驰

Constraint Handling Rules - Basic CHR programs and their analysis

Prof. Dr. Thom Frühwirth | 2009 | University of Ulm, Germany

Table of Contents

Overview

Analysis of CHR programs regarding

- Logical reading and program correctness
- Termination and complexity
  - Upper bound from meta-complexity theorem
  - Actual worst-case complexity in CHR (refined semantics)
- Confluence
- Anytime and online algorithm property
- Concurrency and parallelism

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

Minimum program

```
min(N) \ min(M) <=> N=<M | true.
```

- Computes minimum of numbers given as
  min(n_1), min(n_2), ... , min(n_k)
- Keeps removing larger values until only one value remains

Example computation

<u>min(1)</u>, <u>min(0)</u>, min(2), min(1)

<u>min(0)</u>, <u>min(2)</u>, min(1)

<u>min(0)</u>, <u>min(1)</u>

min(0)

Logical reading (I)

- `min` constraints represent candidates for minimum
  - Actual minimum remains when calculation finished
  - Cannot be expressed straightforward in first-order logic
- First-order logic reading

$$\forall (N \leq M \rightarrow (min(N) \wedge min(M) \leftrightarrow min(N))$$

Logically equivalent to

$$N \leq M \rightarrow (min(M) \leftarrow min(N))$$

  - "Given a minimum, any larger value is also a minimum"

Logical reading (II)

- Linear logic reading

$$!\forall\,((N \leq M) \multimap (min(N) \otimes min(M) \multimap min(N)))$$

- Reads as: Of course, consuming `min(N)` and `min(M)` where `(N=<M)` produces `min(N)`
- Properly reflects the dynamics of the minimum computation.

Correctness

Correctness by contradiction

- Minimum is not correctly computed
  - Case 1: more than one `min` constraint left
  - Case 2: remaining `min` constraint does not contain minimum
- Case 1: rule is still applicable
- Case 2: minimum must have been removed
  - Contradiction: rule always removes larger value

Termination and worst-case complexity

- Termination
  - Rule removes constraints, does not introduce new ones
  - Rule application in constant time (applies to every pair of min constraints)
  - Number of rule applications (derivation length) bounded by number of `min` constraints
- Worst-case time complexity
  - Given n `min` constraints
  - $O(n)$ under refined semantics (left-to-right, immediate reaction, one constraint will be removed)

Meta-complexity

- Abstract semantics: undetermined order of tried constraints and rules
- Meta-complexity theorem (MCT)

$$O(D\sum_i((n+D)^{n_i}(O_{H_i}+O_{G_i}) \;+\; (O_{C_i}+O_{B_i}))),$$

(D derivation length, i ranges over rules, n_i number of head constraints in ith rule, costs O_{H_i} of head matching, O_{G_i} of guard checking, O_{C_i} of imposing built-in constraints of body, O_{B_i} of imposing CHR constraints of body)

- In this case

$$O(n(n^2(1+0) \;+\; (1+0))) \;=\; O(n^3).$$

- Highly over-estimates (applies to all two-head simpagation rules)

Confluence (I)

- Correctness implies result is single specific `min` constraint
  ⇒ Program is confluent for ground queries (ground confluent)
- One rule, only overlaps with itself
- One nontrivial full overlap (all head constraints equated):
  `min(A),min(B), A=<B,B=<A.`
  (equivalent to `min(A),min(A), A=B.`)
- Apply rule in given or reversed order
  - Both cases lead to `min(A), A=B` (hence rule removes duplicates)

Confluence (II)

- Four overlaps where one constraint shared

```
min(A),min(B),min(C), A=<B,B=<C.
min(A),min(B),min(C), A=<B,B=<C.
min(A),min(B),min(C), A=<B,A=<C.

min(A),min(B),min(C), A=<B,C=<B.
```

- First (and second) overlap leads to joinable critical pair

```
                min(A),min(B),min(C), A=<B,B=<C
                   /                      \
min(A),min(B), A=<B,B=<C    min(A),min(C), A=<B,B=<C
                   \                      /
                     min(A), A=<B,B=<C
```

  - Only smallest constraint `min(A)` is left

Confluence (III)

- Next overlap (similar)

```
                   min(A),min(B),min(C),  A=<B,A=<C
                        /                   \
min(A),min(B),  A=<B,A=<C    min(A),min(C),  A=<B,A=<C
                         \                   /
                     min(A),  A=<B,A=<C
```

- Last overlap

```
                   min(A),min(B),min(C),  A=<B,C=<B
                             |                   |
                        min(A),min(C),  A=<B,C=<B
```

  - Cannot proceed until relationship between `A` and `C` known (but then common state is reached)

$\Rightarrow$ Program is confluent

Anytime algorithm property

- Anytime algorithm (approximation)
  - One can interrupt program at any time and restart on immediate result
    - On interrupt: subset of initial `min` constraints containing actual minimum

