Guard and Continuation Optimization for Occurrence Representations of Constraint Handling Rules

ICLP 2005 presentation

Jon Sneyers, Tom Schrijvers, Bart Demoen
{jon,toms,bmd}@cs.kuleuven.be

Department of Computer Science
K.U.Leuven, Belgium

Overview

1. Constraint Handling Rules
2. Occurrence Representation
3. Guard Optimization
4. Continuation Optimization
5. Type and mode information
6. Experiments
7. Future work

1. Constraint Handling Rules

(Thom Frühwirth)

- Write your own constraint solver as CHRs:
  - ♦ application tailored solvers
  - ♦ embedded in Prolog (or other host language)

    $\Rightarrow$ no interfacing problems
  - ♦ high-level specification
    - focus on what, not how
    - fixpoint computation is taken care of
    - very compact programs
    - easy to understand, modify and experiment with
- Also useful as general-purpose language: everything can be implemented in CHR with optimal complexity (cfr our CHR workshop paper on computability and complexity of CHR)

1. Constraint Handling Rules

Three kinds of CHR rules:

- Simplification rules:

  RemovedHeads <=> Guard | Body.

- Propagation rules:

  KeptHeads ==> Guard | Body.

- Simpagation rules:

  KeptHeads \ RemovedHeads <=> Guard | Body.

A rule can be applied if: - head constraints are in the constraint store
- the guard is satisfied

1. CHR example: interval solver

```
:- constraints in/2.
X in A:B <=> A>B | fail.
X in A:B <=> A =:= B | X is A.
X in A:B, X in C:D <=> A<B, C<D | X in max(A,C):min(B,D).
```

- First rule: X is in an empty interval $\Leftrightarrow$ fail
- Second rule: X is in $[a, a] \Leftrightarrow$ X is a
- Third rule: interval intersection

1. CHR: Operational semantics

- abstract (theoretical) operational semantics ω_t
  - rules can be applied in any order
- refined oper. sem. ω_r [Duck, Stuckey, de la Banda, Holzbaur]
  - an instance of ω_t
  - rules applied in textual order $\rightarrow$ more deterministic
  - better termination/complexity possible
  - used in all major CHR implementations

1. CHR: Operational semantics

Implementations use refined operational semantics:

```
a \  b <=> true.
a ,  a <=> b .
b ==> a.
a ,   c <=> d.
```

Query:

```
?- b,     c.
```

Constraint store:

1. CHR: Operational semantics

Implementations use refined operational semantics:

```
a \  b <=> true.
a ,  a <=> b .
b ==> a.
a ,   c <=> d.
```

Query:

```
?- b,     c.
```

Constraint store:

b

1. CHR: Operational semantics

Implementations use refined operational semantics:

```
a \  b <=> true.
a ,  a <=> b .
b ==> a.
a ,   c <=> d.
```

Query:

```
?- b,     c.
```

Constraint store:

b

1. CHR: Operational semantics

Implementations use refined operational semantics:

```
a \  b <=> true.
a ,  a <=> b .
b ==> a.
a ,   c <=> d.
```

Query:

```
?- b,     c.
```

Constraint store:

b

1. CHR: Operational semantics

Implementations use refined operational semantics:

```
a \  b <=> true.
a ,  a <=> b .
b ==> a.
a ,   c <=> d.
```

Query:

```
?- b,     c.
```

Constraint store:

b, a

1. CHR: Operational semantics

Implementations use refined operational semantics:

```
a \  b <=> true.
a ,  a <=> b .
b ==> a.
a ,   c <=> d.
```

Query:

```
?- b,     c.
```

Constraint store:

b, a

1. CHR: Operational semantics

Implementations use refined operational semantics:

```
a \  b <=> true.
a ,  a <=> b .
b ==> a.
a ,   c <=> d.
```

Query:

```
?- b,     c.
```

Constraint store:

a

1. CHR: Operational semantics

Implementations use refined operational semantics:

```
a \  b <=> true.
a ,  a <=> b .
b ==> a.
a ,   c <=> d.
```

Query:

```
?- b,     c.
```

Constraint store:

a

1. CHR: Operational semantics

Implementations use refined operational semantics:

```
a \  b <=> true.
a ,  a <=> b .
b ==> a.
a ,   c <=> d.
```

Query:

```
?- b,     c.
```

Constraint store:

a

1. CHR: Operational semantics

Implementations use refined operational semantics:

```
a \  b <=> true.
a ,  a <=> b .
b ==> a.
a ,   c <=> d.
```

Query:

```
?- b,     c.
```

Constraint store:

a

1. CHR: Operational semantics

Implementations use refined operational semantics:

```
a \  b <=> true.
a ,  a <=> b .
b ==> a.
a ,   c <=> d.
```

