# *Chapter 9:Advanced Programming Techniques*

*A mixed bag of different methods to improve the efficiency of finding a solution*

1

## *Advanced Programming*

- Extending the Constraint Solver
- Combining Symbolic and Arithmetic Reasoning
- Programming Optimization
- Higher-Order Predicates
- Negation
- Dynamic Scheduling

2

## *Extending the Solver*

- CLP program provides a solver for user-defined constraints.
- Efficient only for certain modes of usage as opposed to primitive constraints
- Sometimes worth creating user-defined constraints which will be efficient in all modes of usage

3

## *Solver Extension Examples*

Complex numbers *x* + *iy* represented as *c(x,y)*

```
c_add(c(R1,I1), c(R2,I2), c(R3,I3)) :-
     R3 = R1 + R2, I3 = I1 + I2.
c_mult(c(R1,I1), c(R2,I2), c(R3,I3)) :-
     R3 = R1*R2 - I1*I2, I3 = R1*I2 + R2*I1.
```

- Efficient in all modes of usage
- Only involves a fixed number of primitive constraints

4

## Solver Extension Examples

Sequence constraints (sequences represented by lists) non empty sequence, and concatenation of list of sequences equals a sequence

```
not_empty([_|_]).

concat([S1],S1).
concat([S1,S2|Ss],S) :-
    append(S1,T,S), concat([S2|Ss],T).
```

- concat is efficient only when all the sequences in the first argument are fixed in length

5

## Solver Extension Examples

Problems with solver extensions: user-defined constraints that involve search may behave badly

E.g. Find two sequences L1 and L2 where L2 is not empty but their concatenation is empty.

```
not_empty(L2),concat([L1,L2],L),L = [].
```

No solution, but the goal runs forever.

6

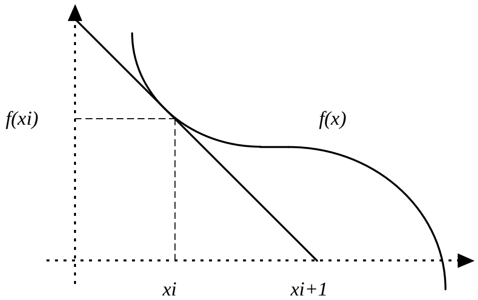
## *Stronger Constraint Solvers*

- Imagine solving $(x+2)^3 = 0$
- `(X+2)*(X+2)*(X+2) = 0`
- Answer is *unknown* (from CLP(R))
- But we can program a constraint solver (Newton-Raphson method) to solve the problem

7

## *Newton-Raphson Method*

From guess *xi* determine where the line of slope *f'(xi)* that passes through (*xi*,*f(xi)*) hits the *x* axis. This is the next guess *xi*+1




Need user-defined constraints for the function and its derivative

```
f(X,F) :- F = (X+2)*(X+2)*(X+2).
df(X,F) :- F = 3*(X+2)*(X+2).
```

8

## Newton-Raphson Program

```
solve_nr(E,X0,X0) :-
     f(X0,F0), -E <= F0, F0 <= E.
solve_nr(E,X0,X) :-
     f(X0,F0), df(X0,DF0),
     F0 = DF0 * X0 + C, 0 = DF0 * X1 + C,
     solve_nr(E,X1,X).
```

`solve_nr(E,Xo,X)` returns value *X* where |*f(X)*| <= *E*

Mode of usage is first and second arg fixed.

Note use of constraint solving to determine *C* and *X1*

9

## Combining Symbolic and Arithmetic Reasoning

- Tree constraints give symbolic reasoning
- We can combine both symbolic and arithmetic reasoning
- E.g. representing mathematical expressions
  - plus(x,y) == $x + y$
  - minus(x,y) == $x - y$
  - mult(x,y) == $x \times y$
  - power(x,y) == $x^y$
  - etc.

10

## Evaluating an Expression

```
evaln(x,X,X).
evaln(N,_,N) :- arithmetic(N).
evaln(power(X,N),X,E) :- E = power(X,N).
evaln(plus(F,G),X,EF+EG) :-
     evaln(F,X,EF), evaln(G,X,EG).
evaln(mult(F,G),X,EF*EG) :-
     evaln(F,X,EF), evaln(G,X,EG).
```

`evaln(T,X,V)` gives value *V* of an expression *T* in variable *x*, using value *X* for *x*. For example

`evaln(plus(power(x,2),3),X,E)` gives $E = X^2 + 3$

11

## Symbolic Differentiation

```
deriv(x,1).
deriv(N,0) :- arithmetic(N).
deriv(power(x,N),mult(power(x,N1)) :-
     N1 = N - 1.
deriv(plus(F,G),plus(DF,DG)) :-
     deriv(F,DF), deriv(G,DG).
deriv(mult(F,G),D) :-
     D = plus(mult(DF,G),mult(f,DG)),
     deriv(F,DF), deriv(G,DG).
```

`deriv(T,DT)` gives expression *DT* which is the differentiation of *T* wrt *x*. For example

`deriv(plus(power(x,2),3),D)`

`D = plus(power(x,1),0)`

12

## Newton-Raphson Revisited

```
dsolve(E,F,X0,X) :- deriv(F,DF),
     solve_nr(E,F,DF,X0,X).
solve_nr(E,F,DF,X0,X0) :-
     evaln(F,X0,F0), -E <= F0, F0 <= E.
solve_nr(E,F,DF,X0,X) :-
     evaln(F,X0,F0), evaln(DF,X0,DF0),
     F0 = DF0 * X0 + C, 0 = DF0 * X1 + C,
     solve_nr(E,F,DF,X1,X).
```