      $\Rightarrow$ interruption and restart possible
  - Intermediate results approximate final result
    - Set of possible minima gets smaller and smaller

$\Rightarrow$ Program is an anytime algorithm

Online algorithm property

- Online (incremental)
  - Possibility to add constraints while program is running
    - Additional `min` constraints can be added at any point
    - Immediately react with other constraints
    - Confluence guarantees same result, no matter when constraint is added

$\Rightarrow$ Program is incremental

Concurrency and parallelism (I)

- Program is well-behaved (terminating, confluent)
  $\Rightarrow$ parallelization easy
- Weak parallelism
  - Apply rule to different nonoverlapping parts of query
    - Rule can be applied to pairs of `min` constraints in parallel
    - Halves number of `min` constraints in each parallel computation step
    - $O(log(n))$ on $n/2$ parallel processing units (processors)

Example computation

<u>`min(1)`</u>, <u>`min(0)`</u>, <u>`min(2)`</u>, <u>`min(1)`</u>

<u>`min(0)`</u>, <u>`min(1)`</u>

`min(0)`

Concurrency and parallelism (II)

- Strong parallelism
  - Apply rule to overlapping parts of query (fix one `min` constraint to be kept)
  - Linear complexity as in sequential execution (worst-case: with largest value fixed, no rule application possible)
- Cost (Time complexity times number of processors)
  - Parallel execution: $O(n\ log(n))$
  - Sequential execution: $O(n)$

Boolean XOR

XOR program

```
xor(X),   xor(X) <=> xor(0).
xor(1) \ xor(0) <=> true.
```

- Implements Exclusive Or operation of propositional logic (0 means false, 1 means true)
- Query: multiset of `xor` constraints for input truth values (e.g. `xor(1), xor(0), xor(0), xor(1)`)
- First rule: Identical inputs replaced by `xor(0)`
- Second rule: Remove `xor(0)` if there is `xor(1)`

Logical reading and correctness

- First-order logical reading
  - $xor(X) \leftrightarrow xor(0)$ (particularly $xor(1) \leftrightarrow xor(0)$)
  - Means all `xor` constraints are equivalent
  - Resort to linear logic reading
- Correctness
  - Map CHR conjunction to xor operation
    - Both associative, commutative, not idempotent
  - Each rule application computes one xor
  - One `xor` constraint left in the end

Termination and complexity

- Terminating
  - Each rule removes more constraints than it introduces
- Complexity
  - For each pair of constraints one rule application in constant time
  - Linear complexity under refined semantics
  - Cubic complexity under abstract semantics

Confluence (I)

XOR program

```
xor(X),  xor(X) <=> xor(0).
xor(1) \ xor(0) <=> true.
```

- Overlap `xor(X), xor(X)`
  - First rule fully with itself
  - Always leads to `xor(0)`
- Overlap `xor(X), xor(X), xor(X)`
  - First rule with itself
  - Always leads to `xor(0), xor(X)`

Confluence (II)

XOR program

```
xor(X),  xor(X) <=> xor(0).
xor(1) \ xor(0) <=> true.
```

- Overlap `xor(1), xor(1), xor(0)`
  - Occurs twice (first and second rule, second rule with itself)
  - Always leads to `xor(0)`
- Overlap `xor(1), xor(0), xor(0)`
  - Occurs twice (first and second rule, second rule with itself)
  - Always leads to `xor(1)`

$\Rightarrow$ Program is confluent

Remaining properties

- Anytime: fewer and fewer `xor` constraints, result not necessarily contained (`xor(1), xor(1)`)
- Online: `xor` constraints can be added at any point
- Rules applicable in parallel (as for `min`) $\Rightarrow O(n\ log(n))$

Greatest common divisor

GCD program

```
      gcd(0) <=> true.
sub @ gcd(N) \ gcd(M) <=> 0<N,N=<M | gcd(M-N).
mod @ gcd(N) \ gcd(M) <=> 0<N,N=<M | gcd(M mod N).
```

- Either `sub` or `mod` rule can be used

Example computation (`sub`)

```
gcd(7), gcd(12)
gcd(7), gcd(5)
gcd(5), gcd(2)
gcd(2), gcd(3)
gcd(2), gcd(1)
gcd(1), gcd(1)
gcd(1), gcd(0)
gcd(1)
```

Example computation (`mod`)

```
gcd(7), gcd(12)
gcd(7), gcd(5)
gcd(5), gcd(2)
gcd(2), gcd(1)
gcd(1), gcd(0)
gcd(1)
```

Logical Reading

- First-order logical reading

$$gcd(0) \leftrightarrow true$$

$$0{<}N \wedge N{=<}M \rightarrow (gcd(N) \wedge gcd(M) \leftrightarrow gcd(N) \wedge gcd(M{-}N))$$