Query:

```
?-  b,  c.
```

Constraint store:

a, c

1. CHR: Operational semantics

Implementations use refined operational semantics:

```
a \  b <=> true.
a ,  a <=> b .
b ==> a.
a ,   c <=> d.
```

Query:

```
?-  b,  c.
```

Constraint store:

a, c

1. CHR: Operational semantics

Implementations use refined operational semantics:

```
a \  b <=> true.
a ,  a <=> b .
b ==> a.
a ,   c <=> d.
```

Query:

```
?-  b,  c.
```

Constraint store:

a, c

1. CHR: Operational semantics

Implementations use refined operational semantics:

```
a \ b <=> true.
a , a <=> b .
b ==> a.
a , c <=> d.
```

Query:

```
?- b, c.
```

Constraint store:

d

1. CHR: Operational semantics

Implementations use refined operational semantics:

```
a \  b <=> true.
a ,  a <=> b .
b ==> a.
a ,   c <=> d.
```

Query:

```
?-  b, c.

d
Yes
```

Constraint store:

1. CHR: Operational semantics

Implementations use refined operational semantics:

```
a \  b <=> true.
a ,  a #X  <=> b  pragma passive(X) .
b ==> a.
a ,   c <=> d.
```

Query:

```
?-  b, c.

d
Yes
```

Constraint store:

1. CHR: Operational semantics

- In refined operational semantics ω_r, rule order is implicit guard:
  X leq X <=> true.
  X leq Y, Y leq X <=> X \== Y | X=Y.
  X leq Y, Y leq Z ==> X \== Y, Y \== Z, X \== Z | X leq Z.
- Programmers remove implied guards to improve performance
- Problem for logical reading of a rule: meaning of a rule depends on all previous rules!
- Solution: keep all necessary guards in program, let compiler remove redundant guards

2. Occurrence Representation

Occurrence representation:

- Defined in terms of *occurrences*, not *rules*
- Every occurrence has…
  - partner constraints
  - guard (default: rule guard)
  - body (default: rule body)
  - success continuation (if occ is not removed) (default: next occ)
  - fail continuation (default: next occ)

Call-based refined **operational semantics**

for occurrence representations (ω_o)

2. Occurrence Representation

- Very close to how CHR rules are compiled
- Mapping CHR program $\rightarrow$ Occ. Repr. such that $\omega_r = \omega_o$
- Can be used to formulate (and show correctness of) optimizations
- Skipping occurrences (pragma passive) can be done by changing continuations

3. Guard Optimization: Idea

```
X in A:B <=> A>B | fail.
X in A:B <=> A =:= B | X is A.
X in A:B, X in C:D <=>  A<B, C<D | X in max(A,C):min(B,D).
```

- When last rule is tried, first two rules did not fire
  (otherwise active constraint was removed)
  $\Rightarrow$ guards of first two rules failed for both constraints

3. Guard Optimization: Idea

```
X in A:B <=> A>B | fail.
X in A:B <=> A =:= B | X is A.
X in A:B, X in C:D <=>  A<B, C<D | X in max(A,C):min(B,D).
```

- When last rule is tried, first two rules did not fire
  (otherwise active constraint was removed)
  $\Rightarrow$ guards of first two rules failed for both constraints
- negations of first guard: A =< B and C =< D
  negations of second guard: A =\= B and C =\= D
  $\Rightarrow$ this entails A < B and C < D
  $\Rightarrow$ guard of 3rd and 4th occurrence can be simplified

3. Guard Optimization: Idea

```
X in A:B <=> A>B | fail.
X in A:B <=> A =:= B | X is A.
X in A:B, X in C:D <=>   X in max(A,C):min(B,D).
```

- When last rule is tried, first two rules did not fire
  (otherwise active constraint was removed)
  $\Rightarrow$ guards of first two rules failed for both constraints
- negations of first guard: A =< B and C =< D
  negations of second guard: A =\= B and C =\= D
  $\Rightarrow$ this entails A < B and C < D
  $\Rightarrow$ guard of 3rd and 4th occurrence can be simplified

3. Guard Optimization

X in_1 A:B <=> A>B | . . .
X in_2 A:B <=> A =:= B | . . .
X in_3 A:B, X in_4 C:D <=> A<B, C<D | . . .

- New guard for the 3rd occurrence: true
- New guard for the 4th occurrence: fail

X in_3 A:B, X in_4#Id C:D <=> true | . . . pragma passive(Id)
X in_3#Id A:B, X in_4 C:D <=> fail | . . . pragma passive(Id)

3. Guard Optimization

- Never-stored constraints are removed before they actually *need* to be inserted
  $\rightsquigarrow$ avoid constraint store insert/remove overhead
- Currently detects only single-head no-guard simplification rules, e.g. X geq Y <=> Y leq X.
- Do guard optimization first:
  - X geq X <=> true.
    X geq Y <=> X \== Y | Y leq X.
  - sign(X,S) <=> X>0 | S=pos.
    sign(X,S) <=> X =:= 0 | S=zero.
    sign(X,S) <=> X<0 | S=neg.

