight_to ==> flight(I,A,B).
```

T-box axioms of inclusion between concept terms, written $C \sqsubseteq S$:

```
I:C ==> I:S.
```

DL – Applications

Assessing the contents of web pages (XML, OWL).

Represent medical knowledge (illnesses).

Configuration of technical systems (e.g. PC peripherals).

Description Logic with Rules

Combine DL with logic-based rules

- achieves more expressiveness
- allows role-filler assertions that do not have a tree structure, e.g. definition of `uncle` role (male sibling of person's father)

For atoms A_i and B_j, implications of the form:
$A_1 \vee \ldots A_n \leftarrow B_1 \wedge B_m$

can be translated into propagation rules with disjunction:
`B1,...,Bm ==> (A1 ; ...; An)`

Description Logic with Rules: Example

The `uncle` example yields the CHR rule:

```
Z:male, (Y,Z):hassibling, (X,Y):hasparent ==> (X,Z):hasuncle.
```

A concept definition in CHR:

```
X:uncle <=> (Y,X):hasuncle.
```

The single previous rule is too weak; CHR only applies rules left-to-right. Thus additional rules maybe needed:

```
X:uncle ==> (Y,X):hasuncle.
(Y,X):hasuncle ==> X:uncle.
```

Description Logic with Rules: Example II

Contrapositive of a conditional statement is formed by negating both terms and reversing the direction of inference.

CHR lacks reasoning with contrapositives, e.g. for the rule:

```
X:beer ==> sean:happy
```

the contrapositive can be added:

```
sean:not happy ==> X:not beer
```

Description Logic with Rules - Summary

Rule-based DL approaches translate to CHR propagation rules, whose closure is performed bottom-up.

The rule extension proposals for DL use sophisticated translations into FOL, making reasoning more difficult than in CHR.

Linear Polynomial Equations $\Re$

Constraint System $\Re$

Domain

The set $\Re$ of real numbers.

Signature

- Function symbols.
  - The real numbers 0 and 1
  - Unary prefix operators $+$ and $-$
  - Binary infix operators $+$ and $*$
- Constraint symbols.
  - Nullary symbols *true*, *false*
  - Binary symbols $=, <, \leq, >, \geq, \neq$

Constraint System $\Re$ (2)

Constraint theory

The linear existential fragment of Tarski's axiomatic theory of real closed fields for elementary geometry.

Allowed atomic constraints

Linear equations and inequations:

$$C ::= \mathit{true} \mid \mathit{false} \mid a_1 * X_1 + \ldots + a_n * X_n + b \odot 0, \quad (n \geq 0)$$

- coefficients $a_i, b \in \Re$, $a_i \neq 0$,
- variables $X_1, \ldots, X_n$ totally ordered in strictly descending order
- $\odot \in \{=, <, \leq, >, \geq, \neq\}$

The l.h.s. of the equation is called *(linear) polynomial*.

$\Re$ – Real closed fields

Decidable constraint theory for real arithmetic from Tarski.

C_1	$(x + y) + z = x + (y + z)$	associative $(+)$
C_2	$x + 0 = x$	neutral element $(+)$
C_3	$x + (-1 * x) = 0$	inverse element $(+)$
C_4	$x + y = y + x$	commutative $(+)$
C_5	$(x * y) * z = x * (y * z)$	associative $(*)$
C_6	$x * 1 = x$	neutral element $(*)$
C_7	$x \neq 0 \rightarrow \exists y\ x * y = 1$	inverse element $(*)$
C_8	$x * y = y * x$	commutative $(*)$
C_9	$x * (y + z) = (x * y) + (x * z)$	distributive
C_{10}	$0 \neq 1$	

$\Re$ – Real closed fields (2)

O_1	$\neg(x < x)$	irreflexive
O_2	$x < y \wedge y < z \rightarrow x < z$	transitive
O_3	$x < y \vee x = y \vee y < x$	total order
O_4	$x < y \rightarrow x + z < y + z$	
O_5	$0 < x \wedge 0 < y \rightarrow 0 < x * y$	
R_1	$0 < x \rightarrow \exists y\ y * y = x$	
R_2	$y_n \neq 0 \rightarrow \exists x\ y_n * x^n + y_{n-1} * x^{n-1} + \cdots + y_0 = 0$ for every odd n	

$\Re$ – **Tarski's theory**

- theory is complete and decidable, but intractable
- covers linear and non-linear polynomials
- linear existential fragment is decidable in polynomial time
- but only refers to the real numbers 0 and 1

$\Re$ – Variable Elimination Constraint Solver

- incremental variants of classical variable elimination algorithms
  - Gaussian elimination for equations (cubic complexity in number of different variables)
  - Dantzig's Simplex algorithm for equations and inequations (exponential worst case complexity but polynomial on average)
- implementation problems
  - reals: approximated by floating-point numbers, with unavoidable rounding errors (partial remedy: avoid using variables for elimination that have a small coefficient)
  - rationals: precise, size can grow exponentially in the size of the problem due to multiplication operations

$\Re$ – Variable Elimination Constraint Solver (2)

Conjunction of allowed constraints is solved (*in solved normal form*):

- *false* or
- left-most variable of each equation does not appear in any other equation

Compute solved form by eliminating multiple occurrences of variables, eliminate variables one by one:

- choose an equation $a_1 * X_1 + \ldots + a_n * X_n + b = 0$
- make its left-most variable explicit:
  $X_1 = -(a_2 * X_2 + \ldots + a_n * X_n + b)/a_1$
- replace all other occurrences of X_1 by
  $-(a_2 * X_2 + \ldots + a_n * X_n + b)/a_1$
- simplify resulting equations into allowed constraints

$\Re$ – Variable Elimination Constraint Solver (3)

Logical Form of Variable Elimination

$$a_1 * X_1 + \ldots + a_n * X_n + b_1 = 0 \ \wedge$$
$$a'_1 * X'_1 + \ldots + a'_i * X_1 + \ldots + a'_n * X'_n + b_2 = 0$$
$$\leftrightarrow$$
$$a_1 * X_1 + \ldots + a_n * X_n + b_1 = 0 \ \wedge$$
$$a'_1 * X'_1 + \ldots + a'_i * (-b_1 - (a_2 * X_2 \ldots + a_n * X_n))/a_1 + \ldots + a'_n * X'_n + b_2 = 0$$

$\Re$ – Variable Elimination Constraint Solver

```
eliminate @ A1*X+P1 eq 0 \ PX eq 0 <=>
            find(A2*X,PX,P2) |
            normalize(A2*(-P1/A1)+P2,P3),
            P3 eq 0.

empty @ B eq 0 <=> number(B) | zero(B).
```

Solver is satisfaction-complete since it produces the solved form.

(`A1`, `A2`, `B` coefficients; `X` variable; `P1`, `P2`, `PX` polynoms)

ℜ Example – Elimination

```
1*X+3*Y+5 eq 0, 3*X+2*Y+8 eq 0
```

- match `eliminate` rule
- `X` in second equation removed via
  `normalize(3*(-(3*Y+5)/1) + (2*Y+8), P3)`

