Chapter 10: CLP Systems

Where we examine how CLP systems work and introduce an important concept for constraint solvers: incrementality

1

CLP Systems

- Simple Backtracking Goal Evaluation
- Incremental Constraint Solving
- Efficient Saving and Restoring of the Constraint Store
- Implementing If-Then-Else, Once and Negation
- Optimization
- Other Incremental Constraint Solvers

2

Backtracking Goal Evaln.

- Previously understood as depth-first left-right search through a derivation tree
- Specific algorithm: simple_solve_goal
  - parametric in *solv* and *simpl*
  - uses *defn(P,L)* which returns rules defining *L* in program *P* in the order they occur, renamed to not contain any previous variables
- simple_solve_goal(*G*)
  - **return** *simpl*(*vars*(*G*) , simple_backtrack(<*G*|*true*>})

3

simple_backtrack

- simple_backtrack(<G|C>)
  - **if** *G* is empty **return** *C*
  - **let** *G* be of the form *L, G’*
  - **case** *L* is a primitive constraint
    - if *solv*(*C* /\ *L*) = *false* **return** *false*
    - **return** simple_backtrack(<*G’*|*C* /\ *L*>)
  - **case** *L* is an atom *p(s1,...,sn)*
    - **foreach** *p(t1,...,tn)* :- *B* in *defn(P,L)*
      - *C1* = simple_backtrack(<*s1=t1,..,sn=tn,B,G’*|*C*>)
      - if *C1* != *false* **return** *C1*
    - **return** false

4

Example execution sum(1,S)

```
(S1) sum(0,0).
(S2) sum(N,N+S) :- sum(N-1,S).
```

simple_backtrack(<*sum(1,S)* | *true*>)

simple_backtrack(<*1=0,S=0* | *true*>) rule S1

returns *false*

simple_backtrack(<*1=N',S=N'+S',sum(N' -1,S')*|*true*>) rule S2

simple_backtrack(<*S=N'+S',sum(N' -1,S')*| *1=N'*>)

simple_backtrack(<*sum(N'-1,S')*| *1=N' /\ S=N'+S'* >)

simple_backtrack(<*N'-1=0,S'=0* | *1=N' /\ S=N'+S'* >) rule S1

simple_backtrack(<*S'=0* | *1=N' /\ S=N'+S' / \ N'-1=0* >)

simple_backtrack(<*[]* | *1=N' /\ S=N'+S' / \ N'-1=0/\S'=0*>)

returns *1=N' /\ S=N'+S' / \ N'-1=0 /\ S'=0*

<u>*simpl({S},1=N' /\ S=N'+S' / \ N'-1=0 /\ S'=0) = S = 1*</u>

5

Incremental Solving

- The simple backtracking evaluation is <u>inefficient</u>, consider calls to *solv*
  - *solv(1=0)*
  - *solv(1 = N')*
  - *solv(1= N' /\ S = N + S')*
  - *solv(1= N' /\ S = N + S' /\ N'-1 = 0)*
  - *solv(1= N' /\ S = N + S' /\ N'-1 = 0 /\ S' = 0)*
- Repeated work

6

Incremental Constraint Solver

- An **incremental constraint solver** is a function *isolv* which takes a primitive constraint *c* and returns *true, false* or *unknown.* There is an implicit *constraint store* S
  - if *isolv(c) = true* then *S /\ c* is satisfiable
  - if *isolv(c) = false* then *S /\ c* is unsatisfiable
  - if *isolv(c) != false* then store is updated to *S /\ c*

7

Incremental Gauss-Jordan

- inc_gj(*c*)
  - *c* := eliminate(*c, S*)
  - **if** *c* is of the form 0 = 0 **return** *true*
  - **if** *c* is of the form d = 0 (d != 0) **return** *false*
  - rewrite *c* in the form *x = e*
  - *S* := eliminate(*S, x* = e) /\ *x = e*
  - **return** *true*
- eliminate(*C,x1 = e1* /\ ... /\ *xn = en*)
  - **foreach** *xi*
    - replace *xi* by *ei* throughout *C*
  - **return** *C*

8

Incremental GJ Example

Solving $1= N' \wedge S = N + S' \wedge N'-1 = 0 \wedge S' = 0$

	S	c	eliminate(c,S)
isolv(N'=1)	*true*	$1 = N'$	$1=N'$
isolv(S=N'+S')	$N'=1$	$S = N'+S'$	$S = 1+S'$
isolv(N'-1 = 0)	$N'=1 \wedge S = 1+S'$	$N'-1=0$	$0=0$
isolv(S' = 0)	$N'=1 \wedge S = 1+S'$	$S'=0$	$S'=0$
	$N'=1 \wedge S = 1 \wedge S'=0$		

