Table of Contents

My first CHR programs

CHR as a database language
Multiset transformation
Procedural algorithms
Graph-based algorithms
Overview

- Introduction of simple, yet concise and effective CHR programs
- Informal discussion of basic properties of CHR programs
  - Anytime and online algorithm property
  - Logical correctness
  - Rule confluence
  - Declarative Concurrency
  - Worst-case time complexity

Decidable operational equivalence
Symbolic execution (already at compile-time if confluent)
Every algorithm can be implemented in optimal time and space complexity
CHR as database language

- CHR can be used as information store
- CHR as deductive database
  - Relations modeled as CHR constraints
  - Database tuple is instance of constraint
  - Query contains/generates tuples of database as CHR constraints
  - Queries, views, integrity constraints formulated as CHR propagation rules
    - New data constraints (i.e. database tuples) can be deducted
Examples (I)

Example (Family relationships)

mother(X,Y) ==> parent(X,Y).
father(X,Y) ==> parent(X,Y).
parent(X,Z), parent(Y,Z) ==> sibling(X,Y).

Given
mother(hans,mira), mother(sepp,mira), father(sepp,john)

First two rules derive
parent(hans,mira), parent(sepp,mira), parent(sepp,john)

Last rule adds
sibling(hans,sepp), sibling(sepp,hans)

Avoiding second sibling constraint by adding rule
sibling(X,Y) sibling(Y,X) <=> true.
Example (Family relationships continued)

mother(X, Y) ==> parent(X, Y).
father(X, Y) ==> parent(X, Y).
parent(X, Z), parent(Y, Z) ==> sibling(X, Y).

- Considering also grandparents, great-grandparents and so on using ancestor relation
  
  parent(X, Y) ==> ancestor(X, Y).
  parent(X, Y), ancestor(Y, Z) ==> ancestor(X, Z).

- First rule: “Parent is ancestor”
- Second rule: “Ancestor of parent is ancestor”
- Ancestor relation is transitive closure of parent relation
Example (Crossword)

- Words represented by CHR constraints
  \[\text{word}(n,o), \text{word}(d,o,g), \text{word}(b,o,o,k)\]
- Crossword problem represented as sequence of \text{word} constraints with variables as arguments
  - One variable corresponds to one field
  - Same variable for fields shared by two words
- Sequence of words as head of propagation rule
  \[\text{word}(A,B,C,D), \text{word}(E,F,G), \text{word}(A,E,H)\ldots \implies \text{solution}(A,B,C,D,E,F,G,H)\]
- \text{solution} is auxiliary CHR constraint for output
Multiset transformation

- Programs consisting of essentially one constraint
- Constraint represents active data
- Pairs of constraints rewritten by single simplification rule
- Often possible: more compact notation with simpagation rule
- Simpagation rule removes one constraint, keeps (and updates) other
Minimum I

Minimum program

\[ \text{min}(N) \ \text{\lor} \ \text{min}(M) \iff N<M \ \text{\lor} \ \text{true}. \]

- Computing minimum of multiset of numbers \( n_i \)
- Numbers given as query \( \text{min}(n_1), \text{min}(n_2), \ldots, \text{min}(n_k) \)
- \( \text{min}(n_i) \) means \( n_i \) is potential minimum
- Simpagation rule takes two \( \text{min} \) constraints and removes the one representing the larger value.
- Program continues until only one \( \text{min} \) constraint left
- This \( \text{min} \) constraint represents smallest value
Minimum II

Minimum program

\[
\min(N) \ \text{and} \ \min(M) \iff N\leq M \cup \text{true}.
\]

- Rule corresponds to intuitive algorithm:
  “Cross out larger numbers until one, the minimum remains”

- Illustrates use of multi-headed rule to iterate over data
  - No explicit loops or recursion needed
  - Keeps program code compact
  - Makes program easier to analyze
Program properties (II)

- Program is **terminating** because rule only removes constraints without adding new ones.
- Number of rule applications is one less than number of \( \text{min} \) constraints.
- Rule can be applied in constant time.
- Given two \( \text{min} \) constraints rule can always be applied (in one order or the other).
  - \( \Rightarrow \) **Complexity** linear in number of \( \text{min} \) constraints
    (invariant: at most one active and at most one passive constraint)
Example computations

Computation using top-down rule application and left-to right goal processing order

Example computation
\[
\text{min}(1), \text{min}(0), \text{min}(2), \text{min}(1) \\
\text{min}(0), \text{min}(2), \text{min}(1) \\
\text{min}(0), \text{min}(1) \\
\text{min}(0) \\
\]

Computation using different order

Example computation
\[
\text{min}(1), \text{min}(0), \text{min}(2), \text{min}(1) \\
\text{min}(1), \text{min}(0), \text{min}(1) \\
\text{min}(0), \text{min}(1) \\
\text{min}(0) \\
\]
Program properties (I)

- Both example computations lead to same answer for the given query.
- In general: answer (for given query) does not depend on order of rule applications.
- This property is called **confluence**.
- Rule can be applied in parallel without changing the program.