```
1*X+3*Y+5 eq 0, -7*Y+ -7 eq 0
```

- match `eliminate` rule
- `Y` in first equation removed via
  `normalize(3*(-(-7)/-7) + (1*X+5), P3)`

```
1*X+2 eq 0, -7*Y+ -7 eq 0
```

$\Re$ – Variable Elimination Constraint Solver (6)

- *terminates*
  - finite number of variables for given constraint problem, no new variables are introduced during derivation
  - variables ordered strictly descending, variable of an equation is replaced by several strictly smaller ones
- *not confluent*
  - for two equations with same left-most variable, `eliminate` can be applied in two different ways, resulting in different pairs of equations

$\Re$ – Variable Elimination Constraint Solver (7)

$\Re - O(c^2v^2)$ Complexity

- at most cv occurrences of variables in a state of derivation (c and v do not increase during derivation)
- at most cv applications of `eliminate` rule (removes a single occurrence of a variable from one equation)
  - `find` complexity linear in v
  - `normalize` complexity linear in v
- $O(cv)$ complexity in trying to apply the rule to a given constraint
- $O(c)$ rule application attempts (at most c possible partner constraints), if indexing on variables is used
  - `find` complexity linear in v

(c equations, v different variables)

$\Re$ – Variable Elimination Constraint Solver (8)

$\Re$ – Determined Variables

```
determine  @ A*X+B eq 0 <=> number(B) | X is -B/A.
determined @ P eq 0     <=> find(A*X,P,P1), number(X) |
                            normalize(A*X+P1,P2), P2 eq 0.
```

$\Re$ – Inequations

- flatten inequation into equation and inequation on one *slack* variable
  - inequation replaced by equation with added *slack* variable
  - $P \odot 0$ flattened into $P = S \wedge S \odot 0$, where $\odot \in \{<, \leq, >, \geq, \neq\}$
  - Example: $P \geq 0$ is rewritten into $P = S \wedge S \geq 0$
- then apply solver for equations and ignore the inequations
- until values for the slack variables are known (determination)
- no longer satisfaction-complete: *slack-only equations* may be inconsistent even if different from *false* in solved form.

Example: $3 * S_1 + 4 * S_2 + 0 = 0 \wedge S_1 \geq 0 \wedge S_2 > 0$ is inconsistent

$\Re$ – Inequations (2)

Equation with only non-negative slack variables can be simplified if

- all coefficients have same sign

Examples:

- $3 * S_1 + 4 * S_2 + 0 = 0 \wedge S_1 \geq 0 \wedge S_2 \geq 0$
- $2 * S_1 + 3 * S_2 + 1 = 0 \wedge S_1 \geq 0 \wedge S_2 \geq 0$ is inconsistent

But: Incomplete as multiple equations are not considered:
$-S_1 + S_2 = 2 \ \wedge \ S_3 + S_2 = 1$ (variable S_2 not in first position)

$\Re$ – Inequations (3)

satisfaction-completeness in the presence of slack-only equations

- introduce a more strict solved form (as done in CHIP) (for $S_i \geq 0$)
  - slack variables in all equations *re*ordered (and *re*solved) s.t. coefficient of left-most slack variable has different sign than the constant
  - inconsistent, if this reordering is not possible
  - reordering may affect termination
- do more variable elimination to derive all implicit equalities (as done in CLP($\Re$)), e.g. Fourier Elimination

$\Re$ – Fourier Elimination

Set of linear inequations:

$$C = C^0 \cup C^+ \cup C^-.$$

Projection onto $var(C) \setminus \{Y\}$:

	C^0	Y does not appear
Clauses:	C^+	inequations $Y \leq t_1$ appear
	C^-	inequations $t_2 \leq Y$ appear

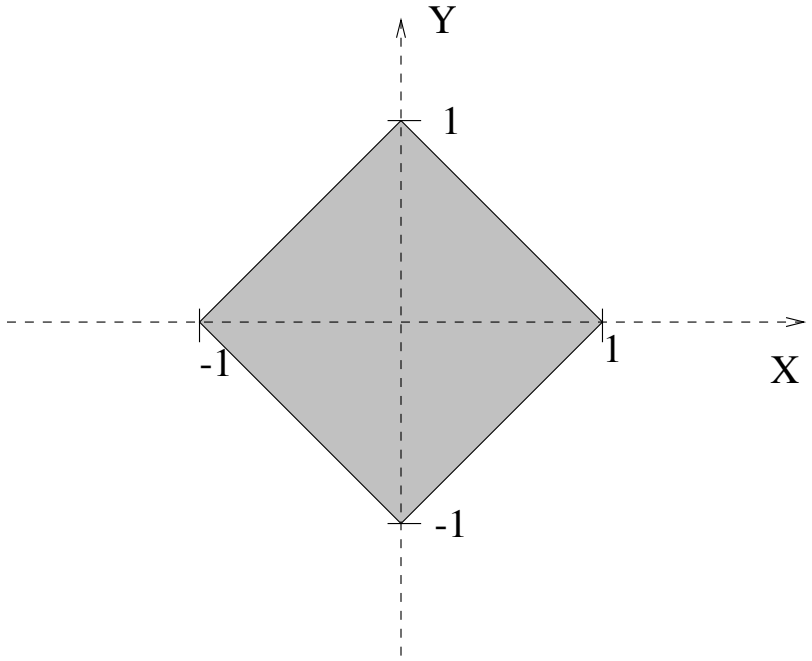
Resolution between C^+ and C^- (eliminate Y):

$C^Y = \{t_2 \leq t_1 \mid t_2 \leq Y \in C^-, Y \leq t_1 \in C^+\}$

Result:

$$C = C^0 \cup C^Y$$

ℜ Example – Fourier Elimination



Diamond:

$$\begin{array}{llll} X+Y & \leq 1 & X-Y & \leq 1 \\ -X+Y & \leq 1 & -X-Y & \leq 1 \end{array}$$

Split:

$C^0 = \{\}$

$C^+ = \{Y \leq 1 - X, Y \leq 1 + X\}$

$C^- = \{X - 1 \leq Y, -1 - X \leq Y\}$

Fourier elimination:

$$\begin{array}{llllll} X-1 \leq Y & \text{and} & Y \leq 1-X & \text{yield} & X \leq 1 \\ X-1 \leq Y & \text{and} & Y \leq 1+X & \text{yield} & -1 \leq 1 \\ -1-X \leq \mathrm{Y} & \text{and} & Y \leq 1-X & \text{yield} & -1 \leq 1 \\ -1-X \leq Y & \text{and} & Y \leq 1+X & \text{yield} & -1 \leq X \end{array}$$

ℜ Example – Fourier Elimination (2)

```
A1*X+P1 >= 0, XP >= 0 ==>
   find(A2*X,XP,P2),
   opposite_sign(A1,A2) |
   compute(P2-(P1/A1)*A2,P3),
   P3 >= 0.