9

Incremental goal solver

- CLP systems use a global constraint store *S* and incremental solvers
- inc_backtrack similar to simple_backtrack
  - uses incremental solver
  - store is not part of argument
  - functions: save_store, restore_store for saving and restoring the implicit store

10

inc_backtrack

- inc_backtrack(G)
  - **if** *G* is empty **return** *true*
  - **let** *G* be of the form *L*, *G'*
  - **case** *L* is a primitive constraint
    - if i*solv*(*L*) = *false* **return** *false*
    - **return** inc_backtrack(*G'*)
  - **case** *L* is an atom *p(s1,...,sn)*
    - **foreach** *p(t1,...,tn)* :- *B* in *defn(P,L)*
      - save_store()
      - **if** inc_backtrack(*s1=t1,..,sn=tn,B,G'*) **then**
        - **return** *C1*
      - restore_store()
    - **return** *false*

11

inc_solve_goal

- The incremental goal solving algorithm, making use of auxiliary functions to initialize and get the constraint store
- inc_solve_goal(*G*)
  - *W* := *vars(G)*
  - initialize_store()
  - **if** inc_backtrack(*G*) then
    - **return** *simpl*(*W*, get_store())
  - **return** *false*

12

Example execution sum(1,S)

	constraint store stack
inc_backtrack(*sum(1,S)*)	*<empty>*
inc_backtrack(*1=0, S = 0*)	*true* |
return *false*	*<empty>*
inc_backtrack(*1=N', S = N' + S', sum(N' -1, S')*)	*true* |
inc_backtrack(*S = N' + S', sum(N' -1, S')*)	*true* |
inc_backtrack(*sum(N'-1, S')*)	*true* |
inc_backtrack(*N'-1 = 0, S' = 0*)	*true*| *N' = 1 /\ S = 1 + S'*|
inc_backtrack(*S' = 0*)	*true*| *N' = 1 /\ S = 1 + S'*|
inc_backtrack(*[]*)	*true*| *N' = 1 /\ S = 1 + S'*|

simpl({*S*}, *N' = 1 /\ S = 1 /\ S' = 0*) = *S = 1*

13

Efficient saving and restoring

- Incremental solver requires saving/restoring the constraint store
- Dont need to save the entire store
- Save enough information to recreate store
  - **Trailing**: save modified parts of the constraint store in a trail and recover on backtracking
  - **Semantic backtracking**: store operations necessary to recover store

14

Trailing

- Associate a **timestamp** with each primitive constraint
- At a choicepoint
  - store the current timestamp
- Backtracking
  - remove all constraints with a later stamp
- Doesnt handle when an old primitive constraint is modified

15

Trailing

- Whenever an old constraint (from before the last choicepoint) is modified
  - save the old value in the **trail**
- Note we dont have to trail the same constraint again if it is modified again before another choicepoint

16

Trailing Gauss-Jordan

- Index each equation by arrival number
- Choicepoint saves:
  - index of last equation, *last*
  - trail of changes (initially empty)
- Whenever equation *i* is modified, if *i* <= *last* then each modified coefficient is added to trail <*i*,*x*,*a*> or <*i*,*constant*,*b*>

17

Semantic Backtracking

- Save high-level operations of how to restore the constraint store (domain dependent)
- For Gauss-Jordan
  - a new constraint only eliminates a variable *x*
  - remember the old coefficients of *x* and
  - **undo** the elimination on backtracking

18

Semantic Backtracking Ex.

Imagine store is

$1: \quad X = \boxed{Y} + 2Z + 4$
$2: \quad U = \boxed{3Y} + Z - 1$
$3: \quad V = 3$

Removing constraint

Adding constraint $Y + 2V + X = 2$

Eliminate vars $Y = -Z - 4$

Add coefficient * (*Y*+*Z*+*4*) to eqns 1,2 and remove 4

Eliminate Y using equation and add. Remember coefs

$[\langle 1, Y, 1\rangle, \langle 2, Y, 3\rangle]$

$1: \quad X = Z$
$2: \quad U = -2Z - 13$
$3: \quad V = 3$
$4: \quad Y = -Z - 4$

$1: \quad X = Y + 2Z + 4$
$2: \quad U = 3Y + Z - 1$
$3: \quad V = 3$

19

Extra Constructs

- So far "pure" programs (Chapter 4)
- Chapters 7 and 9 introduce
  - if-then-else
  - once
  - negation
  - optimization
- How are they implemented?

