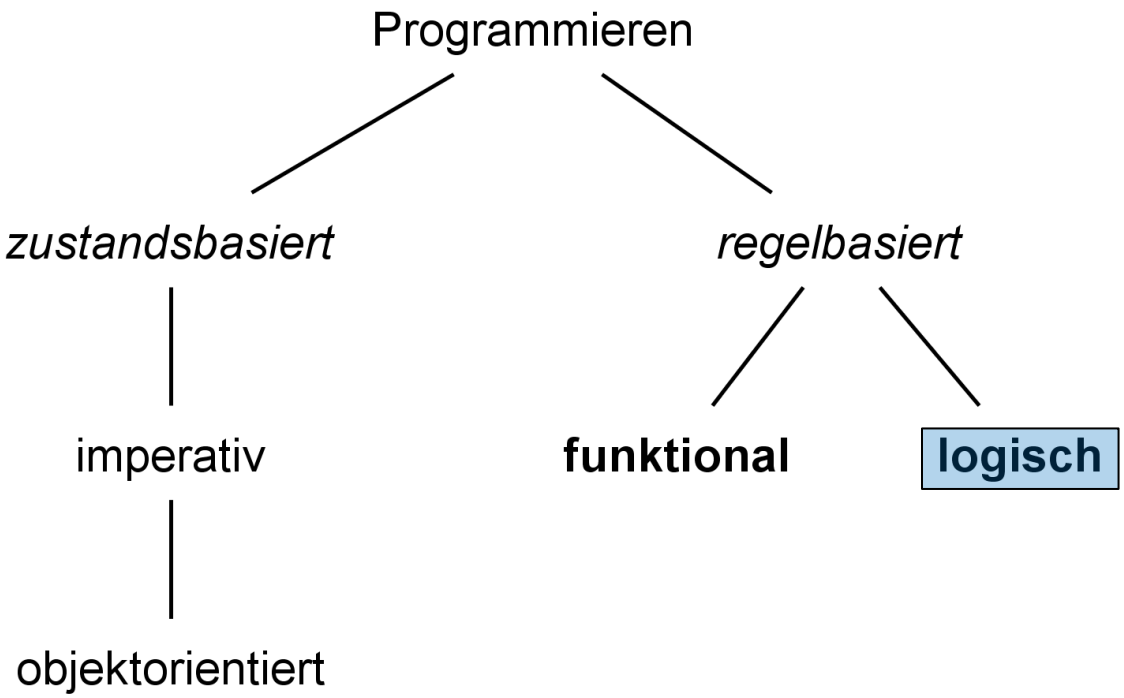


Paradigmen der Programmierung – 11. Deklarative Programmierung mit CHR

Überblick

- Einführung in CHR
- Regeltypen und deren Verhalten
- Logische Variablen und built-in Constraints
- Syntax

Constraint Handling Rules (CHR)

- CHR ist eine *high level* Programmiersprache
- CHR ist ein regelbasiertes Multimengen-Ersetzungssystem
- CHR kann als eigenständige Sprache betrachtet werden oder als Erweiterung einer Host-Sprache (Prolog, Haskell, Java, …)

CHR-Logo

CHR in dieser Veranstaltung

- CHR Ausprobieren in jedem Browser: WebCHR
  http://chr.informatik.uni-ulm.de/~webchr/
- Für Übungen: SWI-Prolog
  (WebCHR Beispielprogramme laufen meistens direkt in SWI-Prolog)
- Vortragsmitschnitt über Anwendungen von CHR
  http://www.youtube.com/watch?v=umjPNWW3EF0

Deklarationen

Einführung von CHR Constraints durch Deklarationen

```
:- module(wetter, [regen/0]).
:- use_module(library(chr)).
:- chr_constraint regen/0, nass/0, schirm/0.
```

- Funktor Notation `c/n`: Name `c`, Anzahl Argumente `n` (Arität, Stelligkeit) des Constraints `c` $(t_1, \ldots, t_n)$
- Zeile 1: optionale Prolog Moduldeklaration: deklariert Modul `wetter`, wobei nur Constraint `regen/0` exportiert wird
- Zeile 2: CHR Bibliothek laden
- Zeile 3: Definiert CHR Constraints `regen`, `nass` und `schirm`
  - Mindestens Namen und Arität müssen angegeben werden
  - Namen beginnen mit einem kleinen Anfangsbuchstaben