3. Guard Optimization

- Never-stored constraints are removed before they actually *need* to be inserted
  $\rightsquigarrow$ avoid constraint store insert/remove overhead
- Currently detects only single-head no-guard simplification rules, e.g. X geq Y <=> Y leq X.
- Do guard optimization first:
  - X geq X <=> true.
    X geq Y <=> Y leq X. $\rightsquigarrow$ geq/2 is never-stored!
  - sign(X,S) <=> X>0 | S=pos.
    sign(X,S) <=> X =:= 0 | S=zero.
    sign(X,S) <=> S=neg. $\rightsquigarrow$ sign/2 is never-stored!

4. Continuation Optimization

- Idea: skip occurrences that cannot result in rule application
- Guard optimization can simplify an occ guard to fail
  $\rightsquigarrow$ modify continuation of previous occurrence
- Always failing occ guard = pragma passive(occ) = continuations of occ-1 are occ+1

4. Continuation Optimization

X in_1 A:B <=> A>B | ...
X in_2 A:B <=> A =:= B | ...
X in_3 A:B, X_4 in C:D <=> A<B, C<D | ...

- Removed occurrences (e.g. simplification rules): only fail continuation is relevant
- Recall: after guard optimization: $g'(\mathsf{X}\ \mathsf{in}_4\ \mathsf{C} : \mathsf{D}) = \mathsf{fail}$
- Optimizing $n_f(\mathsf{in}_3) = \mathsf{in}_4$ results in $n'_f(\mathsf{in}_3) = \mathsf{in}_5$ (skipping 4th occurrence)

4. Continuation Optimization

fib_1(0,M) ==> M = 1.
fib_2(1,M) ==> M = 1.
fib_3(N,M) ==> N > 1 | fib(N-1,M1), fib(N-2,M2), M is M1+M2.

- Occurrences in Occ^k (e.g. propagation rules):
  fail *and* success continuations are relevant
  current subderivation has to continue whether or not the rule was applied
- Optimizing $n_s(\mathsf{fib}_1) = \mathsf{fib}_2$ results in $n'_s(\mathsf{fib}_1) = \mathsf{fib}_4$
  (skipping 2nd and 3rd occurrence)
- Optimizing $n_s(\mathsf{fib}_2) = \mathsf{fib}_3$ results in $n'_s(\mathsf{fib}_2) = \mathsf{fib}_4$
- Fail continuations cannot be optimized in this example

5. Type and mode information

- Type and mode information can be used to improve optimizations
- Optional type/mode declarations:
  :- chr_type list(T) —> [] ; [T | list(T)].
  :- constraints sum(+list(int), ?int).
- Mode: + for ground arguments, ? for unknown mode.
- Type: built-in types like any, int, natural, …
  new types can be defined using (generic) type definitions
- Backwards-compatibility:
  :- constraints sum/2 is now syntactic sugar for
  :- constraints sum(?any,?any).

5. Type and mode information

- Can use hash-table store for ground constraints!
- Mode/type also useful in guard/continuation optimization
- E.g. if first argument is ground list:

```
sum([],S) <=> S=0.
sum( [X|Xs] ,S) <=>  sum(Xs,T), S is X+T.
```

5. Type and mode information

- Can use hash-table store for ground constraints!
- Mode/type also useful in guard/continuation optimization
- E.g. if first argument is ground list:
  sum([],S) <=> S=0.
  sum(L ,S) <=> L=[X|Xs], sum(Xs,T), S is X+T.
- $\rightsquigarrow$ sum/2 is never-stored!
- Generated code: ...