Latter is equivalent to

$$0{<}N \wedge 0{=<}M \rightarrow (gcd(N) \wedge gcd(M{+}N) \leftrightarrow gcd(N) \wedge gcd(M))$$

  - Correct, but does not characterize gcd, only all its multiples
  - Linear-logic semantics reflects dynamics of computation properly

Correctness

- All divisors d preserved under rule application
- Computation produces smaller and smaller values
  - $N = Ad$, $M = Bd$
  - From logical reading

$$0{<}Ad \land Ad{=<}Bd$$
$$\rightarrow (gcd(Ad) \land gcd(Bd) \leftrightarrow gcd(Ad) \land gcd(Bd{-}Ad))$$

  - $gcd(Bd - Ad)$ is equivalent to $gcd((B - A)d)$

$\Rightarrow$ Divisor d preserved during computation

- Computation continues until $M = N = gcd$
- Rule is applied a last time
- `gcd(0)` is removed leaving only actual gcd

Termination and complexity

- Termination
  - Guard condition ensure new value smaller than removed `M`
  - New value cannot become negative
- Complexity
  - Rules applicable in constant time to any `gcd` pair
  - Two `gcd` constraints
    - `sub`: complexity linear in larger number
    - `mod`: complexity logarithmic in larger number
  - More than two `gcd` constraints: consider all numbers
    - `sub` linear in sum of numbers
    - `mod` logarithmic in product of numbers

Confluence

- GCD program is ground confluent (unique result for given values)
- Not confluent in general:
  - Overlaps analogous to `min`
  - Difference: rule not only removes constraints but also adds
  - Nonjoinable critical pair (cp)

```
          gcd(A),gcd(B),gcd(C),0<A,A=<B,0<C,C=<B
                |                           |
gcd(A),gcd(B-A),gcd(C),0<A,A=<B,0<C,C=<B    |
                                            |
        gcd(A),gcd(B-C),gcd(C),0<A,A=<B,0<C,C=<B
```

  - Computation cannot proceed until relationship of `A`, `B`, and `C` is known

Remaining properties

- Anytime: fewer and fewer `gcd` constraints with smaller and smaller numbers (result not necessarily contained)
- Online: additional `gcd` constraints can be added anytime
- Complexity of parallel execution not better nor worse than sequential (since $O(max(a, b)) = O(a + b)$)
- But gcd's may get smaller more quickly
- In practice: super-linear speed up with parallel CHR implementation in Haskell

Prime sieve

Prime sieve program

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

- Removes multiples in given set until only prime numbers left
- Query: prime candidates from 2 upto N
  (`prime(2), prime(3), prime(4),..., prime(N)`)

Example computation

```
prime(7), prime(6), prime(5), prime(4), prime(3), prime(2)
prime(7), prime(5), prime(4), prime(3), prime(2)
prime(7), prime(5), prime(3), prime(2)
```

Logical reading and correctness

- First-order logical reading

$$\forall((M \text{ mod } N = 0) \rightarrow (prime(M) \wedge prime(N) \leftrightarrow prime(N)))$$

  - Means a number is prime if it is a multiple of another prime number
  - Linear logic reading reflects dynamics of filtering correctly
- Correctness
  - Program confluent $\Rightarrow$ result always the same
  - All composite numbers removed (with correct query)
  - Primes not removed (only multiple of 1, not included)

Termination and complexity

- Termination
  - Rule only removes constraints
- Complexity
  - Rule not applicable to all pairs of numbers
  - Thus complexity quadratic in number of constraints (refined semantics)
  - Runtime can be improved by starting from lower numbers

Confluence

- Program is confluent
  - Reason: transitivity of divisibility
  - `I|J` and `J|K` $\Rightarrow$ `I|K`
  - Overlaps and joinability analogous to `min`

```
prime(A),prime(B),          A|B,B|A
prime(A),prime(B),prime(C), A|B,B|C
prime(A),prime(B),prime(C), A|B,A|C
prime(A),prime(B),prime(C), A|B,C|B
```

  - First three overlaps lead to joinable critical pair
  - Last overlap also:

```
prime(A),prime(B),prime(C), A|B, C|B
           |                  |
         prime(A),prime(C), A|B, C|B
```

Other properties

- Anytime and online properties as for `min`
- `sift` does not hold for all pairs
  - All $O(n^2)$ pairs have to be tried in $O(n)$ rounds
    $\Rightarrow$ some scheduling needed
- Strong parallelism
  - Fix one prime constraint for first head constraint
  - Search for prime constraint matching second head constraint
  - Needs $O(n)$ rounds
- Cost same as for sequential execution (quadratic)
- Linear time: maximal, linear parallel speed-up