Regeln (1/2)

CHR kennt drei Regelarten:

- Simplifizierungsregel: Dinge vereinfachen, Zustandsänderungen, dynamisches Verhalten

```
r @ Kopf <=> Wächter | Rumpf.
```

- Propagierungsregel: Schlussfolgerungen aus existierenden Informationen ziehen

```
r @ Kopf ==> Wächter | Rumpf.
```

- Simpagationsregel: Kombination von Simplifizierung und Propagierung

```
r @ Kopf1 \ Kopf2 <=> Wächter | Rumpf.
```

Regeln (2/2)

```
regel1 @ regen ==> nass.
regel2 @ regen ==> schirm.
```

- Constraints werden nicht entfernt, nur neue hinzugefügt
- Constraints werden im Constraintspeicher gespeichert
- Regel 1: „Wenn es regnet, dann wird es nass.“
- Regel 2: „Wenn es regnet, brauchen wir einen Schirm.“
- Kopf beider Regeln ist `regen`
- Rumpf: `nass` und `schirm`
- Keine Wächter

Anfragen

- Anfragen starten die Berechnung
- Regeln werden erschöpfend angewandt (bis keine Änderungen mehr im Constraintspeicher auftreten)
- Regelanwendungen manipulieren die Anfrage durch Entfernen und Hinzufügen von Constraints
- Ergebnis (genannt Antwort) besteht aus verbliebenen Constraints

```
regel1 @ regen ==> nass.
regel2 @ regen ==> schirm.
```

Stellt man die Anfrage `regen`, erhält man `regen`, `nass`, `schirm` (nicht notwendigerweise in dieser Reihenfolge).

Top-down Ausführung

- Regeln werden in textueller Reihenfolge angewandt
- Im Allgemeinen: Wenn mehr als eine Regel anwendbar ist, wird eine Regel ausgewählt
- Regelanwendungen können nicht wie in Prolog rückgängig gemacht werden (**kein** Backtracking)

  ➔ CHR ist eine committed-choice Sprache

```
regel1 @ regen <=> nass.
regel2 @ regen <=> schirm.
```

- Anwendung der ersten Regel entfernt `regen`
- Zweite Regel wird nie angewandt

Beispiel: Bewegung

- Bewegung ausgedrückt durch Richtungen `ost, west, sued, nord`
- Multiplizität der Schritte ist wichtig, Reihenfolge nicht
- Simplifizierungsregeln drücken aus, dass Schritte sich gegenseitig aufheben können (z.B. `ost` und `west`):
  ```
  ost, west <=> true.
  sued, nord <=> true.
  ```
  Regeln feuern auch, wenn andere Schritte dazwischen sind
- Regeln vereinfachen Bewegung zu minimaler Anzahl Schritte
- Anfrage `ost, sued, west, west, sued, sued, nord, ost, ost` ergibt Antwort `ost, sued, sued`

Logische Variablen

- Bestandteil deklarativer Sprachen wie Prolog und CHR
- Ähnlich zu mathematischen Unbekannten und Variablen in Logik
- Können gebunden oder ungebunden sein
- Gebundene Variablen sind nicht von dem an sie gebundenen Wert unterscheidbar
- Gebundene Variablen können nicht überschrieben werden
- Sprachen mit solchen Variablen heißen *single-assignment* Sprachen
- Andere Sprachen (wie C oder Java) bieten destruktive (mehrfache) Zuweisungen

Beispiel: Paarbildung (1/2)

- Zwei Constraints mit einem Argument repräsentieren Männer (z.B. `mann(joe)`) und Frauen (z.B. `frau(sue)`)
- Zuordnung von Männern und Frauen zum Tanzen mit einer Simplifizierungsregel:
  `mann(X), frau(Y) <=> paar(X,Y).`
- Variablen `X`, `Y` sind Platzhalter für Werte von Constraints bei Regelanwendung
- Gültigkeitsbereich der Variablen ist die Regel in der sie stehen
- Für eine Anfrage mit mehreren Männern und Frauen paart die Regel sie, bis nur noch Personen eines Geschlechts übrig sind

Beispiel: Paarbildung (2/2)

```
mann(X), frau(Y) <=> paar(X,Y).
```

Beispielrechnung:

```
mann(joe), frau(sue), frau(mary), frau(ann), mann(bobby).
paar(joe,sue), frau(mary), frau(ann), mann(bobby).
paar(joe,sue), paar(bobby,mary), frau(ann).
```