B >= 0 <=> number(B) | geq(B).
```

Example:

```
1*X+1*Y+0 >= 0, -1*X+1*Y+0 >= 0

   compute((1*Y+0) - ((1*Y+0)/1)*-1,P3)

1*X+1*Y+0 >= 0, -1*X+1*Y+0 >= 0, 2*Y+0 >= 0
```

$\Re$ – Combination of Algorithms

Fourier's Algorithm for >=

```
A1*X+P1 >= 0, XP >= 0 ==>
   find(A2*X,XP,P2),
   opposite_sign(A1,A2) |
   compute(P2-
(P1/A1)*A2,P3),
   P3 >= 0.
```

Gaussian Elimination for =

```
A1*X+P1 = 0 \ XP = 0 <=>
   find(A2*X,XP,P2) |
   compute(P2-
(P1/A1)*A2,P3),
   P3 = 0.
```

Bridge Rule for = and >=

```
A1*X+P1 = 0 \ XP >= 0 <=>
   find(A2*X,XP,P2) |
   compute(P2-
(P1/A1)*A2,P3),
   P3 >= 0.
```

Redundant Inequations

```
A1*X+P1 >= 0 \ XP >= 0 <=>
   find(A2*X,XP,P2),
   \+opposite_sign(A1,A2),
   compute(P2-
(P1/A1)*A2,P3),
   number(P3), P3 >= 0 | true.
```

$\Re$ – Optimization and Related Approaches

- move from one solution to the next better one until an optimum is found, not directly suitable for implementation inside constraint solvers
  - *linear programming* with Simplex algorithm: find the solution that maximizes a given *objective function* (linear polynom)
  - *barrier or interior-point method*: non-linear programming
- symbolic arithmetic software packages like Mathematica, Maple and CPLEX
  - more solving power
  - not tightly integrated in a programming language for solving conjunctions of allowed constraints incrementally and efficiently
  - successfully loosely coupled with the CLP language Eclipse at IC-PARC and in the ILOG Optimization Suite

ℜ Application – Finance

```
% D: Amount of Loan, Debt, Principal
% T: Duration of loan in months
% I: Interest rate per month
% R: Rate of payments per month
% S: Balance of debt after T months