Exchange sort

Exchange sort program

```
a(I,V), a(J,W) <=> I>J, V<W | a(J,V), a(I,W).
```

- Exchanges values that are in the wrong order
- Query: array of values A_i (`a(1,A1),...a(n,An)`)

Example computation

```
a(0,1), a(1,7), a(2,5), a(3,9), a(4,2)
a(0,1), a(1,5), a(2,7), a(3,2), a(4,9)
a(0,1), a(1,5), a(2,2), a(3,7), a(4,9)
a(0,1), a(1,2), a(2,5), a(3,7), a(4,9)
```

Logical reading and correctness

- First-oder logical reading

$$I>J \land V< W \rightarrow (a(I,V) \land a(J,W) \leftrightarrow a(J,V) \land a(I,W))$$

  - Means all arrays with same set of values are equivalent
  - Resort to linear logic reading
- Correctness
  - Sorted: for each `(a(I,V), a(J,W))` with `I>J` it holds that `V>=W`
  - If condition $V \geq W$ does not hold, rule is applicable
    $\Rightarrow$ condition holds after application
    $\Rightarrow$ if rule not applicable, array must be sorted

Termination

- Rule application cannot introduce more wrong than right orderings
- Guard: counter in each array entry
  - Counts how many values with larger index are smaller
  - On exchange:
    - Counter of smaller value increases
    - Counter of larger value decreases by same number $+1$
    - Counter of values in between can only decrease

  $\Rightarrow$ Sum of counters decrease with each rule application

Complexity

- Derivation length quadratic in number of constraints (cf. counter)
- Two head constraints: MCT gives overestimated complexity

$$O(n^2((n^2)^2(1+1)\ +\ (0+1)))\ =\ O(n^6)$$

- Fix one value (refined semantics): Each try costs $O(n)$
  - Rule can be applied in constant time once pair found
- At most $O(n^2)$ applications $\Rightarrow$ actual worst-case complexity $O(n^3)$

Confluence (I)

- Program is ground confluent by correctness (unique result)
- Not confluent in general
- First critical pair is joinable

```
      a(I,V), a(J,W), a(K,U), I>J,V<W, J>K,W<U
         / I,J                            | J,K
        /       a(I,V), a(K,W), a(J,U), I>J,V<W, J>K,W<U
       /                                  |
a(J,V), a(I,W), a(K,U), I>J,V<W, J>K,W<U  |
      | I,K                               | I,K
      |   a(K,V), a(I,W), a(J,U), I>J,V<W, J>K,W<U
      |                                   |
a(J,V), a(K,W), a(I,U), I>J,V<W, J>K,W<U  |
       \ J,K                              | I,J
        \                                 |
      a(K,V), a(J,W), a(I,U), I>J,V<W, J>K,W<U
```

Confluence (II)

- Two nonjoinable critical pairs

```
          a(I,V), a(J,W), a(K,U), I>J,V<W, I>K,V<U
                 / I,J                        |
a(J,V), a(I,W), a(K,U), I>J,V<W, I>K,V<U     |
                                              | I,K
          a(K,V), a(J,W), a(I,U), I>J,V<W, I>K,V<U
```

  - Only joinable when relationship between `J` and `K` as well as `W` and `U` known
- Analogous situation for

```
          a(I,V), a(J,W), a(K,U), I>K,V<U, J>K,W<U
```

Remaining properties

- Number of wrongly ordered pairs decreases over time
- Additional array entries can be added at any point
- Rule not applicable to arbitrary pairs of constraints
  $\Rightarrow$ only weak parallelism possible
  - Associate each array entry with a processor
  - Try all pairs in $O(n)$ (macro-step)
  - Each entry reacts with at most $O(n)$ other entries
  - Overall $O(n^2)$ rule applications
  - All rule applications can be performed in $O(n)$ macro-steps $\Rightarrow$ Complexity quadratic, cost cubic

Square root

Square root program

```
sqrt(X,G) <=> abs(G*G/X-1)>eps | sqrt(X,(G+X/G)/2).
```

- Rule implements Newton's method
- `sqrt(X,G)`: square root of `X` is approximated by `G`
- `eps` is greater but close to 0
- Start with positive numbers `X` and `G`

Logical reading and termination

- Logical reading

$$abs(G * G/X - 1) > \epsilon \rightarrow (sqrt(X, G) \leftrightarrow sqrt(X, (G + X/G)/2))$$

  - Means that any value is an approximation of $\sqrt{\mathtt{X}}$
  - Resort to linear logic reading
- Termination
  - After first rule application $\mathtt{G} \geq \sqrt{\mathtt{X}}$
  - If $\mathtt{G} = \sqrt{\mathtt{X}}$ rule not applicable
  - Otherwise rule applicable, second argument will decrease