Regelarten: Propagierungsregel

- Alle möglichen Paare mit einer Propagierungsregel berechnen (`mann` und `frau` Constraints bleiben erhalten)
  ```
  mann(X), frau(Y) ==> paar(X,Y).
  ```

- Anzahl der Paare ist quadratisch in Anzahl der Personen
  ➔ Propagierungsregeln können teuer sein

Regelarten: Simpagationsregel

- Ein Mann tanzt mit mehreren Frauen, ausgedrückt durch eine Simpagationsregel

```
mann(X) \ frau(Y) <=> paar(X,Y).
```

- Kopf-Constraints links vom Backslash bleiben erhalten, Kopf-Constraints rechts vom Backslash werden entfernt

Built-in Constraints

- CHR kennt zwei Arten von Constraints:
  - CHR Constraints (benutzerdefinierte Constraints - wie alle bisherigen Beispiele)
    → im aktuellen Programm deklariert und durch CHR Regeln definiert
  - Built-in Constraints (built-ins)
    → vordefiniert in Hostsprache oder aus anderen Modulen importierte CHR Constraints
- Nur CHR Constraints im Kopf erlaubt, nur built-in Constraints im Wächter
- Im Rumpf dürfen beide Arten gemischt werden

Beispiel: Fusion und Akquisition (1/2)

- CHR Constraint `firma(Name, Wert)` repräsentiert Firma `Name` mit Marktwert `Wert`
- Größere Firma kauft Firma mit kleinerem Wert, ausgedrückt durch Regel

```
firma(Name1,Wert1), firma(Name2,Wert2) <=>
     Wert1>Wert2 | firma(Name1,Wert1+Wert2).
```

- Wächter `Wert1>Wert2` fungiert als Voraussetzung für Regelanwendung
- Nur built-ins sind im Wächter erlaubt

Beispiel: Fusion und Akquisition (2/2)

- Inline arithmetische Ausdrücke wie `Wert1+Wert2` funktionieren für Hostsprache Java
- In Prolog muss stattdessen is verwendet werden:

```
firma(Name1,Wert1), firma(Name2,Firma2) <=>
     Wert1>Wert2 | Wert is Wert1+Wert2, firma(Name1,Wert).
```

- Regel ist auf jedes Paar von Firmen mit verschiedenem Wert anwendbar

- Nach erschöpfender Anwendung bleiben nur wenige Firmen (mit gleichem Wert)

CHR als Datenbanksprache

CHR als deduktive Datenbank

- CHR kann als Informationsspeicher benutzt werden
- Relationen werden mit CHR Constraints modelliert
- Datenbanktupel sind Instanz eines Constraints
- Anfrage enthält/generiert Tupel der Datenbank als CHR Constraints
- Anfragen, Views, Integritätsconstraints werden als CHR Propagierungsregeln formuliert
  → neue Daten-Constraints (z.B. Datenbanktupel) können gefolgert werden

Beispiel: Verwandtschaftsbeziehungen (1/3)

- Propagierungsregel mit Namen mm drückt Großmuttter-Beziehung aus
  `mm @ mutter(X,Y), mutter(Y,Z) ==> grossmutter(X,Z).`
- Constraint `grossmutter(joe,sue)` bedeutet „Großmutter von Joe ist Sue“
- Erlaubt Ableiten der Großmutter-Beziehung von Mutter-Beziehung
- `mutter(joe,ann), mutter(ann,sue)` propagiert `grossmutter(joe,sue)` mit Regel `mm`

Beispiel: Verwandtschaftsbeziehungen (2/3)

```
mutter(X,Y) ==> elternteil(X,Y).
vater(X,Y) ==> elternteil(X,Y).
elternteil(X,Z), elternteil(Y,Z) ==> geschwister(X,Y).
```

Gegeben

```
mutter(hans,mira), mutter(sepp,mira), vater(sepp,john)
```

Aus den ersten beiden Regeln leitet sich ab:

```
elternteil(hans,mira), elternteil(sepp,mira),
   elternteil(sepp,john)
```

Die letzte Regel fügt hinzu:

```
geschwister(hans,sepp), geschwister(sepp,hans)
```