20

If-Then-Else,Once+Negation

- All three are implemented using a single construct, the **cut**, written !
- Cut prunes derivations from a tree
  - when reached: commit to this clause and remove any choices set up within this clause
- Very powerful, and dangerous
- Preferable to use if-then-else, once or negation rather than the lower level cut

21

Cut Example

Sum program for mode of usage: first arg fixed

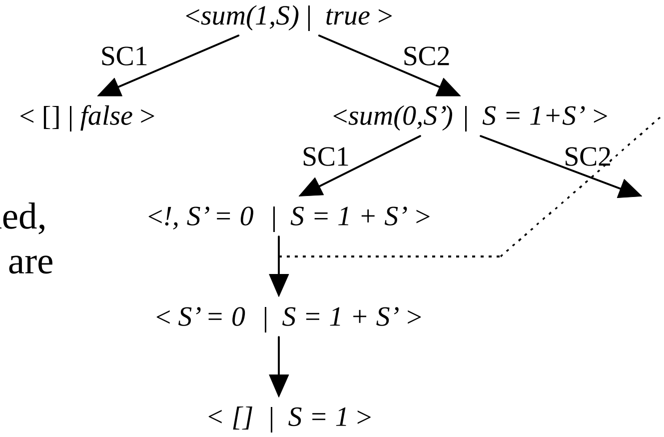
```
sum(N,SS) :-
     (N = 0 ->
          SS = 0
     ;
          N >= 1, SS = N + S,
          sum(N-1,S)
     ).
```

Equivalent version with cut

```
sum(N,SS) :- N = 0, !, SS = 0.
sum(N,SS) :- N >= 1, SS = N + S,
             sum(N-1, S).
```

22

Cut Derivation Tree



When ! reached, other choices are pruned away

23

Cut

- Cut commits to all choices made since when the atom which was rewritten that introduced the cut
- Assume rewriting atom *A’* using rule
  - *A* :- *L1, ..., Li, !, Li+1, ..., Ln*
- When ! reached all choices for rewriting *A’* and all choices in evaluation *L1, ..., Li* are removed

24

Implementing Cut

- Need save_store to return an index of the last store
- remove_choicepoints(*i*) removes all choicepoints with indexes >= *i*
- simply modify inc_backtrack for case introducing a cut

25

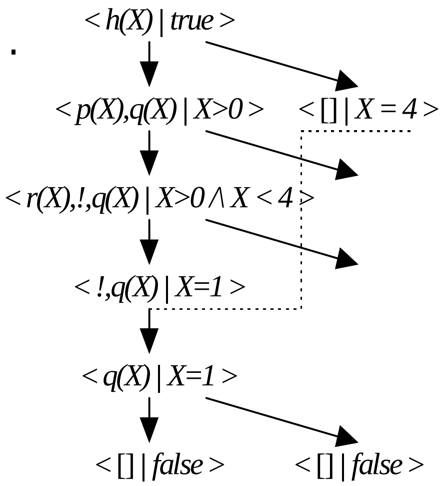
Modifying inc_backtrack

- **case** *L* is an atom *p(s1,...,sn)*
  - **foreach** *p(t1,...,tn)* :- *L1,...,Ln* in *defn(P,L)*
    - *i* := save_store()
    - **if** some *Lj* = ! **then**
      - **if** inc_backtrack(*s1=t1,..,sn=tn,L1,...,Lj-1)* **then**
        - remove_choicepoints(i)
        - **return** inc_backtrack(*Lj+1,...,Ln,G'*)
    - **elseif** inc_backtrack(*s1=t1,..,sn=tn,,G'*) then
      - **return** *true*
    - restore_store()
  - **return** *false*

26

Cut Example 2

```
h(X):- X > 0,p(X),q(X).
h(4).
p(X) :-X < 4, r(X),!.
p(3).
r(1).
r(2).
q(2),
q(3).
```



27

Cut Example 2

		constraint storestack
inc_backtrack(*h(X)*)		*<empty>*
inc_backtrack(*X > 0, p(X), q(X)*)		*true* |
inc_backtrack(*p(X), q(X)*)	**index 2**	*true* | *X > 0* |
inc_backtrack(*X < 4, r(X)*) (before cut)		*true* | *X > 0* |
inc_backtrack(*r(X)*)		*true* | *X > 0* | *X > 0* ∧ *X < 4* |
return *true*	**remove upto 2**	*true* |
inc_backtrack(*q(X)*) (after cut)		*true*| *X = 1* |
return *false*	**restore 2**	*true* |
return *false*	**restore 1**	*<empty>*
inc_backtrack(*X = 4*)		*true*|
return *true*	Answer: *X = 4*	