Remaining properties

- Confluence, anytime, online algorithm
  - Hold trivially (single rule with single head constraint)
- Concurrency, parallelism
  - Several constraints can run independently in parallel

Maximum

Maximum program

```
max(X,Y,Z) <=> X=<Y | Z=Y.
max(X,Y,Z) <=> Y=<X | Z=X.
```

- `max(X,Y,Z)` means `Z` is the maximum of `X` and `Y`
- `=<` and `<` built-ins

Example computation

`max(1,2,M)`: first rule applicable, reduces to `M=2`
`max(1,2,3)`: fails because of built-in `3=2`
`max(1,1,M)`: both rules applicable, reduces to `M=1`

Logical reading and correctness

- First-order logical reading is

$$X \leq Y \rightarrow (max(X, Y, Z) \leftrightarrow Z = Y)$$

$$Y \leq X \rightarrow (max(X, Y, Z) \leftrightarrow Z = X)$$

- Logical consequences of the definition of `max`

$$max(X, Y, Z) \leftrightarrow (X \leq Y \wedge Z = Y \ \vee \ Y \leq X \wedge Z = X)$$

- This shows logical correctness

Termination and complexity

- One constraint removed in each step
  $\Rightarrow$ At most n (number of constraints) derivation steps
- In each step at most n constraints checked against rules
- Checking or establishing syntactic equality in constant time
- Matching constraint against rule in quasi-constant time
- Rule application in quasi-constant time

$\Rightarrow$ Worst-case complexity slightly worse than $O(n^2)$

- Same complexity is obtained using MCT

Remaining properties

- Confluence
  - Only overlap is $\mathtt{max}(\mathtt{X}, \mathtt{Y}, \mathtt{Z}) \wedge X \leq Y \wedge Y \leq X$
  - Leads to critical pair

$$(Y = Z \wedge X \leq Y \wedge Y \leq X, X = Z \wedge X \leq Y \wedge Y \leq X)$$

  - Both states equivalent to $X = Y \wedge Y = Z$
- Anytime, online algorithm
  - Hold trivially (single-headed simplification rule)
- Concurrency, parallelism
  - `max` constraint may have to wait for result of other constraint (e.g. `max(X,Y,Z), max(Y,Z,W)`)

Fibonacci numbers

Fibonacci program

```
f0 @ fib(0,M)  <=> M=1.
f1 @ fib(1,M)  <=> M=1.
fn @ fib(N,M)  <=> N>=2 |
       fib(N-1,M1), fib(N-2,M2), M is M1+M2.
```

- `fib(N,M)` holds if `M` is `N`th Fibonacci number

Example computations

Query `fib(8,A)` yields `A=34`
Query `fib(12,233)` succeeds
Query `fib(11,233)` fails
Query `fib(N,233)` delays

Logical reading and correctness

- First-oder logical reading

$$fib(0, M) \leftrightarrow M = 1$$

$$fib(1, M) \leftrightarrow M = 1$$

$$N \geq 2 \rightarrow (fib(N, M) \leftrightarrow fib(N-1, M1) \wedge fib(N-2, M2) \wedge M = M1 + M2)$$

- Shows correctness (coincides with mathematical definition)

Termination and complexity

- Program terminates
  - First argument of `fib` decreases in each call
  - Call only possible with positive first argument
- Ranking gives upper bound on derivation length

$$rank(fib(n,m)) = 2^n.$$

- Expected exponential complexity $O(2^n)$
- If first argument unknown complexity may increase (depending on wake-up policy)
- MCT reflects this and gives $O(4^n)$

Remaining properties

- Confluence:
  - No overlaps (single-headed simplification rules whose heads and guards exclude each other)
- Anytime, online algorithm, and concurrency
  - Hold trivially (single-headed simplification rule)

Fibonacci numbers (memorization version)

Fibonacci program with memorization

```
mem @ fib(N,M1) \ fib(N,M2) <=> M1=M2.

f0 @ fib(0,M) ==> M=1.
f1 @ fib(1,M) ==> M=1.
fn @ fib(N,M) ==> N>=2 |
       fib(N-1,M1), fib(N-2,M2), M is M1+M2.
```

Example computations

Query `fib(8,A)` returns all Fibonacci numbers up to `8`:

`fib(0,1), fib(1,1), fib(2,2), ..., fib(7,21), fib(8,34)`

Complexity

- With indexing on the first argument
  - Linear complexity (each Fibonacci number only computed once)
- Without indexing on first argument
  - Quadratic complexity (Searching for suitable pairs in `mem`)
- MCT does not apply here (propagation rules)