mortgage(D, T, I, R, S) <=>
        T = 0,
        D = S
          ;
        T > 0,
        T1 = T - 1,
        D1 = D + D*I - R,
        mortgage(D1, T1, I, R, S).
```

ℜ Application – Finance (2)

- `mortgage(100000,360,0.01,1025,S)` yields `S=12625.90`.
- `mortgage(D,360,0.01,1025,0)` yields `D=99648.79`.
- `S=<0, mortgage(100000,T,0.01,1025,S)`
  yields `T=374, S=-807.96`.
- `mortgage(D,360,0.01,R,0)` yields `R=0.0102861198*D`.
- If the interest rate `I` is unknown, the equation `D1 = D + D*I - R` will be non-linear after one recursion step, since `D1`, the new `D`, is not determined either.

Finite Domains *FD*

Basic Idea

- each variable takes its value from a given, finite set
- integers for values allows for arithmetic expressions as constraints
- constraint propagation proceeds by removing values from the sets of possible values that do not participate in any (partial) solution

FD – History

- many real-life combinatorial problems
  - scheduling and planning
- synthesis of LP and finite-domain constraint networks as explored in artificial intelligence research since the late 1960s.
- appeared in one of the first CLP languages CHIP
- Other influential languages clp(FD) and cc(FD)

Constraint System *FD*

Domain

The set $\mathcal{Z}$ of integers.

Signature

- Function symbols.
  - Constant 0 and unary successor function s
  - Lists (used for enumeration domains)
  - Binary infix operators + and .. (for interval domains)
- Constraint symbols.
  - Nullary symbols *true*, *false*
  - Binary symbols $=, <, \leq, >, \geq, \neq$, and *in* (for domains)

Interpretation maps arithmetic expressions to integers.

Constraint System *FD* (2)

Allowed atomic constraints

Linear equations and inequations:

$C ::= \mathit{true} \mid \mathit{false} \mid X \texttt{ in } n..m \mid X \texttt{ in } [k_1, \ldots, k_l] \mid X \odot Y \mid X{+}Y{=}Z$

$n,\ m,\ k_1, \ldots, k_l$: integers ($l \geq 0$)
$\odot \in \{=, <, >, \leq, \geq, \neq\}$
$X,\ Y$ and Z: pairwise distinct variables

domain constraint X `in` D: $X \in D$ (D: given finite domain)

- *enumeration domain constraint* X `in` $[k_1, \ldots, k_l]$:
  values of X explicitly enumerated
- *interval domain constraint* X `in` $n..m$:
  values of X in interval $n..m$ (bounds included)

Constraint System *FD* (3)

Constraint theory: Presburger Arithmetic

$$0 = s(X) \rightarrow \bot$$

$$X = X$$

$$X = Y \rightarrow s(X) = s(Y)$$

$$s(X) = s(Y) \rightarrow X = Y$$

$$X = Y \wedge Y = Z \rightarrow X = Z$$

$$X + 0 = X$$

$$X + s(Y) = s(X + Y)$$

Decidable and complete CT for integer linear arithmetic.

Peano arithmetic adds $*$ and induction principle – incomplete.

Gödel: Any consistent extension of Peano arithmetic is incomplete.

Constraint System *FD* (4)

Constraint theory

Presburger's arithmetic extended by

- $0 < s(X)$, $s(X) \leq s(Y) \leftrightarrow X \leq Y$, ...
- X `in` $n..m \leftrightarrow n \leq X \wedge X \leq m$
- X `in` $[k_1, \ldots, k_l] \leftrightarrow X{=}k_1 \vee \ldots \vee X{=}k_l$

Empty domain X `in` [] or X `in` $n..m$ with $n > m$ is unsatisfiable.

FD Interval vs. Enumeration Domain

Different implementations:

- interval domain
  - constraint simplification performed only on interval bounds
- enumeration domain
  - each element in the enumeration considered
  - allow more simplification (tighter domains)
  - only tractable for sufficiently small enumerations

Example:

- $X\ in\ [1,2,3] \wedge X{\neq}2$ yields tighter domain constraint $X\ in\ [1,3]$
- $X\ in\ 1..3 \wedge X{\neq}2$: no propagation since interval bounds do not change

FD Flat Normal Form, Linear Polynomial

- allowed atomic constraints in *flat normal form* and integers are not allowed in the place of variables
- determined variable ($X{=}v$) is expressed by a domain constraint $X\ in\ [v]$ or $X\ in\ v..v$.
- linear polynomial equation can be expressed as a conjunction of allowed constraints
  - multiply coefficients of polynomial s.t. they are all integers
  - rewrite multiplications as sums, e.g., $3X$ becomes $X + X + X$
  - flatten the resulting expression, e.g., $X{+}X{+}Y{>}5$ becomes $W{>}F \wedge X{+}V{=}W \wedge X{+}Y{=}V \wedge F\ in\ [5]$

Constraint Networks

Binary constraint network:

- Variables
- Binary constraints between variables

The network can be represented by a *directed constraint graph* where

- Nodes: variables
- Arcs: binary constraints

Solution of a constraint network:

Assignment of values to variables that satisfies all constraints

FD – Example Consistency

laender.ps

Map Coloring Problem

faerbe.ps

FD – Arc Consistency

Atomic constraint $c(X_1, \ldots, X_n)$ *(hyper-)arc consistent* with respect to a conjunction of enumeration domain constraints X_1 *in* $D_1 \ \wedge \ldots \wedge \ X_n$ *in* D_n, if for all $i \in \{1, \ldots, n\}$ and for all values v_i in D_i ($v_i \in \{k_{1i}, \ldots, k_{li}\}$) constraint $\exists(X_1$ *in* $D_1 \ \wedge \ldots \wedge X_i{=}v_i \wedge \ldots \wedge \ X_n$ *in* $D_n \ \wedge \ c(X_1, \ldots, X_n))$ is satisfiable, i.e. if for each variable in the constraint and for each value in the domain of the variable, there exist values in the domains of the other variables such that the constraint is satisfied. Every value of every domain takes part in a solution

($X_1, \ldots, X_n$: pairwise distinct variables)

FD – Arc Consistency (2)

Logically, each domain is the projection of the constraints.

$$\exists_{-X_i} (c(X_1, \ldots, X_n) \wedge X_1 \ in \ D_1 \wedge \cdots \wedge X_n \ in \ D_n) \rightarrow X_i \ in \ D_i$$

A conjunction of constraints is arc consistent if each atomic constraint in it is arc consistent.

FD – Arc Consistency (3)

Examples:

- $X\ in\ [1,2,3]\ \wedge\ X{\neq}2$ not arc consistent
  But: $X\ in\ [1,3]\ \wedge X{\neq}2$ arc consistent
- $X\ in\ [1,2,3]\ \wedge\ Y\ in\ [1,2,3]\ \wedge\ X{<}Y$ not arc consistent
  But: $X\ in\ [1,2]\ \wedge\ Y\ in\ [2,3]\ \wedge\ X{<}Y$ arc consistent

FD – Arc Consistency (4)

Limitations:

- must rename apart multiple occurrences of same variable
  $X{\neq}X$ represented by $X{\neq}Y \wedge X{=}Y$
- arc consistency does not imply satisfiability (global consistency):
  $X\ in\ D\ \wedge\ Y\ in\ D\ \wedge X{\neq}Y \wedge X{=}Y$ arc consistent for all $|D| > 1$
- arc consistency sensitive to flattening:
  $X\ in\ [1,2]\ \wedge\ Z\ in\ [2,3,4]\ \wedge\ 2X{=}Z$ not arc consistent
  But:
  $X\ in\ [1,2]\ \wedge\ Y\ in\ [1,2]\ \wedge\ Z\ in\ [2,3,4]\ \wedge\ X{=}Y\ \wedge\ X{+}Y{=}Z$
  arc consistent

FD – Arc Consistency (5)

Local Consistency Methods

Look at a small, fixed number of variables and constraints:

- Use them to propagate new, redundant constraints,
- Simplify the new constraints with old constraints
- Reconsider the changed old constraints until no more change (fixpoint reached)

Local consistency (Propagation) + Search (Labeling, Enumeration) = Complete Solver (Global consistency)

FD – Arc Consistency (6)

local-consistency algorithm = local propagation algorithm

- make atomic constraint arc consistent by deleting values from domain of its variables that do not participate in any solution
- make conjunction of constraints arc consistent by making each atomic constraint arc consistent
- worst case time complexity $O(cd^n)$ (c: number of at most n-ary constraints, d: size of largest domain)

FD – Arc Consistency (7)

Achieving Arc Consistency

An algorithm based on the logical formulation:

$X_1 :: D_1 \wedge \ldots \wedge X_i :: D_i \wedge \ldots \wedge X_n :: D_n$
$\wedge\ c(X_1, \ldots, X_n) \rightarrow X_i :: D'_i$
Then $D'_i \cap D_i$ is the new domain.

Example:

Domain constraints: $X :: [1,2,3], Y :: [1,2,3], Z :: [1,2,3]$
Binary constraints: $X < Y \wedge Y < Z \wedge Z \leq 2$

1. $X < Y$ implies $X :: [1,2]$ and $Y :: [2,3]$
2. $Z \leq 2$ implies $Z :: [1,2]$
3. $Y < Z$ implies $Y :: []$ and $Z :: []$

FD – Global Constraints

Take arbitrary number of variables as arguments

- *alldifferent* $(X_1, \ldots, X_n)$ logically equivalent to $\bigwedge_{1 \leq i < j \leq n} X_i \neq X_j$
- arc consistency does not detect unsatisfiability of $X_1 \neq X_2 \wedge X_1 \neq X_3 \wedge X_2 \neq X_3$ when the variables are constrained to the same domain of two values
- sophisticated algorithms for global constraints achieve more propagation than arc consistency and detect unsatisfiability in this case

FD – Bounds Consistency

For interval domains, a weaker but analogous form of arc consistency proves useful.

Atomic constraint $c(X_1, \ldots, X_n)$ is *bounds (or: box) consistent* with respect to a conjunction of interval domain constraints $X_1 \; in \; D_1 \; \wedge \ldots \wedge \; X_n \; in \; D_n$, if for all $i \in \{1, \ldots, n\}$ and for all bounds v_i in D_i ($D_i = n_i..m_i$, $v_i \in \{n_i, m_i\}$) the constraint $\exists(X_1 \; in \; D_1 \; \wedge \ldots \wedge X_i{=}v_i \wedge \ldots \wedge \; X_n \; in \; D_n \; \wedge \; c(X_1, \ldots, X_n))$ is satisfiable.

($X_1, \ldots, X_n$: pairwise distinct variables)

FD – Bounds Consistency (2)

A conjunction of constraints is bounds consistent if each atomic constraint in it is bounds consistent.

Analogously to arc consistency enforcement, constraints can be made bounds consistent by tightening their interval domains.

Examples:

- $X \; in \; 1..3 \; \wedge \; X \neq 2$ is bounds consistent
- – $X \; in \; 1..3 \; \wedge \; Y \; in \; 1..3 \wedge X < Y$ is not bounds consistent
  – $X \; in \; 1..2 \; \wedge \; Y \; in \; 2..3 \; \wedge \; X < Y$ is bounds consistent
- $X \; in \; D \; \wedge \; Y \; in \; D \; \wedge \; X \neq Y \; \wedge \; X = Y$ is bounds consistent for all $|D| > 1$

FD – Local-Propagation Constraint Solver for Interval Domains

- `in`, `le`, `eq`, `ne`, `add`: CHR constraints
- `<`, `>`, `=<`, `>=`,`\=`: built-in arithmetic constraints
- `min`, `max`, `+`, `-`: built-in arithmetic functions
- rules for bounds consistency affect interval in constraints only
- rules based on interval arithmetic

```
inconsistency @ X in A..B <=> A>B | false.
intersection  @ X in A..B, X in C..D <=>
                X in max(A,C)..min(B,D).
```

FD Interval Domains – Inequalities

Sample rules for inequalities:

```
le @ X le Y, X in A..B, Y in C..D <=> B>D |
     X le Y, X in A..D, Y in C..D.
le @ X le Y, X in A..B, Y in C..D <=> C<A |
     X le Y, X in A..B, Y in A..D.

eq @ X eq Y, X in A..B, Y in C..D <=> A\=C |
     X eq Y, X in max(A,C)..B, Y in max(C,A)..D.
eq @ X eq Y, X in A..B, Y in C..D <=> B\=D |
     X eq Y, X in A..min(B,D), Y in C..min(D,B).

ne @ X ne Y, X in A..B, Y in C..D <=> A=C,C=D |
     X ne Y, X in (A+1)..B, Y in C..D.
```

FD Interval Domains – Inequalities (2)

Example:

```
      A in 2..3, B in 1..2, A le B
↦le   B in 1..2, A le B, A in 2..2
↦le   A le B, A in 2..2, B in 2..2.
```

FD Interval Domains – Add

$X+Y=Z$ represented in relational form as `add(X,Y,Z)`:

```
add @ add(X,Y,Z), X in A..B, Y in C..D, Z in E..F <=>
      not (A>=E-D,B=<F-C,C>=E-B,D=<F-A,E>=A+C,F=<B+D) |
      add(X,Y,Z),
      X in max(A,E-D)..min(B,F-C),
      Y in max(C,E-B)..min(D,F-A),
      Z in max(E,A+C)..min(F,B+D).
```

Guard negates condition that describes arc consistency of `add`.

FD Interval Domains – Add (2)

Examples

- `A in 1..3, B in 2..4, C in 0..4, add(A,B,C)` $\mapsto_{\texttt{add}}$
  `add(A,B,C), A in 1..2, B in 2..3, C in 3..4`
- `X in 1..1000, Y in 1..1000, Z in 1..1000,`
  `X eq Z, add(X,Y,Z)` $\mapsto^*$
  `...`

FD Interval Domains – Termination

- rules `inconsistency` and `intersection` remove one interval constraint each
- assume that remaining rules deal with non-empty intervals only
- in each rule, at least one interval in the body is strictly smaller than the corresponding interval in the head, while the other intervals remain unaffected

FD Interval Domains – Confluence

The solver is confluent provided the intervals are given.

FD Interval Domains – Complexity (1)

- arithmetic built-in constraints take constant time to compute
- find domain of a variable in constant time using indexing
- each rule application or try takes constant time
- – $w = m - n + 1$: maximum *width (size)* of constraint $X \; in \; n..m$
  – v: number of different variables, c: number of constraints

  w, c and v do not increase during derivation

FD Interval Domains – Complexity (1)

- worst number of rule applications is $O(vw)$, not dependent on number of constraints (v can not exceed $O(c)$)
- there are at most $O(c)$ rule tries
- worst case time complexity is $O(cvw)$

Example:

```
   X in 1..1000, Y in 1..1000, X le Y, X eq Y
↦ X in 1..999, Y in 2..1000, X le Y, X eq Y
↦ ...
```

FD – Local-Propagation Constraint Solver for Enumeration Domains

```
inconsistency @ X in [] <=> false.
intersection  @ X in L1, X in L2 <=>
                intersection(L1,L2,L3), X in L3.

le @ X le Y, X in L1, Y in L2 <=> max(L1) > max(L2) |
     filter_max(L1,L2,L3),
     X le Y, X in L3, Y in L2.
...
```

(`filter_max` removes all values from a list that are larger than all values in another list)

FD Example – Enumeration Domains

- `X le Y, X in [4,6,7], Y in [3,7]` $\mapsto^*$
  `X le Y, X in [4,6,7], Y in [7]`
- `X le Y, X in [2,3,4,5], Y in [1,2,3]` $\mapsto^*$
  `X le Y, X in [2,3], Y in [2,3]`
- `X le Y, X in [2,3,4], Y in [0,1]` $\mapsto^*$
  `false`

FD Enumeration Domains – `eq` and `add`

```
eq @ X eq Y, X in L1, Y in L2 <=> diff(L1,L2) |
     intersection(L1,L2,L3),
     X eq Y, X in L3, Y in L3.

add @ add(X,Y,Z), X in L1, Y in L2, Z in L3 ==>
    all_substractions(L3,L2,L4),
    all_substractions(L3,L1,L5),
    all_additions(L1,L2,L6),
    not (L1 se L4, L2 se L5, L3 se L6),
    |
    X in L4, Y in L5, Z in L6.
```

(`se`: set equality)

FD Enumeration Domains – Termination, Confluence

Termination and confluence similar to interval domain solver.

FD Enumeration Domains – Complexity

- replace interval width w by maximum size of an enumeration domain d
- built-in constraints take up to $O(d^2)$ for operations on arbitrarily large enumeration domains
- worst time complexity is $O(cvd^3)$

FD – Search

Use search to achieve satisfaction-completeness.

```
enum([]) <=> true.
enum([X|Xs]) <=> indomain(X), enum(Xs).

indomain(X), X in [V|L] <=> L=[_|_] |
            (X in [V] ; X in L, indomain(X)).
```

For interval domains, search is usually done by splitting intervals in two halves until bounds of interval are the same

```
indomain(X), X in A..B <=> A<B |
             C is (A+B)//2,
            (X in A..C ; X in (C+1)..B), indomain(X).
```

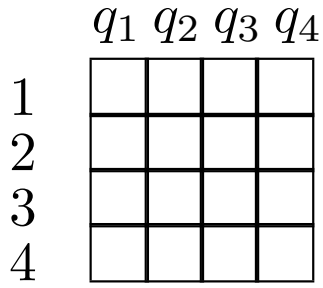
The guards ensure termination.

FD – Implementations

- linear polynomial equations are allowed as constraints
- hybrid, compact form of domains is used in implementations
- domain is list of intervals, so an interval can have holes since it can be split if a value inside the interval needs to be removed
- for small enumeration domains, bit vectors can be used

FD Application – n-Queens Problem

Place n queens $q_1, \ldots, q_n$ on an $n \times n$ chess board, such that they do not attack each other.



$$q_1, \ldots, q_n \in \{1, \ldots, n\}$$

$$\forall\ i \neq j.\ q_i \neq q_j \wedge |q_i - q_j| \neq |i - j|$$

- no two queens on same row, column or diagonal
  - each row and each column with exactly one queen
  - each diagonal at most one queen
- q_i: row position of the queen in the i-th column

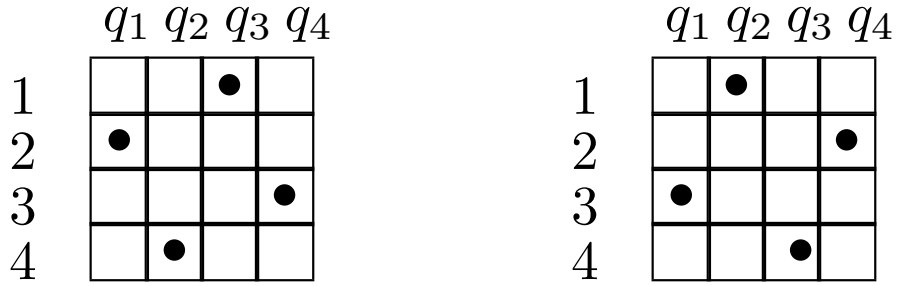
FD Application – n-Queens Problem (2)