**Example computation (parallel)**

```
min(1), min(0), min(2), min(1)
min(0), min(1)
```

- This property is called **logical parallelism** or **declarative concurrency** (Ultraparallelism: constant time complexity).
Program properties (III)

- Program can be stopped at any time and intermediate answer (current store) can be observed.
- Computation can be continued after that without restarting from scratch.
- Intermediate results become closer and closer to final answer (fewer and fewer \( \min \) constraints).
- Intermediate answers approximate final answer.
- Called **anytime algorithm property**.
- Makes algorithm also an **approximation algorithm**.
Program properties (IV)

- Assuming $\min$ constraint is added during computation
- Will eventually participate in computation
- Same situation as if added constraint was there from beginning but has been ignored for some time
  $\Rightarrow$ Result will still be correct
- This property is called online algorithm property or incrementality
Abstract semantics (I)

- So far \( \min \) constraints contain given values
  - Guard acts as a test comparing values
- In general (abstract operational semantics):
  Guard holds if it is logically implied by current store
- In practical implementations:
  Error or silent fail if unbound variables occur in guard to check
- Guard check may still succeed under abstract semantics
Abstract semantics (II)

Example

- Query $\text{min}(A), \text{min}(B), A=B$ reduces to $\text{min}(A), A=B$
  - Guard asks for $A=B$ and $A=B$ is known
- Query $\text{min}(A), \text{min}(B), A<B$ reduces to $\text{min}(A), A<B$
  - Same argumentation
- Query $\text{min}(A), \text{min}(A)$ reduces to $\text{min}(A)$
- Query $\text{min}(A), \text{min}(B)$ will not proceed
  - Relationship of $A$ and $B$ not know
Operational equivalence

Different minimum program

\[
\begin{align*}
\min(N) \setminus \min(M) &\iff N < M \mid \text{true.} \\
\min(N) \setminus \min(M) &\iff N = M \mid \text{true.}
\end{align*}
\]

- First rule does not remove duplicates of final minimum constraint
- Second rule only removes duplicates
- Both rules together have same behavior as original program (when working with known values)
- Query \( \min(A), \min(B), A=\leq B \) will not reduce in new program under abstract semantics (\( \leq \) does not imply = neither <)
- Programs are not operationally equivalent but logically equivalent
Program variations (I)

- Compute minimum from values which should not be removed
  - Add propagation rule generating  \( \text{min} \) constraint for each value to consider e.g.
    \[
    \text{c}(\ldots, X, \ldots) \implies \text{min}(X).
    \]

- Trigger computation at certain point
  - Add auxiliary dummy constraint which triggers computation e.g.
    \[
    \text{findmin}, \min(N) \setminus \min(M) \implies N=\leq M \lor \text{true}.
    \]
  - Can also be used to return computed minimum via \( \text{ismin} \) by adding the rule
    \[
    \text{findmin}, \min(N) \implies \text{ismin}(N).
    \]
  - Minimum can also be returned in \( \text{findmin} \) with rule
    \[
    \text{findmin}(\text{Min}), \min(N) \implies \text{Min}=N.
    \]
Program variations (II)

- Computing several minima from different sources
  - Add identifier to \( \text{min} \) constraint and change rule to
    \[
    \text{min}(\text{Id},N) \setminus \text{min}(\text{Id},M) \leftrightarrow N<M \mid \text{true}.
    \]
  - Rule only applies to \( \text{min} \) constraints with same identifier
  - Technique can be used to implement local constraint stores
Boolean Exclusive Or (I)

- Multiset of $\text{xor}$ constraints denote input (similar to minimum)
- Result is single remaining $\text{xor}$ constraint
- Four rules according to truth table
  
  $\text{xor}(0), \text{xor}(0) \iff \text{xor}(0)$.
  $\text{xor}(0), \text{xor}(1) \iff \text{xor}(1)$.
  $\text{xor}(1), \text{xor}(0) \iff \text{xor}(1)$.
  $\text{xor}(1), \text{xor}(1) \iff \text{xor}(0)$.

- Second and third rule redundant (differ only in order of head constraints)
- First and forth rule can be generalized into one rule
Boolean Exclusive Or (II)

### XOR program

\[
\text{xor}(X), \quad \text{xor}(X) \iff \text{xor}(0). \\
\text{xor}(1) \setminus \text{xor}(0) \iff \text{true}.
\]

- First rule: “Replace identical constraints by \(\text{xor}(0)\)”
- Second rule: “Remove \(\text{xor}(0)\) constraints if an \(\text{xor}(1)\) constraint is present”
- Confluence of program can be shown
- Example for confluence with query \(\text{xor}(1), \text{xor}(1), \text{xor}(0)\)
  - Apply first rule twice or second rule followed by first rule
  - Result is always \(\text{xor}(0)\)
Boolean Exclusive Or (III)

- Applying first rule until exhaustion to any query results in either \( \text{xor}(1) \) or \( \text{xor}(0) \) or \( \text{xor}(1), \text{xor}(0) \)
  - Second rule has to be applied at most once in this case

- Simplifying rule \( \text{xor}(1) \ \text{\textbackslash} \ \text{xor}(0) \iff \text{true} \) to \( \text{xor}(0) \iff \text{true} \) would be incorrect
  - Query \( \text{xor}(0), \text{xor}(0) \) would result in wrong answer \( \text{true} \)