Confluence

- Nontrivial overlaps between `mem` and each propagation rule
- First critical pair: `fib(0,M1), fib(0,M2)`

```
                fib(0,M1), fib(0,M2)
                / mem          \  f0
fib(0,M2), M1=M2    fib(0,M1), M1=1, fib(0,M2)
             | f0               | mem
fib(0,M2), M1=M2, M2=1 ≡ M1=M2, M1=1, fib(0,M2)
```

Confluence

- Second critical pair `fib(N,M1), fib(N,M2)` (shown split)

```
                fib(N,M1)
                  | mem
              fib(N,M2),M1=M2
                  | fn
fib(N,M2),M1=M2,fib(N-1,M5),fib(N-2,M6),M2 is M5+M6
```

```
                fib(N,M2)
                  | fn
fib(N,M1),fib(N-1,M3),fib(N-2,M4),M1 is M3+M4,fib(N,M2)
                  | mem
M1=M2,fib(N-1,M3),fib(N-2,M4),M1 is M3+M4,fib(N,M2)
```

- The two reached states are equivalent
- Overlap with rule `f1` analogous

Other properties

- Online: trivial
- Anytime
  - In theory: no computation steps redone when started on intermediate result
  - In practice: recomputation my occur (propagation history not explicit)
  - Additional computations absorbed (confluence and `mem` rule)
  - Execution of two recursive calls in parallel possible
  - No gain: `mem` rule will absorb multiple computations

Fibonacci numbers (program variations)

- Similar reasoning, results hold for `fib` as function with given first argument
- Exception: finite bottom-up computation

```
fn @ fib_upto(Max), fib(N1,M1), fib(N2,M2)
          ==> Max>N2,N2=:=N1+1 | fib(N2+1,M1+M2).
```

  - Quadratic complexity (no indexing between to `fib` constraints in head)

Depth-first search

Depth-first search program

```
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).
```

- Tree encoding `node(Data, Lefttree, Righttree)`
- Data ordered such that every node in left subtree smaller, every node in right subtree larger than parent node
- Search for datum `Data` in binary tree `Tree` by calling `dfsearch(Tree, Data)`
- All analyzed properties hold in a trivial way (single-headed simplification rules with exclusive heads and guards)
- Complexity linear in depth in tree per search

Depth-first search

Depth-first search program (variant)

```
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: node represented by CHR data constraint
- Tree is set of such `node` constraints
- For valid binary search tree properties of previous programs inherited
- With indexing complexity unaffected
- Data constraints can be added $\Rightarrow$ online algorithm

Depth-first search

Depth-first search program (another variant)

```
found @ node(N) \ search(N) <=> true.
empty @ search(N) <=> fail.
```

- Directly access data by mentioning in rule head
- All properties except anytime break down (due to `empty` rule)
- With indexing constant time complexity

Destructive assignment

Destructive assignment program

```
assign(Var,New), cell(Var,Old) <=> cell(Var,New).
```

- Constraint `assign` assigns new value to variable `Var`
- Not confluent
  - Nonjoinable overlap:

    `assign(Var,New1), assign(Var,New2), cell(Var,Old)`
  - Results in either `cell(Var,New1)` or `cell(Var,New2)`
- Order matters $\Rightarrow$ not executable in parallel as intended
- First-order logical reading does not reflect intended meaning (linear-logic semantics needed)

Transitive closure

Transitive closure program

```
dp @ p(X,Y) \ p(X,Y) <=> true.
p1 @ e(X,Y) ==> p(X,Y).
pn @ e(X,Y), p(Y,Z) ==> p(X,Z).
```

- Relation: edge `e` between two nodes
- Transitive closure: path `p` between two nodes

Example computation

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)`

Logical reading and correctness

- First-order logical reading as implications

  $p(X,Y) \wedge p(X,Y) \leftrightarrow p(X,Y)$
  $e(X,Y) \rightarrow p(X,Y)$
  $e(X,Y) \wedge p(Y,Z) \rightarrow p(X,Z)$

  - Logical reading of duplicate removal is tautology
  - Not expressible in FOL but in linear logic: transitive closure is smallest transitive relation
  - Rules actually calculate smallest relation (left to right application produces relation bottom-up)

$\Rightarrow$ Program is correct

Termination

- Refined semantics
  - Duplicates removed by `dp` before propagation rules applied
  - Finite number of paths in finite graph

$\Rightarrow$ Program terminates

- Abstract semantics
  - `dp` can be applied too late in cyclic graph
  - Same paths generated again and again

$\Rightarrow$ Termination not guaranteed

Complexity (I)

- It holds that $v/2 \le e \le p \le v^2$ (v #vertices, e #edges, p #paths)
- Rules can be applied in constant time
- Without indexing
  - Upper bound for propagation rule attempts: product of number of head constraints occurring during computation
  - `p1` tried at most e times, applies e times
  - `pn` tried at most ep times, applies at most $\max(ev, vp) = vp$ times
  - Path constraint produced with each rule application
  - Thus, `dp` applied pvp times

$\Rightarrow$ Worst-case complexity due to `dp` $O(vp^2) = O(v^5)$

Complexity (II)

- With indexing
  - Index constraints on arguments with shared variables in heads
  - Upper bounds for rule attempts and rule application coincide now
  - `p1` tried and applied at most e times
  - `pn` tried and applied at most $\max(ev, vp) = vp$ times
  - Thus, `dp` applied vp times now

$\Rightarrow$ Worst-case complexity due to `pn` $O(vp) = O(v^3)$

- Optimal for this algorithm

Confluence

- Only nontrivial overlap (between `dp` and `pn`)