Zweites Geschwister-Constraint vermeidbar durch die Regel

```
geschwister(X,Y) \ geschwister(Y,X) <=> true.
```

Beispiel: Verwandtschaftsbeziehungen (3/3)

```
mutter(X,Y) ==> elternteil(X,Y).
vater(X,Y) ==> elternteil(X,Y).
elternteil(X,Z), elternteil(Y,Z) ==> geschwister(X,Y).
```

Berücksichtigung von Großeltern, Urgroßeltern usw. durch Vorfahren-Relation

```
elternteil(X,Y) ==> vorfahre(X,Y).
elternteil(X,Y), vorfahre(Y,Z) ==> vorfahre(X,Z).
```

- Erste Regel: „Elternteil ist Vorfahre“
- Zweite Regel: „Vorfahre eines Elternteils ist Vorfahre“
- Vorfahren-Relation ist transitive Hülle der Elternteil-Relation

Syntax

- CHR-spezifischer Teil eines Programms beinhaltet Deklarationen und Regeln
- Deklarationen sind abhängig von der gewählten Implementierung
- Im Folgenden EBNF Grammatik:
  - Terminale in einfachen Anführungszeichen
  - Ausdrücke in eckigen Klammern sind optional
  - Alternativen getrennt durch |

Regeln (1/2)

```
Rule --> [Name '@']
   (SimplificationRule | PropagationRule | SimpagationRule) '.'

SimplificationRule -->
   Head          '<=>' [Guard '|'] Body

PropagationRule    -->
   Head          '==>' [Guard '|'] Body

SimpagationRule    -->
   Head '\' Head '<=>' [Guard '|'] Body
```

- Drei Regelarten in CHR
- '|' trennt Wächter vom Rumpf einer Regel
- '\' teilt Kopf von Simpagationsregeln in zwei Teile auf

Regeln (2/2)

```
Head                 -->   CHRConstraints
Guard                -->   BuiltInConstraints
Body                 -->   Goal

CHRConstraints       -->   CHRConstraint
                         | CHRConstraint ’,’ CHRConstraints
BuiltInConstraints   -->   BuiltIn
                         | BuiltIn ’,’ BuiltInConstraints
Goal                 -->   CHRConstraint | BuiltIn | Goal ’,’ Goal
Query                -->   Goal
```

- Kopf einer Regel ist eine Sequenz von CHR Constraints
- Wächter ist eine Sequenz von built-in Constraints
- Rumpf ist eine Sequenz von built-in und CHR Constraints

Einfache built-in Constraints (1/2)

- Prädikate aus der Host-Sprache Prolog
- Verwendung für Hilfsberechnungen in Regelrumpf
- Built-ins im Wächter normalerweise als Test (Erfolg oder Fehler)
- Die einfachsten built-ins:
  - `true/0` ist immer erfüllt
  - `fail/0` ist nie erfüllt
- Testen, ob Variablen gebunden sind:
  - `var/1` testet, ob Argument eine ungebundene Variable ist
  - `nonvar/1` testet, ob Argument eine gebundene Variable ist

Einfache built-in Constraints (2/2)

- Syntaktische Gleichheit von Ausdrücken (Infix):
  - `=/2` macht Argumente syntaktisch gleich durch Variablenbindung (schlägt fehl, wenn Bindung nicht möglich)
  - `==/2` testet Argumente auf syntaktische Gleichheit
  - `\==/2` testet Argumente auf syntaktische Ungleichheit
- Berechnen und Vergleichen von arithmetischen Ausdrücken (Infix):
  - `is/2` bindet erstes Argument auf den Wert des arithmetischen Ausdrucks im zweiten Argument (schlägt fehl, wenn Bindung nicht möglich)
  - `</2, =</2, >/2, >=/2, =:=/2, =\=/2` testet, ob Argumente arithmetische Ausdrücke sind, deren Wert den Vergleich erfüllt

Informelle Semantik

- Im Folgenden Beschreibung der aktuellen sequentiellen Implementierung
- Basierend auf der sog. *refined operational semantics*
- Es gibt auch experimentelle Implementierungen mit paralleler Regelanwendung
- Alle Implementierungen respektieren die sog. *abstract operational semantics*