- Generalization to \( \text{xor}(X) \ \text{\textbackslash} \ \text{xor}(0) \iff \text{true} \) is possible
  - \( \text{xor}(X), \text{xor}(X), \text{xor}(X) \) reduces to \( \text{ xor}(X) \)
  - With original rule the result would be \( \text{ xor}(0), \text{ xor}(X) \)
Greatest common divisor (I)

**GCD program**

\[ \text{gcd}(N) \setminus \text{gcd}(M) \iff 0 < N, N = M \mid \text{gcd}(M-N). \]

- Computes greatest common divisor of natural number represented as \( \text{gcd}(N) \)
- Result is remaining nonzero \( \text{gcd} \) constraint

**Example computation**

\[ \text{gcd}(12) \setminus \text{gcd}(8) \]
\[ \text{gcd}(8) \setminus \text{gcd}(4) \]
\[ \text{gcd}(4) \setminus \text{gcd}(4) \]
\[ \text{gcd}(4) \setminus \text{gcd}(0) \]
Greatest common divisor (II)

- **Condition** $0 < N$ in guard leads to ignoring $\text{gcd}(0)$ constraints
- **Usability** can be improved by adding $\text{gcd}(0) \iff \text{true}$
- **Efficiency** can be improved by replacing subtraction with modulo operation
  
  Complexity from linear to logarithmic

$\text{gcd}(N) \setminus \text{gcd}(M) \iff 0 < N, N =\leq M \mid \text{gcd}(M \mod N)$.

**Example computation**

\[
\begin{align*}
\text{gcd}(7), & \quad \text{gcd}(12) \\
\text{gcd}(7), & \quad \text{gcd}(5) \\
\text{gcd}(5), & \quad \text{gcd}(2) \\
\text{gcd}(2), & \quad \text{gcd}(1) \\
\text{gcd}(1), & \quad \text{gcd}(0) \\
\text{gcd}(1)
\end{align*}
\]
Greatest common divisor (III)

- Program also works for several $\text{gcd}$ constraints
  - Query $\text{gcd}(94017), \text{gcd}(1155), \text{gcd}(2035)$ results in $\text{gcd}(11)$ Since confluent and thus incremental
- Code also works for rational numbers (given according arithmetic operations)
- Termination is ensured for natural numbers
  - New value always smaller than $M$
  - New value cannot be negative due to guard
- Program is confluent when used with known numbers

Correctness: Set of common divisors invariant under rule application
Parallelism gives ultralinear speed-up
Binary gcd (I)

- gcd of odd and even number cannot be even
  ⇒ divide even number by 2 until it becomes odd
- Add new rule as first rule
  \[ \text{gcd}(M) \setminus \text{gcd}(N) \iff \text{odd}(M), \text{even}(N) \mid \text{gcd}(N//2). \]
- \( \text{gcd}(M) \) ensures that there is at least one odd number
- If odd number present logarithmic complexity even in subtraction version
  - New rule immediately applicable after subtraction
  - Difference of two odd numbers always even

Generalisation: Represent numbers as product of odd number and power of two
Prime sieve

Prime sieve (I)
sift @ prime(I) \ prime(J) <= J mod I =:= 0 | true.

- Rule removes multiples of each of the numbers
- Query: Prime number candidates from 2 to up to \( N \)
i.e. \( \text{prime}(2), \text{prime}(3), \text{prime}(4), \ldots \text{prime}(N) \)
- Each number absorbs multiples of itself, eventually only prime numbers remain

Example computation

- \( \text{prime}(7), \text{prime}(6), \text{prime}(5), \text{prime}(4), \text{prime}(3), \text{prime}(2) \)
- \( \text{prime}(7), \text{prime}(5), \text{prime}(4), \text{prime}(3), \text{prime}(2) \)
- \( \text{prime}(7), \text{prime}(5), \text{prime}(3), \text{prime}(2) \)
Prime sieve (II)

- Rule can be seen as specialization of modulo version of gcd
  - Result of modulo operation required to be zero
- Also similar to minimum rule
  - Two numbers compared, one removed
  - But not applicable to arbitrary pairs
- Program is terminating (only removes constraints without introducing new ones)
- Anytime and online algorithm property also given
  (Ultraparallelism: constant time complexity)
Generating numbers

- **Auxiliary constraint** `upto` to generate prime candidates
  
  `upto(1) <=> true.`
  
  `upto(N) <=> N>1 | prime(N), upto(N-1).`

- **Prime constraint** itself can be used as well

  `prime(N) ==> N>2 | prime(N-1).`

- **Both versions generate constraints in descending order**

- **Ascending order preferably for efficiency**
  
  - Smaller candidates increase probability that rule is applicable

- **`upto` rule can be fixed:**

  `upto(N) <=> N>1 | upto(N-1), prime(N).`
Exchange sort (I)

Exchange sort program

\[ a(I, V), a(J, W) \iff I > J, V < W \mid a(I, W), a(J, V). \]

- Rule sorts array by exchanging values which are in wrong order
- Array is sequence of constraints \( a(\text{Index}, \text{Value}) \)
  i.e. \( a(1, A1), \ldots, a(n, An) \)