```
          e(X,Y), p(Y,Z), p(Y,Z)
            / dp                \ pn
e(X,Y), p(Y,Z)    e(X,Y), p(Y,Z), p(Y,Z), p(X,Z)
            \ pn                / dp
          e(X,Y), p(Y,Z), p(X,Z)
```

- Program is confluent

Remaining properties

- Anytime: Repeated application of propagation rule does not matter
  - Confluence, duplicate paths removed
- Online: edges can be added during computation
- Strong parallelism
  - Apply `p1` to all edges in parallel
  - Next rounds: all possible applications of `pn` and then `dp`
  - With indexing vp application of those rules
  - Given v processors parallel complexity $O(v^2)$
  - Cost $O(v^3)$

Single-source and single-target paths

Transitive closure program(single-source)

```
dp @ p(X,Y) \ p(X,Y) <=> true.
s1 @ source(X), e(X,Y) ==> p(X,Y).
sn @ source(X), p(X,Y), e(Y,Z) ==> p(X,Z).
```

- Only paths from (or to) a certain node computed
- Complexity
  - Number of created path constraints reduced by factor v $(p \leq v \leq 2e \leq 2v^2)$
  - Without indexing $O(vp^2) = O(v^3)$
  - With indexing: $O(vp) = O(v^2)$

Shortest path

Shortest path program

```
dp @ p(X,Y,N) \ p(X,Y,M) <=> N=<M | true.
e(X,Y) ==> p(X,Y,1).
e(X,Y), p(Y,Z,N) ==> p(X,Z,N+1).
```

- Computes shortest path length between all pairs of nodes

Example computation

Query `e(X,Y),e(Y,Z),e(X,Z)` adds path constraints

`p(X,Z,1), p(Y,Z,1), p(X,Y,1)`

Termination and complexity

- New active path constraint only removed by `dp` if equal or longer
- Otherwise old path removed (work repeated, at most v times)
  $\Rightarrow$ Worst-case complexity with indexing $O(ev^2) = O(v^4)$
- Better complexity (i.e. ev) needs more clever scheduling
  - E.g. in Dijkstra's algorithm, computation always continues with shortest path found so far

Partial order constraint

Partial order program

```
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.
```

- Maintains nonstrict partial order relation leq $\leq$

Example computation

```
A leq B, C leq A, B leq C
A leq B, C leq A, B leq C, C leq B
A leq B, C leq A, B=C
A=B, B=C
```

Termination and complexity

- `duplicate` and `transitivity` analog to transitive closure (i.e. cubic)
- `reflexivity` does not change complexity
- With indexing
  - Application of `antisymmetry` triggers at most $O(v)$ constraints (all `leq` with `X` and `Y`)
  - In those constraints, one variable is replaced by other
    $\Rightarrow$ problem shrinks by one variable (at most $O(v)$ times)
    $\Rightarrow$ Thus, `antisymmetry` applied $O(v)$ times
    $\Rightarrow$ Trying and applying of `antisymmetry`: $O(v^2)$

$\Rightarrow$ Overall complexity $O(v^3)$

Remaining properties

- Algorithm is anytime and online (as discussed in chapter 4)
- Similar to transitive closure: `transitivity` can be applied in parallel to all pairs, then all other rules can be applied
- First-order logical reading

$$\begin{array}{l}
\texttt{(duplicate)}\ \forall\ X,Y\ (X \leq Y \land X \leq Y \Leftrightarrow X \leq Y)\\
\texttt{(reflexivity)}\ \forall\ X\ (X \leq X \Leftrightarrow \mathit{true})\\
\texttt{(antisymmetry)}\ \forall\ X,Y\ (X \leq Y \land Y \leq X \Leftrightarrow X = Y)\\
\texttt{(transitivity)}\ \forall\ X,Y,Z\ (X \leq Y \land Y \leq Z \Rightarrow X \leq Z)
\end{array}$$

  - `duplicate` rule is tautology
  - Other rules give axioms of partial order
  - FOL reading suffices and shows correctness (see also chapter 3)

Cocke-Younger-Kasami

CYK algorithm

