ulm university universität ulm

Constraint Handling Rules - Properties of CHR

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

Table of Contents

Overview

- CHR programs automatically implement algorithms with certain properties
  - **Anytime** (approximation): Interrupt program, see approximation of result, restart from this intermediate result
  - **Online** (incremental): Additional constraints can be added during execution without restart
  - **Concurrency**: rules can be applied in parallel to certain parts of goal
- Properties result from operational semantics
  - (Very) abstract semantics used in this chapter
  - Results carry over to refined semantics if program is confluent

Anytime property

- Anytime property
  - No need to restart computation from scratch after interruption
  - Intermediate results approximate final result better and better
  - Can guarantee response times (for embedded systems, for hard problems)
- In CHR
  - When program interrupted, states are meaningful (logical reading)
  - All information needed contained in states
  - Approximation to to answer: more built-ins added and CHR constraints simplified

Example

Example computation (partial order constraint)

$$\underline{A \leq B} \wedge \underline{B \leq C} \wedge C \leq A \mapsto \quad \text{(transitivity)}$$

$$A \leq B \wedge B \leq C \wedge \underline{C \leq A} \wedge \underline{A \leq C} \mapsto \quad \text{(antisymmetry)}$$

$$\underline{A \leq B} \wedge \underline{B \leq C} \wedge A = C \mapsto \quad \text{(antisymmetry)}$$

$$A = B \wedge A = C$$

- Interruption at any point possible
- Proceed the same way as if not interrupted

Limitations

Limitations of current CHR implementations

- No support for interruption and adding constraints
  - But rules can be used to support this behavior
- Usually only logical reading of state observable
  - Refined semantics: propagation history, location of constraints, identifiers and order of CHR constraints lost

Differences regarding semantics

Differences between semantics

- Abstract semantics
  - If logical reading of intermediate state used as initial goal, transitions **Solve** and **Introduce** reconstruct state
  - Now same transitions are possible as from original state
- Refined semantics
  - Transition system may not reconstruct state because constraint order is lost
  - Confluence can still guarantee same answer

Monotonicity property

If transition is possible in state, it is possible in any larger state

- Implies incremental processing of constraints
- Allows for concurrently composed computations

Lemma

If $A \mapsto B$
then $A \wedge D \mapsto B \wedge D$

(for states A, B, and D)

Example computation (partial order constraint)

$\underline{A \leq B} \wedge \underline{B \leq C} \mapsto$ (transitivity)

$A \leq B \wedge B \leq C \wedge \underline{A \leq C} \wedge \underline{C \leq A}$ (added) $\mapsto$ (antisymmetry)

$A \leq B \wedge B \leq C \wedge \underline{A = C} \mapsto$

$\ldots$

Monotonicity property

Lemma can be restated for abstract semantics

- In nonfailed state:
  - can add satisfiable constraints and global variables
  - can remove entries from propagation history

Lemma

Nonfailed ω_t states (no shared local variables, $B \wedge B'$ consistent)

$$A = \langle G, S, B, T \rangle_n^{\mathcal{V}} \text{ and } D = \langle G', S', B', T' \rangle_m^{\mathcal{V}'},$$

Then the combined state $A \wedge D = \langle G \uplus G'_{+n}, S \cup S'_{+n}, B \wedge B', T'' \rangle_{n+m}^{\mathcal{V} \cup \mathcal{V}' \cup \mathcal{V}''}$ admits same transitions as state A (and D).

($+n$ increments identifier, $T'' \subset T \cup T'_{+n}$, $\mathcal{V}''$ arbitrary variables)

In refined semantics two different ways of combining states (order matters in stack)

Online property

- Online algorithm property
  - Adding constraints during execution without restart from scratch
  - Same behavior as if constraints were there from beginning
  - Useful for interactive, reactive, control systems, agent programming

Concurrency and parallelism (I)

- **Concurrency** allows for logically independent computations
- **Parallelism** allows for computations that happen simultaneously
- Concurrency can be implemented in sequential or parallel way
  - Parallel implementations require suitable hardware
  - Sequential implementations use interleaving of computations
- **Interleaving semantics** of concurrency
  - For each parallel computation exists a sequential interleaving with same result
  - Parallel execution can be simulated by sequential one

Concurrency and parallelism (II)

- Declarative programming languages are compositional
  - Different computations can be composed without interference
  - Techniques for programming, reasoning, analysis still apply for composed program
  - Makes concurrency and distribution easier

Example

Destructive assignments `x:=5` and `x:=7` versus equations `x=5` and `x=7`

- In imperative languages: unpredictable result in parallel execution (`x=5` or `x=7`)
- In concurrent constraint languages: unintended interference leads to failure (noticeable)

Processes

- Main notion in concurrent and distributed programming
- Programs that are executed independently but interact (concurrent programs)
- Can communicate and synchronize by sending and receiving messages
- Can build a dynamically changing process network

Concurrency in CHR

- Concurrent processes are CHR constraints
  - Communication via shared built-in constraint store
  - Built-in constraints: (partial) messages
  - Shared variables: communication channels
  - Communication usually asynchronous
- Running processes: active CHR constraints
  - check/ask (in guard) and place/tell (in body) built-in constraints on shared variables