Constraints

- Constraints sind sowohl aktive Operationen als auch passive Daten
- Constraints in Zielen werden von links nach rechts abgearbeitet
- Wenn ein CHR Constraint bearbeitet wird:
  - Evaluierung wie ein Prozeduraufruf
  - Überprüfung der Anwendbarkeit von Regeln in denen es auftritt
  - Constraint wird *aktives Constraint* genannt
- Regeln werden in textueller Reihenfolge angewandt (von oben nach unten)
- Wenn keine Regel auf aktives Constraint anwendbar ist, wird es *passiv* und in den Constraint-Speicher abgelegt
- Passive Constraints werden wieder aktiv bei Kontextänderungen (Variablenbindungen)

Matching des Kopfes

- Aktives Constraint matcht gegen ein Kopf-Constraint der Regel
- Matching kann Variablen im Kopf binden (nicht im aktiven Constraint)
- Wenn Matching erfolgreich ist und Regelkopf aus mehr als einem Constraint besteht, wird der Constraint-Speicher nach weiteren Partner-Constraints zum Matchen der verbliebenen Kopf-Constraints durchsucht
- Kopf-Constraints werden von links nach rechts durchsucht
- Ausnahme: Simpagationsregeln
  - zu entfernende Constraints werden zuerst gesucht
- Wenn Matching erfolgreich ist, wird der Wächter geprüft
- Wenn aktives Constraint mehrere Kopf-Constraints matched, wird die Regel für jedes Matching versucht
- Wenn kein erfolgreiches Matching existiert, versucht das aktive Constraint die nächste Regel

Wächterüberprüfung

- Wächter ist Voraussetzung für Regelanwendung
- Test, der entweder gelingt oder fehlschlägt
- Wenn der Wächter erfolgreich ausgewertet wird, wird die Regel angewandt
- Wenn der Wächter fehlschlägt, versucht aktives Constraint nächstes Kopf-Matching

Auswertung des Rumpfes

- Wenn Regel angewandt wird, sagen wir sie *feuert*
- Simplifikationsregel: Match-Constraints werden entfernt, Rumpf ausgewertet
- Simpagationsregel: ähnlich zu Simplifikationsregel, aber Match-Constraints aus dem Kopf-Teil vor \ bleiben erhalten
- Propagierungsregel: Rumpf wird ausgeführt ohne Constraints zu entfernen
- Propagierungsregel feuert nicht nochmal mit den selben Constraints
- Abhängig vom Regeltyp werden Kopf-Constraints entweder *behalten* oder *entfernt*
- Nächste Regel wird versucht, wenn aktives Constraint nicht entfernt wurde

Vergleich: CHR und Prolog

	CHR	Prolog
Objekte	Constraints	Prädikate
Definition der Objekte durch	Regeln	Klausel
Syntax	`kopf <=> wächter | rumpf.`	`kopf :- rumpf.`
#Kopfobjekte	1,2,3,…	1
Anwendungs-bedingung	Matching+Wächter	Unifikation
Versch. Anwendungs-möglichkeiten	Commited-Choice	Backtracking
Keine Anwendungs-möglichkeit	Partielles Ergebnis	Fehler

Überblick: Beispielprogramme

Einfache, kompakte und effiziente CHR-Programme

- Multimengen Transformation
- Größter gemeinsamer Teiler
- Sieb des Eratosthenes
- MergeSort

Multimengen Transformation

- Programme bestehen im Wesentlichen aus einem Constraint
- Constraint repräsentiert aktive Daten
- Paare von Constraints werden durch eine Simplifikationsregel ersetzt
- Oft möglich: kompaktere Notation mit Simpagationsregel
- Simpagationsregel entfernt ein Constraint und behält anderes

Minimum

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

- Berechnet Minimum einer Multimenge von Zahlen n_i analog zum bereits bekannten Maximum-Beispiel
- Zahlen gegeben in Anfrage `min(`n_1`), min(`n_2`), ..., min(`n_k`)`
- `min(`n_i`)` bedeutet n_i ist mögliches Minimum
- Simpagationsregel nimmt zwei `min` Constraints und entfernt das mit dem größeren Wert
- Programm läuft bis nur noch ein `min` Constraint übrig ist
- Dieses `min` Constraint repräsentiert das Minimum

Größter gemeinsamer Teiler (1/4)

Euklidischer Algorithmus:

- *a > b: ggT(a,b) = ggT(a-b,b)*
- sonst: *ggT(a,b) = ggT(a,b-a)*
- Solange, bis ein Element null ist: *ggT(a,0) = ggT(0,a) = a*

```
ggT(N) \ ggT(M) <=> 0<N,N=<M | L is M-N, ggT(L).
```

- Berechnet größten gemeinsamen Teiler von Zahlen dargestellt als `ggT(N)`

- Ergebnis ist das verbleibende `ggT` Constraint mit Wert ungleich 0

Größter gemeinsamer Teiler (2/4)

```
ggT(N) \ ggT(M) <=> 0<N,N=<M | L is M-N, ggT(L).
```

Beispielrechnung:

```
ggT(12), ggT(8)
ggT(8),  ggT(4)
ggT(4),  ggT(4)
ggT(4),  ggT(0)
```

Größter gemeinsamer Teiler (3/4)

- Bedingung `0<N` im Wächter führt zum Ignorieren von `ggT(0)` Constraints
- Useability-Verbesserung durch zusätzliche Regel `ggT(0) <=> true.`
- Bessere Effizienz, wenn Subtraktion durch Modulo-Operation ersetzt wird

```
ggT(N)\ ggT(M) <=> 0<N,N=<M | L is M mod N, ggT(L).
```

Beispielrechnung:

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

Größter gemeinsamer Teiler (4/4)

- Programm funktioniert auch für viele `ggT` Constraints
  → Anfrage `ggT(94017), ggT(1155), ggT(2035)` ergibt `ggT(11)`

- Programm funktioniert auch für rationale Zahlen (mit entsprechenden arithmetischen Operatoren)

- Terminierung garantiert für natürliche Zahlen
  - Neuer Wert ist immer kleiner als M
  - Neuer Wert kann wegen Wächter nicht negativ werden

- Effizient parallel ausführbar

Sieb des Eratosthenes (1/2)

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

- Regel entfernt Vielfache von jeder Nummer
- Anfrage: Primzahlkandidaten von `2` bis `N`
  z.B. `prime(2),prime(3),prime(4), ..., prime(N)`
- Jede Zahl absorbiert Vielfache ihrer selbst, bis nur noch Primzahlen verbleiben

Beispielrechnung:

```
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)
```

Sieb des Eratosthenes (2/2)

- Regel kann als Spezialisierung der Modulo-Version des **ggT** gesehen werden
  → Ergebnis der Modulo-Operation muss 0 sein

- Auch ähnlich zur Minimumsregel
  - Zwei Zahlen werden vergleichen, eine entfernt
  - Aber nicht anwendbar auf beliebige Paare

- Programm terminiert (da nur Constraints entfernt werden, keine neuen erzeugt)

Merging und Sortieren

MergeSort (Wiederholung)

- Idee: Aufteilen der Liste in einzelne Kanten, die sortiert zusammengefügt werden
- Nutzung verketterter Listen

Notation und Repräsentation

- Gerichtete Kante von Knoten `A` zu Knoten `B` repräsentiert durch binäres Constraint `A -> B` (infix)
- Sequenzen repräsentiert durch Kette von Kanten
- Beispiel: Sequenz `0, 2, 5` kodiert als `0 -> 2, 2 -> 5`

Merging (1/3)