Example computation

\[
\begin{align*}
\text{a (0, 1), a (1, 7), a (2, 5), a (3, 9), a (4, 2)} \\
\text{a (0, 1), a (1, 5), a (2, 7), a (3, 2), a (4, 9)} \\
\text{a (0, 1), a (1, 5), a (2, 2), a (3, 7), a (4, 9)} \\
\text{a (0, 1), a (1, 2), a (2, 5), a (3, 7), a (4, 9)}
\end{align*}
\]
Exchange sort (II)

- In sorted array for each pair \( a(I, V), \ a(J, W) \) with \( I > J \) it holds that \( V \geq W \)
- Rule ensures this by exchanging values if necessary
  \( \Rightarrow \) array sorted if rule not applicable anymore
- Every rule application corrects at least one ordering without introducing wrong orderings
  \( \Rightarrow \) program terminates
- Program is confluent for queries with known numbers, but not in general
Newton’s method for square roots (I)

**Square root program**

\[
\text{sqrt} (X, G) \iff \text{abs} (G \times G / X - 1) > 0 \lor \text{sqrt} (X, (G + X / G) / 2).
\]

- \text{sqrt} (X, G) means that the square root of X is approximated by G
- Rule is based on formula \( G_{i+1} = (G_i + X/G_i)/2 \)
- Straightforward implementation because CHR programs already anytime, i.e. approximation algorithms
- Rule replaces \text{sqrt} constraint with one containing next approximation
- Query is \text{sqrt}(\text{GivenNumber}, \text{Guess}) (both numbers positive)
Newton’s method for square roots (II)

Square root program

\[ \text{sqrt}(X, G) \leftrightarrow \text{abs}(G \times G/X - 1) > 0 \mid \text{sqrt}(X, (G + X/G)/2). \]

- Guard stops rule application if approximation is exact
- Unlikely in praxis, hence 0 should be replaced by small \( \varepsilon \)
- Demand driven version:

\[ \text{improve} (\text{sqrt}(X)), \text{sqrt}(X, G) \leftrightarrow \text{sqrt}(X, (G + X/G)/2). \]

- Approximation step only performed on demand (expressed by \text{improve})
- Can be extended with counter or quality check
Procedural algorithms

- More traditional style of programming used in this section
- Constraints as relations that resemble procedures
- Results returned as values bound to variables
- Fibonacci example will show CHR’s support for different programming styles
Maximum

Maximum program

\[
\text{max}(X, Y, Z) \iff X=Y \quad \text{or} \quad Z=Y.
\]

\[
\text{max}(X, Y, Z) \iff Y=X \quad \text{or} \quad Z=X.
\]

- \( \text{max}(X, Y, Z) \) means \( Z \) is the maximum of \( X \) and \( Y \)
- \textbf{Goal} \( \text{max}(1, 2, M) \) reduces to \( M=2 \) with first rule
- \textbf{Goal} \( \text{max}(1, 2, 3) \) fails because of inconsistent built-in \( 3=2 \)
- \textbf{To the goal} \( \text{max}(1, 1, M) \) both rules are applicable and result in \( M=1 \) (indeed program is confluent)
- Program terminating because body only contains built-ins
Fibonacci numbers

- n-th Fibonacci number defined recursively by

\[ fib(0) = fib(1) = 1; \quad fib(n) = fib(n-1) + fib(n-2) \text{ for } n \geq 2 \]

- Implement by translating functional notation into relational one and equivalence into simplification rule

- General approach to this is called flattening
Top-down evaluation

Fibonacci top-down

\[
\begin{align*}
& f_0 \ @ \ fib(0, M) \iff M = 1. \\
& f_1 \ @ \ fib(1, M) \iff M = 1. \\
& f_n \ @ \ fib(N, M) \iff N \geq 2 \land fib(N-1, M_1), \ fib(N-2, M_2), \\
& \quad M \ is \ M_1 + M_2.
\end{align*}
\]

- \( fib(N, M) \) holds if \( M \) is \( N \)th Fibonacci number
- Recursive approach starting with highest Fibonacci number
- Also called goal-driven or backward-chaining approach
- Examples: \( fib(8, A) \) yields \( A = 34 \), \( fib(12, 233) \) succeeds, \( fib(11, 233) \) fails, and \( fib(N, 233) \) delays
- Problem: exponential complexity
Tabulation and memorization (I)

- Store and look up already computed Fibonacci numbers
- Easy implementation because CHR constraints are both data and operations
  - Turn simplification into propagation rules
  - This will keep constraints in store as data
- Rule for look-up of already computed numbers has to come first

Fibonacci with memorization

\[
\text{mem @ fib}(N,M1) \ \backslash \ fib(N,M2) \iff M1=M2.
\]

\[
\text{f0 @ fib}(0,M) \implies M=1.
\]

\[
\text{f1 @ fib}(1,M) \implies M=1.
\]

\[
\text{fn @ fib}(N,M) \implies N\geq2 \ | \ fib(N-1,M1), \ fib(N-2,M2),
\]
\[
M \text{ is } M1+M2.
\]
Tabulation and memorization (II)

- **mem** enforces functional dependency of input and output of Fibonacci relation
- Query `fib(8, A)` now returns all Fibonacci numbers up to the eighth
- Only linear complexity
  - Each Fibonacci number only computed once
  - Recursive call is only a look-up
- **mem** rule also merges two computations which have the same result
Bottom-up evaluation