28

If-Then-Else,Once+Negation

All are implemented using the meta-programming facilities and the cut.

```
once(G) :- call(G), !.

not(G) :- call(G)!, !, fail.
not(G).

G1 -> G2 ; G3 :- call(G1), !, call(G2).
G1 -> G2 ; G3 :- call(G3).
```

29

Optimization

- Implementing `minimize(G,E)`
  - minimize_store(*E*): returns the minimal value of *E* wrt to current constraint store
  - search the derivation tree of *G* and collect minimum value *m* of *E*, then execute *E* = *m*, *G*
- Multiple approaches to search
  - retry search (restart after finding soln)
  - backtrack search (continue after finding)

30

Retry Optimization

- **case** *L* is minimization literal *minimize(G,E)*
  - *i* := save_store()
  - *m* := + ∞
  - **while** inc_backtrack(*E* < *m*, *G*) **do**
    - *m* := minimize_store(*E*)
    - remove_choicepoints(*i*+*1*)
    - restore_store()
    - *i* := save_store()
  - restore_store()
  - **return** inc_backtrack(*E* = *m*, *G*, *G'*)

31

Retry Example

Evaluating `minimize(butterfly(S,P), -P)`

inc_backtrack(*-P* < + ∞ , *butterfly(S,P)*)
  answer: *-P < + ∞ /\ P = -100 /\ 0 <= S <= 1*
  m := 100
inc_backtrack(*-P < 100 , butterfly(S,P)*)
  answer: *-P < 100 /\ P = 100S - 200 /\ 1 <= S <= 3*
  m := -100
inc_backtrack(*-P < -100, butterfly(S,P)*)
  returns *false*
inc_backtrack(*-P = -100, butterfly(S,P)*)
  answers: *P = 100 /\ S = 3* (twice)

32

Backtracking Optimization

- `minimize(G,E)`
- Search the derivation tree for *G*
- At each success update the minimal value *m* of *E* found (handled by a *catch* literal)
- Then execute *E=m,G*

33

Backtracking Optimization

- **case** *L* is minimization literal *minimize(G,E)*
  - *i* := save_store()
  - *m* := + ∞
  - inc_backtrack(*G*, *catch(m,E)*)
  - restore_store(*i*)
  - **return** inc_backtrack(*E=m,G,G'*)
- **case** *L* is a catch subgoal *catch(m,E)*
  - **if** *isolv*(*E* < *m*) != *false* **then**
    - *m* := minimize_store(*E*)
  - **return** *false*

34

Backtracking Example

inc_backtrack(*butterfly(S,P),catch(+ ,-P)*)

inc_backtrack(*catch(+ ,-P)*)

store: *P = -100 /\ 0 <= S <= 1* **sets** *m* := 100

inc_backtrack(*catch(100,-P)*)

store: *P = 100S - 200 /\ 1 <= S <= 3* **sets** *m* := -100

inc_backtrack(*catch(-100,-P)*)

store: *P = -100S +400 /\ 3 <= S <= 5*

isolv(-P < -100) **fails** no update

inc_backtrack(*catch(100,-P)*)

store: *P = -100 S >= 5* **fails** no update

inc_backtrack(*-P = -100, butterfly(S,P)*)

answers: *P = 100 /\ S = 3* (twice)

35

Other Incremental Solvers

- Incremental Tree Solving
  - Use the store to eliminate variables and solve remainder as before, then use it to eliminate
  - inc_tree_solve(*c*)
    - *c* := eliminate(*c*, *S*)
    - *R* := unify(*c*)
    - **if** *R* = *false* **then return** *false*
    - *S* := eliminate(*S*,*R*) /\ *R*
    - **return** *true*

36

Incremental Tree Solving Ex.