```
constraints solve/2, queens/1, safe/3, no_attack/3.

solve(N,Qs)       <=> make_domains(Qs,N), queens(Qs), enum(Qs).

queens([])        <=> true.
queens([Q|Qs])    <=> safe(Q,Qs,1), queens(Qs).

safe(X,[],N)      <=> true.
safe(X,[Y|Qs],N) <=> no_attack(X,Y,N), NP1 is N+1,
                       safe(X,Qs,NP1).

no_attack(X,Y,N) <=> X ne Y, VN in N..N,
                       add(X,VN,XPN), XPN ne Y,
                       add(Y,VN,YPN), YPN ne X.
```

Application – n-Queens Problem (3)

`solve(4,[Q1,Q2,Q3,Q4])`

- `make_domains` produces
  `Q1 in [1,2,3,4], Q2 in [1,2,3,4]`
  `Q3 in [1,2,3,4], Q4 in [1,2,3,4]`
- `safe` adds `noattack` producing `ne` constraints
- `label` called for labeling
- `[Q1,Q2,Q3,Q4] = [2,4,1,3]`, `[Q1,Q2,Q3,Q4] = [3,1,4,2]`



Application – Send More Money (1)

$$\begin{array}{cccccc} & & S & E & N & D \\ + & & M & O & R & E \\ \hline = & M & O & N & E & Y \end{array}$$

Replace distinct letters by distinct digits,
numbers have no leading zeros.

Application – Send More Money (2)

```
:- use_module(library(clpfd)).

send([S,E,N,D,M,O,R,Y]) :-
     gen_domains([S,E,N,D,M,O,R,Y],0..9),
     S #\= 0, M #\= 0,
     all_distinct([S,E,N,D,M,O,R,Y]),
                  1000*S + 100*E + 10*N + D
     +            1000*M + 100*O + 10*R + E
     #= 10000*M + 1000*O + 100*N + 10*E + Y,
     labeling([],[S,E,N,D,M,O,R,Y]).
```

FD – Scheduling

- planning of temporal order of *tasks (jobs)* in presence of limited resources
  - task, e.g., production step or lecture
  - *resource* e.g., a machine, electrical energy, or lecture room
  - tasks compete for limited resources
  - dependencies between tasks
  - find a schedule with an optimal value for a given objective function (measuring time or use of other resources)
- *job shop scheduling problem*
  - tasks have fixed duration and cannot be interrupted
  - resources are machines for at most one task at a time
  - objective is to minimize the overall production time that is needed to accomplish all the tasks

FD Scheduling – Job Shop

- *task* T_i with $S_i + d_i = E_i$ $(0 \leq S_i, E_i \leq \texttt{maxtime})$
  - S_i: starting time
  - d_i: duration (known)
  - E_i: end time
- *precedence constraint* $S_i + d_i \leq S_j$:
  - task T_i must terminate before task T_j starts
  - partial order between tasks
- *capacity constraint* `ct(Si,di,Sj,dj)`
  - $S_i + d_i \leq S_j \ \vee \ S_j + d_j \leq S_i$:
    tasks T_i and T_j cannot be processed at the same time
  - exponential complexity if disjunction is implemented by search
  - often encoded by a special finite-domain constraint

FD Example – Job Shop

- query: `S1::1..6, S2::1..10, ct(S1,7,S2,6)`
  disjunction in `ct(S1,7,S2,6)` yields two cases:
  - `S1` is before `S2`: `S1::1..3, S2::8..10`
  - `S2` is before `S1`: inconsistent
- query: `S1::0..9, S2::4..9, S3::4..10, ct(S1,5,S2,5)`
  `ct(S1,5,S3,5), ct(S2,5,S3,5)`
  - only solution: `S1=0, S2=5, S3=10`

FD Scheduling – More things to cope with

- additional constraints to model
  - set-up times
  - release times
  - deadlines
  - renewable resources
  - non-availability
  - resources at certain times
- most can be modeled as arbitrary logical formulas over precedence constraints
- often implemented as global constraints

Modeling

How to model a problem?

- Several models for a problem
  - different variables
  - different constraints
- efficiency depends on
  - number of variables and constraints
  - type of constraints (algorithm)
- flexibility, maintainability

Modeling Example – Simple Allocation Problem

- 4 workers: w_1, w_2, w_3, w_4
- 4 products: p_1, p_2, p_3, p_4
- A worker is allocated a product and vice versa.

Profit of worker w_i with product p_j is given by

	p_1	p_2	p_3	p_4
w_1	7	1	3	4
w_2	8	2	5	1
w_3	4	3	7	2
w_4	3	1	6	3

Problem is solved when total profit is at least 19.

Modeling with Boolean Variables

- operations research-method: 16 Boolean variables B_{ij}
- $B_{ij} = 1$: worker i is allocated to product j

Model with linear arithmetic constraints:

- worker w_i is allocated to a single product:

$$\forall i : B_{i1} + B_{i2} + B_{i3} + B_{i4} = 1$$

- product p_j is allocated to a single worker:

$$\forall j : B_{1j} + B_{2j} + B_{3j} + B_{4j} = 1$$

Modeling with Boolean Variables (2)

- total profit:

$$\begin{aligned} P &= 7 * B_{11} + B_{12} + 3 * B_{13} + 4 * B_{14} \\ &\quad + 8 * B_{21} + 2 * B_{22} + 5 * B_{23} + B_{24} \\ &\quad + 4 * B_{31} + 3 * B_{32} + 7 * B_{33} + 2 * B_{34} \\ &\quad + 3 * B_{41} + B_{42} + 6 * B_{43} + 3 * B_{44} \\ P &\geq 19 \end{aligned}$$

- we compute all solutions (no optimization)

Modeling with Boolean Variables (3)

```
assignment(List) :-
      List = [B11,B12,B13,B14,B21,B22,B23,B24,
              B31,B32,B33,B34,B41,B42,B43,B44],
      gen_domains(List,0..1),
      B11 + B12 + B13 + B14 #= 1, B21 + B22 + B23 + B24 #= 1,
      B31 + B32 + B33 + B34 #= 1, B41 + B42 + B43 + B44 #= 1,
      B11 + B21 + B31 + B41 #= 1, B12 + B22 + B32 + B42 #= 1,
      B13 + B23 + B33 + B43 #= 1, B14 + B24 + B34 + B44 #= 1,
      P #= 7 * B11 + B12 + 3 * B13 + 4 * B14
         + 8 * B21 + 2 * B22 + 5 * B23 + B24
         + 4 * B31 + 3 * B32 + 7 * B33 + 2 * B34
         + 3 * B41 + B42 + 6 * B43 + 3 * B44,
      P #>= 19,
      labeling([],List).
```

Modeling with Boolean Variables (4)