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

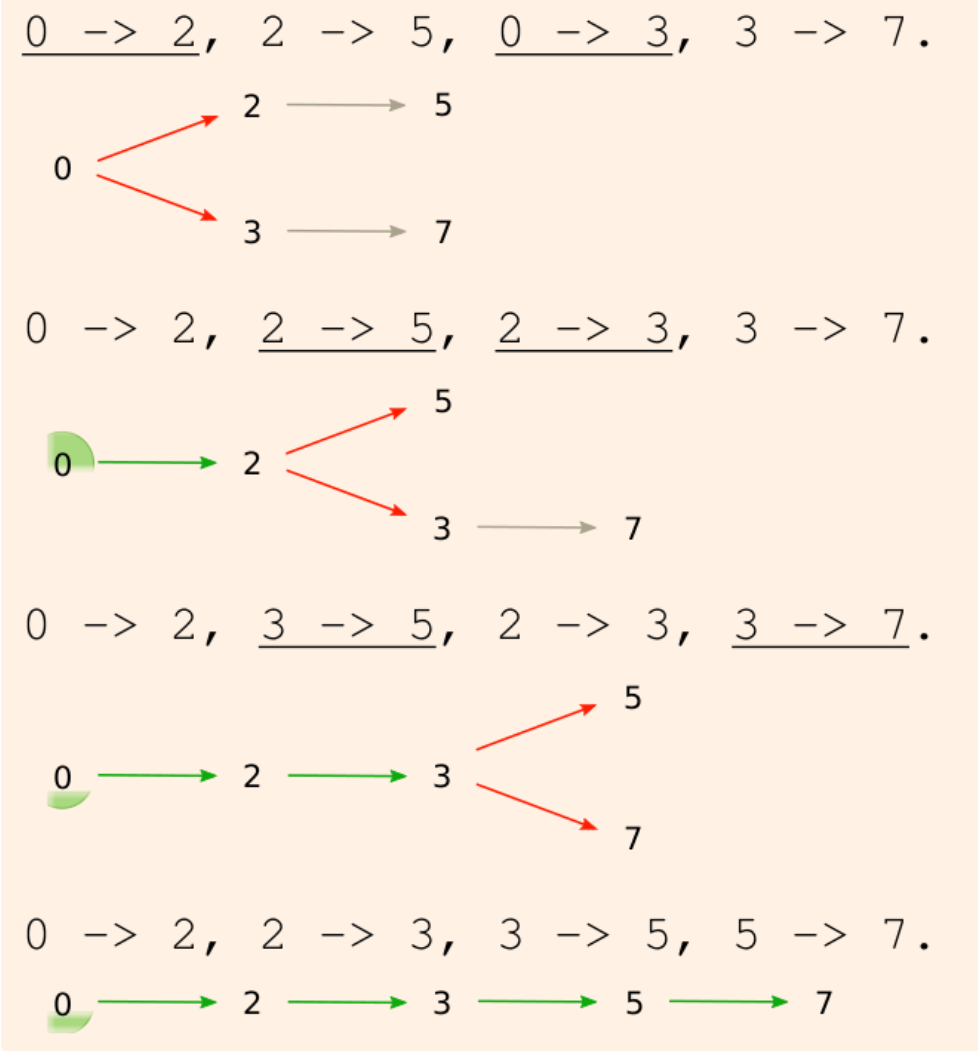
- Annahme: aufsteigend geordnete Ketten (`A -> B` impliziert `A =< B`, `B` ist direkter Nachfolger von `A`)
- Programm verschmilzt Ketten, die mit gleichem kleinsten Wert beginnen
- Gegeben `A -> B` und `A -> C` wird `B -> C` hinzugefügt und die jetzt redundante Kante `A -> C` entfernt
- Anfrage `0 -> 2, 0 -> 5` resultiert in `0 -> 2, 2 -> 5`
- Regel macht transitive Hülle rückgängig

Merging (2/3)

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

- `A` bezeichnet aktuelle Verzweigungsposition
- Alle Knoten bis zu `A` wurden bereits verschmolzen
- Alle Zweige untersucht, kleinerer Nachfolger `B` bleibt erhalten, andere Kante wird ersetzt durch `B -> C`
- Wenn erste Kette nicht beendet, dann verzweigen bei Knoten `B`
- Erschöpfende Regelanwendung entfernt alle solchen Zweige

Merging (3/3)


0 -> 2, 2 -> 5, 0 -> 3, 3 -> 7.
2
5
0
3
7
0 -> 2, 2 -> 5, 2 -> 3, 3 -> 7.
5
0
2
3
7
0 -> 2, 3 -> 5, 2 -> 3, 3 -> 7.
5
0
2
3
7
0 -> 2, 2 -> 3, 3 -> 5, 5 -> 7.
0
2
3
5
7

Sortieren (1/2)

- Mehr als zwei Ketten können gleichzeitig verschmolzen werden
- Menge von Werten repräsentiert als Menge von Kanten der Form `0 -> V`
- Sortieren durch Verschmelzen der Kanten zu einer Kette

Beispiel:

```
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.
```

Sortieren (2/2)

- Programm wandelt folgende bestimmte Arten von Graphen in geordnete Ketten um:
  - Jeder Graph mit geordneten Kanten, bei dem alle Knoten von einer Wurzel aus erreichbar sind, kann sortiert werden
- Quadratische Komplexität für Sortieren (mit Index-Optimierung)
- Optimale $n \cdot log(n)$ Version möglich