5. Effect of Optimizations

```
:-use_module(library(chr_runtime)). :-use_module(library(chr_hashtable_store)). 'attach_sum/2'([],_). 'attach_sum/2'([E|D],C):-(get_attr(E,user,B)->A
=[C|B],put_attr(E,user,A);put_attr(E,user,[C])),'attach_sum/2'(D,C). 'detach_sum/2'([],_). 'detach_sum/2'([E|D],C):-(get_attr(E,user,B)->chr_runtime:
sbag_del_element(B,C,A),(A==[]->del_attr(E,user);put_attr(E,user,A));true),'detach_sum/2'(D,C). '$indexed_variables'(C,B):-C=sum(A,_),term_variables(
A,B). attach_increment([],_). attach_increment([F|E],D):-chr_runtime:not_locked(F),(get_attr(F,user,C)->sort(C,B),chr_runtime:merge_attributes(D,B,A)
,put_attr(F,user,A);put_attr(F,user,D)),attach_increment(E,D). attr_unify_hook(G,F):-sort(G,E),(var(F)->(get_attr(F,user,D)->true;D=[]),sort(D,C),
chr_runtime:merge_attributes(E,C,B),put_attr(F,user,B),chr_runtime:run_suspensions(B);(compound(F)->term_variables(F,A),attach_increment(A,E);true),
chr_runtime:run_suspensions(G)). activate_constraint(H,G,F,E):-arg(2,F,D),D=mutable(C),chr_runtime:update_mutable(active,D),(nonvar(E)->true;arg(4,F,
B),B=mutable(A),E is A+1,chr_runtime:update_mutable(E,B)),(compound(C)->term_variables(C,G),chr_runtime:none_locked(G),H=yes;C==removed->
chr_indexed_variables(F,G),H=yes;G=[],H=no). remove_constraint_internal(E,D,C):-arg(2,E,B),B=mutable(A),chr_runtime:update_mutable(removed,B),(
compound(A)->D=[],C=no;A==removed->D=[],C=no;C=yes,chr_indexed_variables(E,D)). insert_constraint_internal(yes,J,I,H,G,F):-I=..[suspension,E,D,H,C,B,
G|F],chr_indexed_variables(I,J),chr_runtime:none_locked(J),chr_runtime:create_mutable(active,D),chr_runtime:create_mutable(0,C),chr_runtime:
create_mutable(A,B),chr_runtime:empty_history(A),chr_runtime:gen_id(E). chr_indexed_variables(C,B):-C=..[_,_,_,_,_,_,A|_],'$indexed_variables'(A,B).
'$insert_in_store_sum/2'(D):-chr_runtime:global_term_ref_1(C),(get_attr(C,user,B)->A=[D|B],put_attr(C,user,A);put_attr(C,user,[D])).
'$delete_from_store_sum/2'(D):-chr_runtime:global_term_ref_1(C),(get_attr(C,user,B)->chr_runtime:sbag_del_element(B,D,A),(A==[]->del_attr(C,user);
put_attr(C,user,A));true). '$enumerate_suspensions'(C):-chr_runtime:global_term_ref_1(B),get_attr(B,user,A),chr_runtime:sbag_member(C,A). sum(B,A):-
'sum/2__0'(B,A,_). 'sum/2__0'(E,D,C):-E==[],!,(var(C)->true;remove_constraint_internal(C,B,A),(A==yes->'$delete_from_store_sum/2'(C),'detach_sum/2'(B
,C);true)),D=0. 'sum/2__0'(H,G,F):-nonvar(H),H=[E|D],!,(var(F)->true;remove_constraint_internal(F,C,B),(B==yes->'$delete_from_store_sum/2'(F),
'detach_sum/2'(C,F);true)),sum(D,A),G is E+A. 'sum/2__0'(E,D,C):-(var(C)->insert_constraint_internal(B,A,C,user:'sum/2__0'(E,D,C),sum(E,D),[E,D]);
activate_constraint(B,A,C,_)),(B==yes->'$insert_in_store_sum/2'(C),'attach_sum/2'(A,C);true).
```

```
:- use_module(library(chr_runtime)).
:- use_module(library(chr_hashtable_store)).
'$enumerate_suspensions'(_) :- fail.
sum([],A) :- !, A=0.
sum([D|C],B) :- sum(C,A), B is D+A.
```

6. Experiments

Never-stored constraints: (guard optimization)

Benchmark	*Optimize*	*# clauses*	*# lines*	*Runtime (%)*	
sum	no	3	10	5.03	(100)
(10000,500)	type	2	6	4.49	(89)
nrev	no	6	20	13.97	(100)
(30,50000)	type	4	11	8.44	(60)
dfsearch	no	4	16	37.58	(100)
(16,500)	yes	4	15	31.63	(84)
	type	3	11	29.97	(80)

6. Experiments

"Real" constraint solvers: (continuation optimization)

Benchmark	*Optimize*	*# clauses*	*# lines*	*Runtime (%)*	
bool_chain	no	180	2861	12.8	(100)
(200)	yes	147	2463	7.0	(55)
fib	no	10	154	11.2	(100)
(22)	yes	9	125	8.5	(76)
leq	no	18	218	14.1	(100)
(60)	yes	13	162	11.7	(83)

7. Future work

- Stronger simplification:
  now: conditions are only simplified to `true` or `fail`
  better: also simplify expensive conditions to cheap conditions
- Specialize generated code to call patterns in rule bodies
- Type-checking
- New types for more efficient constraint stores
  e.g. `dense_int` for array stores
- Automatic type/mode inference?
- Based on query pattern declarations?