- Use only data, compute larger Fibonacci numbers from smaller ones
- Starting from given fact and proceeding towards solution (also called forward-chaining or data-driven)
- Reverse head and body of top-down rules

```prolog
fn @ fib(N1,M1), fib(N2,M2) ==> 
    N2:=N1+1 | fib(N2+1,M1+M2).
```
- Include `fib(0,1), fib(1,1)` in query instead of first two rules
- Computation infinity, add rule to observe result

```prolog
fib(N,M) ==> write(fib(N,M)).
```
Termination

- Make computation terminating by introducing \(\text{fib\_upto}(\text{Max})\)
- Also used to introduce first two Fibonacci numbers

Fibonacci bottom-up and terminating

\[
\begin{align*}
\text{f01} @ \text{fib\_upto}(\text{Max}) & \implies \text{fib}(0,1), \text{fib}(1,1) \\
\text{fn} @ \text{fib\_upto}(\text{Max}), \text{fib}(\text{N1},\text{M1}), \text{fib}(\text{N2},\text{M2}) & \implies \\
& \text{Max}>\text{N2}, \text{N2}=:\text{N1}+1 | \text{fib}(\text{N2}+1,\text{M1}+\text{M2}).
\end{align*}
\]
Faster version

- Even faster version: Turn propagation rule into simpagation rule
  \[
  \text{fn @ fib(Max), fib(N2,M2) \ \fib(N1,M1) <=> Max>N2, N2:=N1+1 | fib(N2+1,M1+M2).}
  \]

- Only keeps the last two Fibonacci constraints

- Exchanged order in rule head to remove smaller Fibonacci number

  Fibonacci is related to GCD backwards
Procedural style version

- Three head constraints of \( fn \) rule can be merged into one
- Three constraints resulting from rule application can also be merged

Fibonacci procedural style

\[
\begin{align*}
0 & @ \text{fib}(\text{Max}) \iff \text{fib}(\text{Max},1,1,1). \\
\text{fn} & @ \text{fib}(\text{Max},N,M1,M2) \iff \\
& \quad \text{Max} > N \mid \text{fib}(\text{Max},N+1,M2,M1+M2).
\end{align*}
\]
Depth first search (I)

- Binary tree with ordered data: Every node in left subtree is smaller, in right subtree is bigger than parent node
- Binary tree represented by nested term
  \[ \text{node}(\text{Data}, \text{Lefttree}, \text{Righttree}) \]
- **Operation constraint** \( \text{dfsearch}(\text{Tree}, \text{Data}) \) searches tree for given data
- Illustrates recursive descent in CHR

Depth first search program

```prolog
empty @ dfsearch(nil, X) <=> false.
found @ dfsearch(node(N, L, R), X) <=> X=N | true.
left @ dfsearch(node(N, L, R), X) <=> X<N | dfsearch(L, X).
right @ dfsearch(node(N, L, R), X) <=> X>N | dfsearch(R, X).
```
Depth first search (II)

Depth first search program with identifier

```chr
empty @ nil(I) \ dfsearch(I,X) <=> fail.
found @ node(I,N,L,R) \ dfsearch(I,X) <=> X=N | true.
left @ node(I,N,L,R) \ dfsearch(I,X) <=> X<N | dfsearch(L,X).
right @ node(I,N,L,R) \ dfsearch(I,X) <=> X>N | dfsearch(R,X).
```

- Different granularity of data
- Node represented by CHR constraint
- Tree represented as sequence of nodes
- Unique unifier added to each node and `dfsearch`
Direct access

\[
\begin{align*}
\text{found} @ \text{node}(N) \setminus \text{search}(N) & \equiv true. \\
\text{empty} @ \text{search}(N) & \equiv \text{fail}. 
\end{align*}
\]

- Data can be accessed directly by mentioning in head constraint
- Use only unary CHR constraint \text{node}(N)
- Illustrates CHR’s support for topological views of structured data
- Components can be accessed directly
Destructive assignment (I)

- Update or override of bound variables not possible in declarative programming languages
- Simulation of destructive assignment possible in CHR
  - Remove constraint with old value and add on with new value
- Optimizing compiler can translate this into in-place operation
  ⇒ Constant time for simulating destructive assignment
- This is not known for other purely declarative languages
- One source of CHR’s efficiency
Destructive assignment (II)

Destructive assignment

\[
\text{assign}(\text{Var}, \text{New}), \ \text{cell}(\text{Var}, \text{Old}) \iff \text{cell}(\text{Var}, \text{New}).
\]

- Storing name-value pairs as constraint `cell`
- Use `assign` constraint to update values
- Variables introduced by a `cell` constraint
- New cell constraint may trigger new computations (data-driven)
- Order matters for destructive assignments
  - Program containing such a rule is not confluent
  - Cannot be run in parallel without modification
- Standard first-order declarative semantics does not reflect intended meaning
Graph-based algorithms

- Algorithms working on generic class of relations: graphs
- Graph is binary relation over nodes
- Programs in this section deal with
  - Transitive closure
  - Shortest paths
  - Partial order constraints
  - Grammar parsing
  - Ordered merging and sorting
Transitive closure