Parallelism in CHR

- Rules needed such that for parallel $A \overset{\mapsto}{\mapsto} B$ execution exists sequentialized $A \mapsto^{+} B$ execution
- Parallelization without change for confluent programs

Weak and strong parallelism in CHR

- **Weak**: rules can be applied to separate parts of the problem in parallel
  - Due to monotonicity: Adding constraints cannot inhibit rule applicability
- **Strong**: rules can be applied to overlapping parts of problem too
  - Overlap must not be removed

Weak Parallelism (I)

Weak parallelism: rule application to separate parts of problem

Definition (Weak parallelism)

If $A \mapsto B$

and $C \mapsto D$

then $A \wedge C \overset{\mapsto}{\mapsto} B \wedge D$

(A, B, C, D conjunction of constraints, $\overset{\mapsto}{\mapsto}$ parallel transition relation)

Weak parallelism (II)

Theorem (Trivial confluence)

If $A \mapsto B$

and $C \mapsto D$

then $A \wedge C \mapsto S \mapsto B \wedge D$

where S is either $A \wedge D$ or $B \wedge C$

- Two interpretations
  - Rule applications on separate parts of goal can be exchanged
  - Rule applications from different goals can be composed
- Holds for (very) abstract semantics
- Constraints can only be involved in one rule application simultaneously

Weak parallelism (III)

Example (Minimum)

$\texttt{min(N)} \setminus \texttt{min(M)} \Leftrightarrow \texttt{N} \leq \texttt{M} \mid \mathit{true}.$

Rule of `min` can be applied in parallel to different parts of query

$$\underline{\texttt{min(1)}} \wedge \underline{\texttt{min(0)}} \wedge \underline{\texttt{min(2)}} \wedge \underline{\texttt{min(3)}} \overset{\mapsto}{\mapsto}$$
$$\underline{\texttt{min(0)}} \wedge \underline{\texttt{min(2)}} \mapsto$$
$$\texttt{min(0)}$$

One rule instance applied to $\texttt{min(1)} \wedge \texttt{min(0)}$,
the other to
$\texttt{min(2)} \wedge \texttt{min(3)}$

Weak parallelism (IV)

Example (Partial order constraint)

duplicate @ X≤Y ∧ X≤Y ⇔ X≤Y.
reflexivity @ X≤X ⇔ *true*.
antisymmetry @ X≤Y ∧ Y≤X ⇔ X=Y.
transitivity @ X≤Y ∧ Y≤Z ⇒ X≤Z.

Rules antisymmetry and transitivity can be applied in parallel in

$$\underline{A \leq B} \wedge \underline{C \leq A} \wedge \underline{B \leq C} \wedge \underline{B \leq A} \overset{\mapsto}{\mapsto} A = B \wedge C \leq A \wedge B \leq C \wedge B \leq A$$

Antisymmetry rule applied to $A \leq B \wedge B \leq A$,
transitivity rule to $B \leq C \wedge C \leq A$

Shortcomings of weak parallelism

Weak parallelism is too strict

- Application of propagation rules to overlapping parts of state not allowed
- Sharing built-ins for guard checks of parallel applications not allowed

Strong parallelism (I)

Parallel rule applications to overlapping parts allowed if overlap kept

Definition (Strong parallelism)

If	$A \wedge E$	$\mapsto$	$B \wedge E$
and	$C \wedge E$	$\mapsto$	$D \wedge E$
then	$A \wedge E \wedge C$	$\overset{\mapsto}{\mapsto}$	$B \wedge E \wedge D$

- With weak parallelism either two copies of E needed or E must have been empty

Strong parallelism (II)

Theorem (Trivial confluence with context)

If $A \wedge E \mapsto B \wedge E$

and $C \wedge E \mapsto D \wedge E$

then $A \wedge C \wedge E \mapsto S \mapsto B \wedge D \wedge E$

where S is either $A \wedge D \wedge E$ or $B \wedge C \wedge E$

- CHR constraint can be used several times if kept in all rule matchings
- CHR constraint can be used only once if it is removed

Strong parallelism (III)

- *Propagation rules*: only add constraints, any constraint can be in overlap
  - Several propagation rules can be applied simultaneously
- *Simpagation rules*: Constraint which are not removed allowed in overlap
  - Simpagation rule may remove arbitrary number of constraints simultaneously in one concurrent step
- *Simplification rules*: Remove all constraints, cannot be involved in overlap except for built-ins

Strong parallelism (IV)

Example (Parallel computation of minimum)

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

`min(1)` matches kept head constraint of simpagation rule

`min(2)` and `min(3)` match in parallel two different instances of head constraint to be removed

$$\underline{\texttt{min(1)}} \wedge \texttt{min(0)} \wedge \underline{\texttt{min(2)}} \wedge \underline{\texttt{min(3)}} \overset{\mapsto}{\mapsto}$$
$$\underline{\texttt{min(1)}} \wedge \underline{\texttt{min(0)}} \mapsto$$
$$\texttt{min(0)}$$