```
:- assignment([B11,B12,B13,B14,B21,B22,...]).
B11 = 0,   B11 = 0,   B11 = 0,   B11 = 1
B12 = 0,   B12 = 0,   B12 = 1,   B12 = 0
B13 = 0,   B13 = 0,   B13 = 0,   B13 = 0
B14 = 1,   B14 = 1,   B14 = 0,   B14 = 0
B21 = 1,   B21 = 1,   B21 = 1,   B21 = 0
B22 = 0,   B22 = 0,   B22 = 0,   B22 = 1
B23 = 0,   B23 = 0,   B23 = 0,   B23 = 0
B24 = 0,   B24 = 0,   B24 = 0,   B24 = 0
B31 = 0,   B31 = 0,   B31 = 0,   B31 = 0
B32 = 0,   B32 = 1,   B32 = 0,   B32 = 0
B33 = 1,   B33 = 0,   B33 = 1,   B33 = 1
B34 = 0,   B34 = 0,   B34 = 0,   B34 = 0
B41 = 0,   B41 = 0,   B41 = 0,   B41 = 0
B42 = 1,   B42 = 0,   B42 = 0,   B42 = 0
B43 = 0,   B43 = 1,   B43 = 0,   B43 = 0
B44 = 0,   B44 = 0,   B44 = 1,   B44 = 1
```

28 search steps in total

Modeling with variables for the workers

4 variables: W_1, W_2, W_3, W_4

If worker i is allocated to product j, then W_i equals j.

Different value for each variable:

$W_i \neq W_j$ for all $1 \leq i \neq j \leq 4$

Modeling by the constraint `all_distinct`

```
all_distinct([W1,W2,W3,W4])
```

Modeling with variables for the workers (2)

- profit_i: profit of worker i for every product
- $\text{profit}_i[j]$: profit, if product j is done by worker i
- profit of worker i: $\text{profit}_i[W_i]$

Modeling by the array constraint `element`

`element(W1,[7,1,3,4],WP1)`

$(\text{profit}_1[1] = 7, \text{profit}_1[2] = 1, \text{profit}_1[3] = 3, \text{profit}_1[4] = 4)$

`W1 in [2,3], element(W1,[7,1,3,4],WP1)` $\Rightarrow$ `WP1 in [1,3]`

`WP1` ≥ 4, `element(W1,[7,1,3,4],WP1)` $\Rightarrow$ `W1 in [1,4]`

Modeling with variables for the workers (3)

`element(?X,+List,?Y)`

- `X`, `Y`: integers or domain variables
- `List`: list of integers or domain variables
- `element(?X,+List,?Y)`: True if the `X`-th element of `List` is `Y`.
- Operationally, the domains of `X` and `Y` are constrained s.t. for every element from the domain of `X`, there is a compatible element from the domain of `Y`, and vice versa.

Query: `Value #> 5, element(N,[7,8,4,3],Value).`

Answer: `Value in 7..8, N in 1..2`

Modeling with variables for the workers (4)

```
assignment(W1,W2,W3,W4) :-
        W1 in 1..4,
        W2 in 1..4,
        W3 in 1..4,
        W4 in 1..4,
        all_distinct([W1,W2,W3,W4]),
        element(W1,[7,1,3,4],WP1),
        element(W2,[8,2,5,1],WP2),
        element(W3,[4,3,7,2],WP3),
        element(W4,[3,1,6,3],WP4),
        P #= WP1 + WP2 + WP3 + WP4,
        P #>= 19,
        labeling([],[W1,W2,W3,W4]).
```

Modeling with variables for the workers (5)

```
:- assignment(W1,W2,W3,W4).
```

```
W1 = 1,    W1 = 2,    W1 = 4,    W1 = 4
W2 = 2,    W2 = 1,    W2 = 1,    W2 = 1
W3 = 3,    W3 = 3,    W3 = 2,    W3 = 3
W4 = 4,    W4 = 4,    W4 = 3,    W4 = 2
```

14 search steps in total

Modeling with variables for the products (1)

```
assignment(P1,P2,P3,P4) :-
        P1 in 1..4,
        P2 in 1..4,
        P3 in 1..4,
        P4 in 1..4,
        all_distinct([P1,P2,P3,P4]),
        element(P1,[7,8,4,3],TP1),
        element(P2,[1,2,3,1],TP2),
        element(P3,[3,5,7,6],TP3),
        element(P4,[4,1,2,3],TP4),
        P #= TP1 + TP2 + TP3 + TP4,
        P #>= 19,
        labeling([],[P1,P2,P3,P4]).
```

Modeling with variables for the products (2)

```
:- assignment(P1,P2,P3,P4).
```

```
P1 = 1,    P1 = 2,    P1 = 2,    P1 = 2
P2 = 2,    P2 = 1,    P2 = 3,    P2 = 4
P3 = 3,    P3 = 3,    P3 = 4,    P3 = 3
P4 = 4,    P4 = 4,    P4 = 1,    P4 = 1
```

7 search steps in total

Reason: Better propagation

```
TP1 in 3..8, TP2 in 1..3, TP3 in 3..7, TP4 in 1..4,
P #= TP1 + TP2 + TP3 + TP4, P #>= 19
```

$$TP_1 \geq \min(P) - \max(TP_2) - \max(TP_3) - \max(TP_4)$$

yields $TP_1 \geq 5$. The constraint `element(P1,[7,8,4,3],TP1)` then reduces the domain of `P1` to `1..2`.

Modeling Comparison

A difficult task

Some aspects

- efficiency
  - propagation of the constraints
  - number of variables (in general: the less the better)
- flexibility: additional constraints
  - depends on the type of the constraints

Modeling Comparison (2)

Flexibility

Ensure, that never worker 1 is allocated product 1 and worker 4 is allocated product 4.

- Boolean constraints: $B_{11} + B_{44} \leq 1$
- more difficult in the other models

Modeling Comparison (3)

Worker 3 is allocated a product with a number greater than the number of the product done by worker 2

- variables for the workers: $W_3 > W_2$
- more difficult with Boolean constraints
- even more difficult with variables for the products

Combination of different modelings

- provides redundant constraints for more propagation and flexibility
- requires bridge constraints to link variables from different models

Non-linear Equations I

Constraint System I

Domain

The set of real numbers.

Signature

- Function symbols:
  - The real numbers 0 and 1
  - Arithmetic function symbols $+$, $*$, log, sin, exp, ... as well as '..' for intervals.
- Constraint symbols:
  - Nullary symbols *true*, *false*
  - Binary symbols $=, <, \leq, >, \geq, \neq$ as well as *in* for domains

Constraint System I (2)

Constraint theory

An extension of the linear existential fragment of Tarski's axiomatic theory of real closed fields, including $X \; in \; n..m \leftrightarrow n \leq X \wedge X \leq m$

Allowed atomic constraints

Arithmetic equations and inequations:

$$C ::= true \;\; | \;\; false \;\; | \;\; X \; in \; n..m \;\; | \;\; X \odot Y \;\; | \;\; f(X_1, \ldots, X_l) = Z$$

- n, m: real numbers
- $\odot \in \{=, <, \leq, >, \geq, \neq\}$
- X, Y, Z and $X_1, \ldots, X_l$ ($l \geq 0$): pairwise distinct variables
- $f(X_1, \ldots, X_l)$: flat term, i.e., a function symbol from the signature applied to variables

Constraint Theory

- extends constraint theory for linear polynomials $\Re$
- undecidable if trigonometric functions are introduced
- periodicity of trigonometric functions can express the integer number property
- theory must include a model of Peano arithmetic, which is Presburger's arithmetic extended by multiplication
- Gödel: any consistent extension of Peano arithmetic is incomplete

I Approaches (1)

Variable elimination for non-linear polynomial equations

- e.g. Gröbner-Basis-Method (used in programming language CAL)
- no trigonometric functions
- double exponential time complexity

I Approaches (2)

Local consistency methods using interval arithmetics

- developed in AI (since 1960s)
- approximation method – can quickly be imprecise (incomplete)
- logarithmic and trigonometric functions can also be dealt with
- can approximate irrational numbers, e.g.
  $\sqrt{2}$ as `Sqrt2 in 1.41..1.42` or π as `Pi in 3.14..3.15`
- sophisticated extension of finite integer interval domains of *FD*
- polynomial time complexity

I Example – Interval Propagation

- $X^4 - 12X^3 + 47X^2 - 60X = 0$
  interval propagation yields the interval 0.0..5.0 for X (solutions are $0, 3, 4, 5$)
- $X^4 - 12X^3 + 47X^2 - 60X + 24 =_I 0$
  yields 0.8..1.0 (solutions are 0.888305... and 1)
- $X^4 - 12X^3 + 47X^2 - 60X + 24.1 =_I 0$
  no solution, but may yield a non-empty interval

We do not know how many solutions an interval contains.

Boundaries of an interval need not be a solution – due to rounding and irrational numbers

Notion of bounds consistency of FD has to be adopted.

I – Local-Propagation Constraint Solver

- rules of FD-solver modified to work for intervals of reals
- non-trivial real-number intervals admit infinitely many values
- granularity:
  - limits the precision, stop if interval is small enough
  - only heuristics for the size of the smallest useful interval
  - rounding errors in arithmetic computations avoided by rounding bounds of interval outward
- multiplication, exponentiation, logarithmic and trigonometric functions
  - not monotonic anymore, e.g. sin-function
  - interval propagation is difficult to implement and not very effective

I – Interval Domains