Constraints collected by goal `append([a],[b,c],L)`

$[a] = [F|R], [b,c] = Y, L = [F|Z], R = [], Y = Z$

c	S	elim(c)	unify(c)
$[a]=[F\|R]$	*true*	$[a]=[F\|R]$	$F=a \wedge R=[]$
$[b,c]=Y$	$F=a \wedge R=[]$	$[b,c]=Y$	$Y=[b,c]$
$L=[F\|Z]$	$F=a \wedge R=[] \wedge Y=[b,c]$	$L=[a\|Z]$	$L=[a\|Z]$
$R=[]$	$F=a \wedge R=[] \wedge Y=[b,c] \wedge L=[a\|Z]$	$[]=[]$	*true*
$Y=Z$	$F=a \wedge R=[] \wedge Y=[b,c] \wedge L=[a\|Z]$	$[b,c]=Z$	$Z=[b,c]$

$F=a \wedge R=[] \wedge Y=[b,c] \wedge L=[a,b,c] \wedge Z=[b,c]$

37

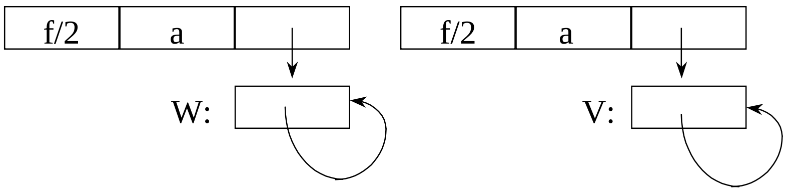
Data Structures for Trees

- Tree constraints are stored/manipulated as dynamic data structures
  - **variable**: unique memory cell (pointer)
    - unconstrained: self-pointer
    - equated to term: pointer at term rep
  - **term** *f(t1,...,tn)*: *n+1* memory cells
    - first: constructor info *f/n*
    - rest: pointers to *t1,...,tn*
  - optimization: store subtrees with no children directly

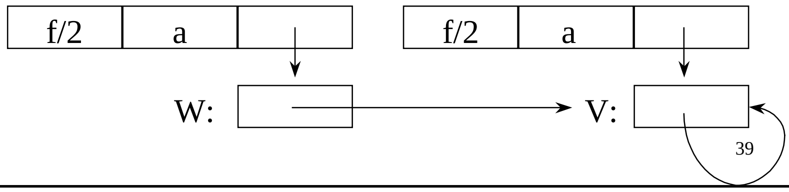
38

Data Structures for Trees

Handling the equation $f(a,W) = f(a,V)$



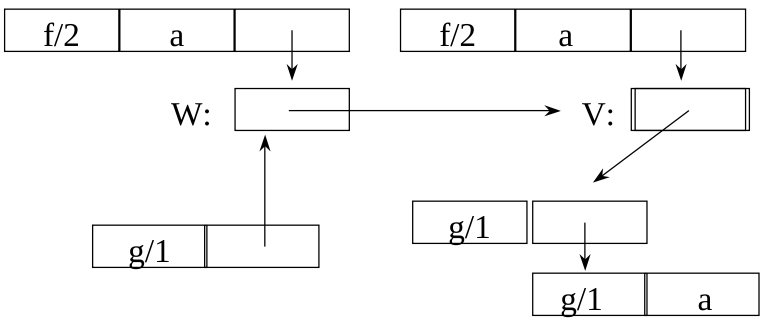
Match constructor/arities and each arg. Eqn $W = V$ binds W to V



39

Data Structures for Trees

Incrementally adding $g(W) = g(g(a))$

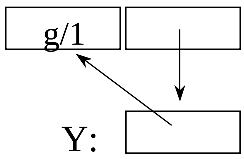


Represents solved form: $V = g(a) \wedge W = g(a)$

40

Occurs Check Revisited

- Most implementations ignore the occurs check!
- Problems: e.g. $Y = g(Y)$
- Builds cyclic structures
- Infinite computation
- e.g. $Y = g(Y), Z = g(Z), Y = Z$



41

Incremental Bounds Cons.

- Propagation is essentially incremental
- incremental bounds consistency:
  - Add new prim. constraint to store and queue
  - Pick prim. constraint from queue
  - Enforce its bounds consistency
  - Add prim. constraint with modified variables to queue
  - Repeat until queue is empty, or empty domain

42

Incremental Bounds Ex.

Smugglers knapsack, no whiskey

capacity *profit*

$$4W + 3P + 2C \leq 9 \wedge 15W + 10P + 7C \geq 30$$

$$D(W) = [0..0], D(P) = [1..3], D(C) = [0..3]$$

Add first constraint

Add second constraint

43

CLP Systems Summary

- Incremental constraint solving
  - essential for efficiency
- Global constraint store
  - require efficient save and restore
- The Cut!
  - implements if-then-else, once + negation
- Minimization
  - many possible implementations

44