- Transitive closure $R^+$ of binary relation $R$ is smallest transitive relation containing $R$
- $xR^+y$ holds iff finite sequence of $x_i$ exist with $xR_1, x_1Rx_2, \ldots, x_nRy$
- Examples
  - $R$ parent relation, $R^+$ ancestor relation
  - $R$ cities connected by direct trains, $R^+$ cities reachable by train
- $R$ can be seen as directed graph with directed edge from node $x$ to node $y$ iff $xRy$
- $R^+$ corresponds to all paths in this graph
Implementation (I)

Transitive closure

\begin{align*}
\text{pl} @ e(X, Y) & \implies p(X, Y). \\
\text{pn} @ e(X, Y), p(Y, Z) & \implies p(X, Z).
\end{align*}

- $R$ implemented as edge constraint, $R^+$ as path constraint
- Propagation rules compute transitive closure bottom-up
- First rule adds path for every edge
- Second rule extends existing path by adding an edge in front
Implementation (II)

Transitive closure

\[\begin{align*}
    pl & @ e(X, Y) \implies p(X, Y). \\
pn & @ e(X, Y), p(Y, Z) \implies p(X, Z).
\end{align*}\]

- **Query** \( e(1, 2), e(2, 3), e(2, 4) \) adds path constraints
  \[ p(1, 4), p(2, 4), p(1, 3), p(2, 3), p(1, 2) \]

- **Query** \( e(1, 2), e(2, 3), e(1, 3) \) computes \( p(1, 3) \) twice
  (two ways to get to node 3 from node 1)

- When distinction of edges and paths dropped program can be simplified to
  \[ p(X, Y), p(Y, Z) \implies p(X, Z). \]
Termination

- Program does not terminate for cyclic graphs
- For query $e(1, 1)$ infinitely many $p(1, 1)$ generated by $p_n$
- Various compiler optimizations and options to avoid repeated generation of same constraint exists
- But program is not terminating for any implementation
Duplicate removal

- Restoring termination by removing duplicate constraints before they are used
- Enforcing set-based semantics for path constraint
- Ensures termination (finite graph contains only finite number of paths)
- Duplicate removal rule (has to come first in program)
  \[ dp @ p(X, Y) \setminus p(X, Y) \iff \text{true}. \]
- Implementations try to remove new constraint and keep old one
Candidates (I)

Transitive closure using candidates

\[
\begin{align*}
n(X) & \implies cp(X, X). \\
n(X), n(Y) & \implies cp(X, Y). \\
e(X, Y) \setminus cp(X, Y) & \iff p(X, Y). \\
e(X, Y), p(Y, Z) \setminus cp(X, Z) & \iff p(X, Z). 
\end{align*}
\]

- Direct and declarative way not relying on rule order
- Given nodes as unary constraints \( n \) compute all candidates for paths and use simpagation rules
- First rule necessary (head constraints in second rule can only match different constraints)
- Remaining \( cp \) constraints are those for which no path exist
Candidates (II)

Transitive closure using candidates

\[
\begin{align*}
n(X) & \implies cp(X, X). \\
n(X), n(Y) & \implies cp(X, Y). \\
e(X, Y) \setminus cp(X, Y) & \iff p(X, Y). \\
e(X, Y), p(Y, Z) \setminus cp(X, Z) & \iff p(X, Z).
\end{align*}
\]

- **Query:** \( n(1), n(2), n(3), e(1, 2), e(2, 3) \)
- **Paths added:** \( p(1, 2), p(2, 3), p(1, 3) \)
- \( cp \) constraint added between all other six pairs of nodes
- **When adding** \( e(3, 1) \) **to query graph becomes cyclic**
- **All pairs become paths, no** \( cp \) **constraints left**
 Reachability: single-source and single-target paths (I)

- Specialize transitive closure so that only paths which reach single target are computed

  
  \[
  \text{target}(Y), \ e(X,Y) \implies p(X,Y).
  \]
  
  \[
  \text{target}(Z), \ e(X,Y), \ p(Y,Z) \implies p(X,Z).
  \]

- (Almost) analogous for source

  
  \[
  \text{source}(X), \ e(X,Y) \implies p(X,Y).
  \]
  
  \[
  \text{source}(X), \ p(X,Y), \ e(Y,Z) \implies p(X,Z).
  \]

- Can be simplified (any path produced has same first argument)

  
  \[
  p(X) \setminus p(X) \iff \text{true}.
  \]
  
  \[
  \text{source}(X), \ e(X,Y) \implies p(Y).
  \]
  
  \[
  p(Y), \ e(Y,Z) \implies p(Z).
  \]

- If source replaced with p, second rule no longer needed
Shortest path (I)

Shortest path

\[ p(X, Y, N) \ \backslash \ p(X, Y, M) \iff N \leq M \ | \ true. \]
\[ e(X, Y) \implies p(X, Y, 1). \]
\[ e(X, Y), \ p(Y, Z, N) \implies p(X, Z, N+1). \]

- Adding argument which stores length of path
- Keeping shorter path in duplicate removal (ensures termination)
- Path propagated from edge has length 1
- Path of length \( n \) extended by edge has length \( n + 1 \)
## Shortest path (II)