```
intersect @  X in A..B, X in C..D <=> X in max(A,C)..min(B,D).
empty @  X in Min..Max          <=> Min>Max | fail.

le @ X le Y, X in A..B, Y in C..D ==> Y in A..D, X in A..D.
eq @ X eq Y, X in A..B, Y in C..D ==> Y in A..B, X in C..D.
ne @ X ne Y, X in A..A, Y in A..A <=> fail.

add @ add(X,Y,Z), X in A..B, Y in C..D, Z in E..F ==>
          X in E-D..F-C, Y in E-B..F-A, Z in A+C..B+D.
```

- Smallest interval guards not implemented.
- Outward rounding not implemented.
- `ne` rule applies, problem also with (strict) `<`, since successor function does not exist for reals.

I – Interval Domains (2)

- variables (never) determined
  - stopping at small intervals
  - outward rounding
  - infinitely many possible values in domains

I – Multiplication

`mult(X,Y,Z)` means $X{*}Y{=}Z$

`has_zero(A..B)` iff $\texttt{A} \leq 0 \wedge 0 \leq \texttt{B}$

```
mult_z @ mult(X,Y,Z), X in A..B, Y in C..D ==>
         M1 is A*C, M2 is A*D, M3 is B*C, M4 is B*D,
         Z in min(M1,M2,M3,M4)..max(M1,M2,M3,M4).

mult_y @ mult(X,Y,Z), X in A..B, Z in E..F ==>
         not has_zero(A..B) |
         M1 is E/A, M2 is E/B, M3 is F/A, M4 is F/B,
         Y in min(M1,M2,M3,M4)..max(M1,M2,M3,M4).

mult_x @ mult(Y,X,Z), X in A..B, Z in E..F ==> ...
```

I – Multiplication (2)

Examples:

- `A in 0..0.3, B in 0..0.3, C in 0..0.3, A eq B, B eq C,`
  `mult(A,B,C)`
  yields `A in 0.0..1.0e-07, B in 0.0..1.0e-07, C in 0.0..1.0e-07`
  (assuming size of smallest intervals is 1.0e-07)
- `A in -1..1, B in -1..1, C in -1..1, B eq C,`
  `mult(A,B,C)`
  - propagation has no effect
  - solutions `A=1` or `B=C=0` cannot make intervals smaller

I – Multiplication (3)

More propagation: Take care of non-monotonicity around 0. Split intervals in positive and negative part.

*	−	0	+
−	+	0	−
0	0	0	0
+	−	0	+

- If possible remove complete positive or negative sub-interval.
- `mult(X,Y,Z), X in -2..3, Y in -3..4, Z in 7..12` does not propagate, but should simplify to `mult(X,Y,Z), X in 1.75..3, Y in 2.33..4, Z in 7..12`, since `-2*-3` is only `6`.

I – Multiplication (4)

```
mult_xyz @ mult(X,Y,Z), X in A..B, Y in C..D, Z in E..F ==>
        has_zero(A..B), has_zero(C..D), not has_zero(E..F) |
        mult0(X,Y,Z).

   mult0(X,Y,Z), X in A..B, Y in C..D, Z in E..F ==>
         A*C<E | D>0, X in E/D..B.
   mult0(X,Y,Z), X in A..B, Y in C..D, Z in E..F ==>
         B*D<E | C<0, X in A..E/C.
   mult0(X,Y,Z), X in A..B, Y in C..D, Z in E..F ==>
         F<A*D | C<0, X in F/C..B.
   mult0(X,Y,Z), X in A..B, Y in C..D, Z in E..F ==>
         F<B*C | D>0, X in A..F/D.
```

I – Local-Propagation Constraint Solver (2)

- *terminates* (ref. *FD*-solver):
  intervals get smaller with each rule application (even with outward rounding) and too small intervals not considered by rules
- *not confluent*:
  order of rule applications, resulting intervals may be different due to stopping at small intervals and accumulated roundings
- $O(cvw)$ complexity (ref. *FD*-solver):
  width (size) w of $n..m.$: number of floating point numbers or smalles intervals between n and m

I – Local-Propagation Constraint Solver (3)

- *search*: try to isolate solutions
  - solver remains incomplete (variables never determined)
  - *domain splitting*: divide intervals in halves until they are too small
  - *probing*: try out a smallest interval taken from the given one
  - *shaving*: remove intervals around interval bounds if probing determines unsatisfiability

I – Improving and Extending Interval Propagation

Improve by:

- non-flat or global constraints (but no tractable method to determine best representation)
- convergence acceleration

Combine with:

- variable elimination (Groebner Bases)
- Newton's approximation
- Taylor expansion

I – Applications

- physics, chemistry, mathematics (geometry)
- physical, chemical, and molecular-biological modeling and simulation of hybrid systems
- hybrid circuits, spatial reasoning, robot control, equilibrium of chemicals, financial analysis