```
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).
```

- Parses a string according to a context-free grammar bottom-up.
- Specialization of transitive closure

Termination and complexity

General idea: With indexing:

- Arguments of constraints can be associated with finite domains
  $\Rightarrow$ Product of domain sizes of variables in rule head gives upper bound on number of rule applications and attempts
- Chain representing string has v nodes and $e(= v - 1)$ edges
- Grammar with t terminals and n nonterminals
- Number of grammar rules $r \leq nt + n^3$(assuming $t \leq n^2$)
- Products of domain sizes
  - `terminal` (variables `A, T, I, J`): $ntvv = ntv^2$
  - `nonterminal` (variables `A, B, C, I, J, K`): n^3v^3
  - `duplicate` tried with each `p` produced: n^3v^3

$\Rightarrow$ Overall complexity of $O(n^3v^3)$ with indexing (n usually fixed)

Confluence

- Confluent when used on ground chains
- Not confluent in general
- Nonjoinable critical pair from overlap

```
             A->B*B, p(B,I,I), p(B,I,I)
              / nonterminal          \ duplicate
A->B*B, p(B,I,I), p(B,I,I), p(A,I,I)  A->B*B, p(B,I,I)
              | duplicate
A->B*B, p(B,I,I), p(A,I,I)
```

Mergesort

Merge sort program

```
A -> B \ A -> C <=> A<B,B=<C | B -> C.
```

- Implements merge sort algorithm
- Query contains only arcs `0 -> ` A_i
- Answer: sequence of values stored as arcs (e.g `0,2,5` is `0 -> 2, 2->5`)

Example computation

```
0 -> 2, 0 -> 5, 0 -> 1, 0 -> 7.
0 -> 2, 2 -> 5, 0 -> 1, 0 -> 7.
1 -> 2, 2 -> 5, 0 -> 1, 0 -> 7.
1 -> 2, 2 -> 5, 0 -> 1, 1 -> 7.
1 -> 2, 2 -> 5, 0 -> 1, 2 -> 7.
1 -> 2, 2 -> 5, 0 -> 1, 5 -> 7.
```

Logical reading and correctness

- Classical logical reading is sufficient

$$A{<}B \land B{->}C \rightarrow (A{->}B \land A{->}C \leftrightarrow A{->}B \land B{->}C).$$

- `A -> B` means $A \leq B$, thus logical correctness is consequence of axioms for $\leq$

$$A{<}B \land B{\leq}C \rightarrow (A{\leq}B \land A{\leq}C \leftrightarrow A{\leq}B \land B{\leq}C)$$

Termination and complexity

- Complexity of merging two ordered chains (lengths n and m)
  - Indexing on the first argument of arc constraint:
    $\Rightarrow$ Second arc constraint found in constant time since rule is applicable to arbitrary pairs of arcs with same first argument
  - Each rule application processes one arc constraint
    $\Rightarrow O(m+n)$
- Complexity of sorting n values given as second argument of arc
  - First argument can be replaced at most n times in each arc
    $\Rightarrow$ Worst time complexity $O(n^2)$

Confluence

- Ground confluent (correct, unique result)
- Overlaps and joinability analog to gcd
  (`gcd(N)` mapped to `X->N`, `gcd(M-N)` to `N->M`)
- One nonjoinable overlap

```
          X->A,X->B,X->C, X<A,A=<B, X<C,C=<B
                /                     |
X->A,A->B,X->C, X<A,A=<B, X<C,C=<B    |
                                      |
          X->A,C->B,X->C, X<A,A=<B, X<C,C=<B
```

- Cannot proceed until relationship between `A` and `C` is known

Anytime and online algorithm

- Anytime property
  - Intermediate results: connected acyclic graph
  - Smallest value is root
  - Longer and longer chains without branches
- Online property
  - Sorting incrementally, new arcs can be added at any time

Mergesort (optimal complexity sorting)

- Complexity can be improved to optimal $O(n\ \log(n))$ by optimal merging order
- Merging chains of same length
  - Precede chain with length (`N=>Firstnode`)
  - Rule to initiate merging of chains of same length

    ```
    N=>A, N=>B <=> A<B | N+N=>A, A->B.
    ```
  - Works only if length of query is a power of 2
- Start by merging n chains of length 1 then merge $n/2$ chains of length 2 and so on
- Finished after $\log(n)$ rounds $\Rightarrow$ complexity $O(n\ log(n))$
- works for any length with one more rule

Concurrency and parallelism

- Follows structure of proof of optimal complexity
- Merging of two chains strictly sequential
- In second round start merging new chains while tail of chains still produced
- $\log(n)$ rounds of merging, last round my need n more steps
- Overall $n + \log(n)$ steps
- With n processors: complexity $O(n)$ and cost $O(n^2)$
- Also possible for original version (scheduling)