Choosing `min(0)` as kept constraint would have computed result in just one step

Strong parallelism (V)

Example (Parallel computation of partial order constraint)

Apply transitivity rule, then antisymmetry rule, each three times in parallel:

$$\underline{A \leq B} \wedge \underline{B \leq C} \wedge \underline{C \leq A} \overset{\mapsto}{\mapsto}$$

$$\underline{A \leq B} \wedge \underline{B \leq C} \wedge \underline{C \leq A} \wedge \underline{A \leq C} \wedge \underline{B \leq A} \wedge \underline{C \leq B} \overset{\mapsto}{\mapsto}$$

$$A = B \wedge B = C \wedge A = C$$

First transition only possible with strong parallelism (several propagation rules applied to same constraint simultaneously)

Stronger parallelism

- Even stronger parallelism (removing constraints from overlap) may lead to incorrect behavior
- Example:
  - Rule `min(N) \ min(M)` $\Leftrightarrow$ `N`$\leq$`M` $\mid$ *true*
  - Query `min(1)`$\wedge$`min(1)`
  - One rule tries matching in given, one in reversed order
    $\Rightarrow$ Both constraints will be removed (not correct)
- This behavior is not allowed by weak or strong parallelism
- Still, even stronger parallelism is possible...

Implementation of parallelism

Locking mechanism for atomic CHR constraints

- *Weak parallelism*
  - On rule application lock all matched constraints
  - If one fails, unlock all, try redoing rule application
  - If all constraints locked successfully, apply rule
  - Unlock kept constraints before executing rule body
- *Strong parallelism*
  - Analogous, but only removed head constraints locked
  - No unlocking necessary (locked constraints removed)

Use standard algorithms to avoid deadlocks and cyclic behavior

Programs under refined semantics(I)

- Parallelization of every program in (very) abstract semantics possible
- Incorrect results under refined semantics possible
  - Order of rules in program and constraints in goal
- Trivial confluence does not hold in in refined semantics
  - Combination of states is not symmetric
- Solution: make programs confluent
  - Order of rule application and constraints in goal does not matter
  - $\Rightarrow$ Can always be run in parallel

Programs under refined semantics(II)

Example (Destructive assignment in parallel)

Destructive assignment rule:

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

Query:

```
cell(x,2), assign(x,1), assign(x,3)
```

Undetermined which update through `assign` comes first

Computational power and expressiveness

- **CHR machine**: fragment of CHR language (analogous to RAM or Turing machines)
- CHR, RAM, and Turing machine simulate each other in polynomial time
  - $\Rightarrow$ CHR is Turing complete
  - $\Rightarrow$ CHR can implement every algorithm without performance penalty
    (Not known for other purely declarative paradigms)
  - $\Rightarrow$ CHR first declarative language with “optimal complexity”

Sufficiently strong constraint theory

CHR machine needs sufficiently strong constraint theory

Definition (Sufficiently strong constraint theory)

$\mathcal{CT}$ is sufficiently strong if it defines at least the built-ins $true, false, =, \neq$ and the arithmetic operations $+, -, *, /$ over integers

CHR machine (I)

CHR program to simulate RAM machine

- Memory cell represented by `m(A,V)` (address `A`, value `V`)
- Program counter pointing to code line `L` represented by `c(L)`
- Instruction `i(L,L1,I,D1,D2)` consists of
  - `L`, `L1` current and new line number
  - Name of instruction `I`
  - Arguments of instruction `D1`, `D2`

CHR machine (II)

Typical rules of the Ram machine simulation in CHR

add

```
i(L,L1,add,B,A), m(B,Y) \ m(A,X), c(L)
                 <=> Z is X+Y, m(A,Z), c(L1).
```

move

```
i(L,L1,move,B,A), m(B,X) \ m(A,Y), c(L)
                  <=> m(A,X), c(L1).
```

jump

```
i(L,L1,jump,A) \ c(L) <=> c(A).
```

halt

```
i(L,L1,halt) \ c(L) <=> true.
```

CHR machine (III)

- **Query**: instructions, memory cells, line number of first instruction
- Instruction at line `L` is executed, updating memory `m` and program counter
- Essential aspect: Destructive assignment effectively simulated

CHR machine (IV)

Theorem

Given a sufficiently strong constraint theory $\mathcal{CT}$, there exists a CHR (machine) program which can simulate in $O(T + P + S)$ time and $O(P + S)$ space a T-time, S-space RAM machine with a program of P lines.

CHR machine (V)

Effectively realized in optimized CHR implementation of K.U. Leuven

Theorem

For every (RAM machine) algorithm which uses at least as much time as it uses space, a CHR program exists which can be executed in the K.U. Leuven CHR system, with time and space complexity within a constant factor from the original complexities.

Summary

- Questions still left
  - Can algorithms be expressed naturally and elegantly?
  - How big are the constant factors?
- Experience with classical algorithms shows conciseness and elegance
- Empirical evidence shows
  - CHR is faster than other rule-based languages
  - constant factor in comparison to low-level imperative languages
  - Slow-down over C is within an order of magnitude now