<table>
<thead>
<tr>
<th><strong>Shortest path</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>$p(X, Y, N) \setminus p(X, Y, M) \iff N \leq M \mid true.$</td>
</tr>
<tr>
<td>$e(X, Y) \implies p(X, Y, 1).$</td>
</tr>
<tr>
<td>$e(X, Y), p(Y, Z, N) \implies p(X, Z, N+1).$</td>
</tr>
</tbody>
</table>

- **Query** $e(X, X)$ reduces to $p(X, X, 1)$
- **For query** $e(X, Y), e(Y, Z), e(X, Z)$ answer is 
  $e(X, Y), e(Y, Z), e(X, Z), p(X, Z, 1), p(Y, Z, 1), p(X, Y, 1)$
- **Rules can be generalized to compute shortest distance** 
  (replace constant $1$ by distance $D$)

| $p(X, Y, N) \setminus p(X, Y, M) \iff N \leq M \mid true.$ |
| $e(X, Y, D) \implies p(X, Y, D).$ |
| $e(X, Y, D), p(Y, Z, N) \implies p(X, Z, N+D).$ |
Partial order constraint

Partial order constraint (I)

duplicate @ X leq Y \ X leq Y <=> true.

reflexivity @ X leq X <=> true.
antisymmetry @ X leq Y , Y leq X <=> X=Y.
transitivity @ X leq Y , Y leq Z ==> X leq Z.

- Solver for partial order constraint $\leq$, represented by $\text{leq}$
- Implements duplicate removal and the three axioms
  - reflexivity removes constraints matching $\text{X} \leq \text{X}$
  - antisymmetry replaces $\text{X} \leq \text{Y}$ and $\text{Y} \leq \text{X}$ by $\text{X=Y}$
  - reflexivity adds $\text{X} \leq \text{Z}$ as redundant constraint
Partial order constraint (II)

Example

Query $A \leq B, C \leq A, B \leq C$ leads to

$A \leq B, C \leq A, B \leq C$

$A \leq B, C \leq A, B \preceq C, C \preceq B$

$A \preceq B, C \preceq A, B=C$

$A=B, B=C$

- Example shows use of propagation rule
- Starting from circular relationship equality of variables has been proven
Grammar parsing (I)

- **Grammar defined by rule of form** $\text{LHS} \rightarrow \text{RHS}$
  - Both sides consist of sequence of symbols
- **Two types of symbols**
  - Terminals corresponding to characters and strings
  - Nonterminals standing for sets of strings (defined by grammar rules)
- **LHS must contain at least one nonterminal**
Grammar parsing (II)

- In context-free grammar each LHS one single nonterminal.
- Grammar in Chomsky-normal form if all rules of the form $A \rightarrow T$ or $A \rightarrow B^* C$ (terminal $T$; nonterminals $A$, $B$, $C$).
- Cocke-Younger-Kasami (CYK) algorithm parses string according to context-free grammar in Chomsky-normal form.
- Bottom-up algorithm, specialization of transitive closure inheriting its properties.
- Example of dynamic programming.
Grammar parsing (III)

Grammar parser

duplicate @ p(A,I,J) \ p(A,I,J) <=> true.
terminal @ A->T, e(T,I,J) ==> p(A,I,J).
nonterminal @ A->B*C, p(B,I,J), p(C,J,K) ==> p(A,I,K).

- Represent string as graph chain of terminal symbols
- Edge constraint get additional argument for terminal
- Path constraint extended with additional argument for nonterminal
- Parse is restricted bottom-up computation of transitive closure
### Parse tree

**Grammar parser with parse tree**

<table>
<thead>
<tr>
<th>Rule Type</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>duplicate</strong></td>
<td>( p(A,I,J,P) \ | \ p(A,I,J,P) \iff true ).</td>
</tr>
<tr>
<td><strong>terminal</strong></td>
<td>( A \rightarrow T, \ e(T,I,J) \implies p(A,I,J,t(T)). )</td>
</tr>
<tr>
<td><strong>nonterminal</strong></td>
<td>( A \rightarrow B<em>C, \ p(B,I,J,P1), \ p(C,J,K,P2) \implies p(A,I,K,nt(B</em>C,J)). )</td>
</tr>
</tbody>
</table>

- Extend rules to generate parse tree
- Storing subparses and according grammar rules for each (partial) parse \( p \)
- Parse trees never removed
  - Parse trees can be looked up, no need to store them again
  - Corresponds to backpointer approach in imperative languages
  - Store arguments of rule head which do not occur in body in \( p \)
  - \( T \) stored for terminals, \( B, C, J \) for nonterminals
Generalizations (I)

Nondeterministic grammars

- Several definitions for one nonterminal
- Such grammars give rise to several possible (sub-) parses
- Generate a forest of (partial) parse trees
- Bottom-up approach generates all possible parse trees
- Several strings (even when sharing substrings) can be parsed simultaneously
Generalizations (II)

- Regular grammars only allow rules of the form
  \[ A \rightarrow T \text{ or } A \rightarrow T^*C \]

- It suffices to replace \( p \) by \( e \) in nonterminal

- Compiling grammar rules
  - For each grammar rule \( A \rightarrow B^*T^*C \) rule instance of the following form can be generated

  \[
  'A\rightarrow B^*T^*C' @ p(B,I,J), e(T,J,K), p(C,K,L) \\
  \Rightarrow p(A,I,L).
  \]

- For arbitrary grammar: chain path and edge constraints in CHR rule according to terminals and nonterminals in grammar
Ordered merging and sorting

Notation and representation

- Directed edge (arc) from node $A$ to node $B$ represented by binary constraint $A \rightarrow B$ (infix)

- Sequences represented as chain of arcs

- Example: sequence $0, 2, 5$ encoded as $0 \rightarrow 2, 2 \rightarrow 5$
Ordered merging (I)

Ordered merging program

\[ \begin{align*}
A \rightarrow B & \quad \text{\textbackslash} \quad A \rightarrow C \leftrightarrow A < B, B < C \quad \text{\textbar} \quad B \rightarrow C.
\end{align*} \]

- Assuming ascendingly order chains
  (A \rightarrow B \text{ implies } A =< B, B \text{ is immediate successor of } A)
- Program merges to chains starting with same smallest value by zipping
- Given A \rightarrow B \text{ and } A \rightarrow C, arc B \rightarrow C \text{ is added and now redundant arc } A \rightarrow C \text{ is removed}
- Query 0 \rightarrow 2, 0 \rightarrow 5 \text{ will result in } 0 \rightarrow 2, 2 \rightarrow 5
- Rule undoes transitive closure, flattens out branch in graph
Ordered merging (II)

Ordered merging program

\[ A \rightarrow B \ \ \ \ \ \ \ \ A \rightarrow C \iff A < B, B < C \ \ | \ \ B \rightarrow C. \]

- **A** denotes current position with branch
- All notes up to **A** have already been merged
- Both branches examined, smaller successor **B** kept, other arc replaced by \( B \rightarrow C \)
- If first chain not finished, now branch at node **B**
- Exhaustive rule application removes all such branches

Example computation

\[
\begin{align*}
0 & \rightarrow 2, \quad 2 \rightarrow 5, \quad 0 \rightarrow 3, \quad 3 \rightarrow 7.
0 & \rightarrow 2, \quad 2 \rightarrow 5, \quad 2 \rightarrow 3, \quad 3 \rightarrow 7.
0 & \rightarrow 2, \quad 3 \rightarrow 5, \quad 2 \rightarrow 3, \quad 3 \rightarrow 7.
0 & \rightarrow 2, \quad 2 \rightarrow 3, \quad 3 \rightarrow 5, \quad 5 \rightarrow 7.
\end{align*}
\]
Termination

- Rule application does not change number of arcs or node values
- Nodes on right side of arc do not change either
- Nodes on left side of arc might be replaced by larger node value
- Only finite number of values (and therefore of larger values)
  ⇒ Program terminates
Correctness

Compare to localized min program

- Query: two ordered chains with common smallest node
- Chains share longer and longer common prefix
- Rule applications maintains following invariant
  - Set of values does not change
  - Individual arcs are ordered
  - Graph is connected
  - Each node reachable from smallest node
  - Along all paths node values in ascending order
- No branch left when rule not applicable (except duplicate arcs)
  ⇒ Each node has unique immediate successor
  ⇒ Each path from smallest node must be in ascending order
  ⇒ Chains have been merged (only one chain which is ordered)
Duplicate removal

- Duplicate values \((A \to A)\) ignored due to rule guard
- Duplicate arcs \((A \to B, A \to B)\) ignored as well
- Two rules to remove duplicates
  
  \[
  A \to A \iff \text{true.}
  \]
  
  \[
  A \to B \setminus A \to B \iff \text{true.}
  \]

- Second rule redundant when guard of original rule generalized
  
  \[
  A \to B \setminus A \to C \iff A < B, B <= C \mid B \to C.
  \]

- \(A \to B, A \to B\) now produces \(A \to B, B \to B\) from which \(B \to B\) is removed
Sorting (I)

- Merging more than two chains simultaneously is possible
- Set of values represented as set of arcs of form $0 \rightarrow V$
- Sorting by merging those arcs into one chain

Example computation

$0 \rightarrow 2, 0 \rightarrow 5, 0 \rightarrow 1, 0 \rightarrow 7.$
$0 \rightarrow 2, 2 \rightarrow 5, 0 \rightarrow 1, 0 \rightarrow 7.$
$1 \rightarrow 2, 2 \rightarrow 5, 0 \rightarrow 1, 0 \rightarrow 7.$
$1 \rightarrow 2, 2 \rightarrow 5, 0 \rightarrow 1, 1 \rightarrow 7.$
$1 \rightarrow 2, 2 \rightarrow 5, 0 \rightarrow 1, 2 \rightarrow 7.$
$1 \rightarrow 2, 2 \rightarrow 5, 0 \rightarrow 1, 5 \rightarrow 7.$
Sorting (II)

- Program turns certain type of graph into ordered chain
- Ayn graph with ordered arcs where all nodes reachable from single root node can be sorted
- Quadratic complexity for sorting (with indexing)
- Optimal lin-log version discussed later
  (Ultraparallelism version: constant time complexity)