Use symbolic differentiation to determine derivative

`dsolve(0.001,plus(power(x,2),plus(mult(3,x),2),5,X)`

gives answer X = -1

13

## Programming Optimization

- Optimization algorithms can be programmed just as constraint solvers
- Examples
  - Branch and Bound minimization
  - Optimistic partitioning

14

## *Programming Branch+Bound*

- Predicate `bounded_prob` defines the problem constraints with bounds
- minimize f subject to f < current best and bounded problem (for current bounds)
- examine the solution
  - if all integer return as new best solution
  - otherwise add new bounds that split on first non-integer variable, try lower bound split then upper bound split

15

## *Optimistic Partitioning*

- Rather than search the entire space
  - first try finding a solution in the lower half of range for the objective function
  - only if that fails try the upper half
- Can avoid finding a long chain of slightly better answers
- Example on the scheduling program

16

## *Optimistic Partitioning*

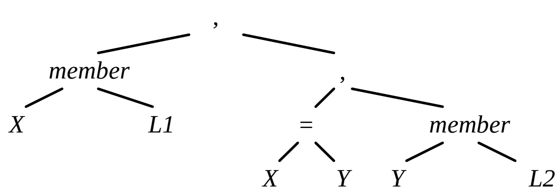
```
split_min(Data,Min0,Max0,JL0,JL) :-
   Mid = (Min0+Max0)//2,
   (Min0 <= End, End <= Mid,
    schedule(Data,End,JL1), indomain(End)->
     Max = End-1,
     split_min(Data,Min0,Max,JL1,JL)
   ; (Mid+1 <= End, End <= Max0,
      schedule(Data,End,JL1),indomain(End)->
         Min = Mid+1, Max = End-1,
         split_min(Data,Min,Max,JL1,JL)
   ;  JL = JL0)).
```

*JL0* is the current best solution, *JL* the minimal solution

17

## *Higher-Order Predicates*

- **higher-order predicates** take a constraint or goal as an argument
  - e.g. once, if-then-else
- goals can be represented using terms, e.g.
  - `member(X,L1),X=Y,member(Y,L2)`




18

## Call

- built-in literal `call(G)` acts like the goal *G*
- requires that *G* is constrained to be a term with the syntax of a goal when executed
- Examples
  - `X = member(A,[a,b]), call(X)`
  - has answers *A* = *a* and *A* = *b*
  - `once(G) :- (call(G) -> true ; fail).`
  - defines once in terms of if-then-else

19

## Negation

- Important higher-order predicate `not(G)`
- Useful to have the negation of a user-defined predicate e.g. member, not_member
- Drawback it only works as expected in quite restricted modes of usage

20

# *Negation*

- **negative** literal: not(*G*)
- if *G* succeeds then fail otherwise succeed
- **negation derivation step**: *G1* is *L1, L2, ..., Lm, where L1* is not(*G*)
- **if** *<G | C1>* succeeds *C2* is *false*, *G2* is *[]*
  - **else** *C2* is *C1*, *G2* is *L2, ..., Lm*

21

# *Negation*

- Implementing disequality
- `ne(X,Y) :- not(X=Y).`
- Goal `X = 2, Y = 3, ne(X,Y)` succeeds
- Goal `X = 2, Y = 2, ne(X,Y)` fails
- Goal `X = 2, ne(X,Y), Y = 3` fails!

$$\langle ne(X,Y) \mid X = 2 \wedge Y = 3 \rangle \Downarrow \langle not(X = Y) \mid X = 2 \wedge Y = 3 \rangle \Downarrow \langle [] \mid X = 2 \wedge Y = 3 \rangle$$

$$\langle X = Y \mid X = 2 \wedge Y = 3 \rangle \Downarrow \langle [] \mid false \rangle$$

$$\langle ne(X,Y), Y = 3 \mid X = 2 \rangle \Downarrow \langle not(X = Y), Y = 3 \mid X = 2 \rangle \Downarrow \langle [] \mid false \rangle$$

$$\langle X = Y \mid X = 2 \rangle \Downarrow \langle [] \mid X = 2 \wedge X = Y \rangle$$

22

## *Safe Negation*

- A negative literal is guaranteed to act right (as the negationof its argument) when the goal is fixed (has no variables)
- Otherwise problems with solver
  - `Y*Y=4, Y >= 0, not(Y >= 1)` fails!
  - `X < 0, Y > 1, Z > 2, not(X=Y*Z)` fails!
- One other usage (testing compatibility)
  - `is_compatible(G) :- not(not(G)).`
  - true if (non-fixed) *G* is compatible with store

23

## *Dynamic Scheduling*

- Because answers do not depend on the execution order of literals we can relax the order of processing
- Dynamic scheduling allows the execution of user-defined constraints to be delayed until the arguments represent a safe mode of usage

24

# Dynamic Scheduling Example

```
:- delay_until(ground(X) and ground(Y),ne(X,Y)).
ne(X,Y) :- not(X = Y).
```

Delays the execution of ne literals until the mode of usage is safe (both arguments are fixed).

$$\begin{array}{c}\langle \underline{ne(X,Y)} \mid X = 2 \wedge Y = 3 \rangle \\ \Downarrow \\ \langle \underline{not(X = Y)} \mid X = 2 \wedge Y = 3 \rangle \\ \Downarrow \\ \langle [] \mid X = 2 \wedge Y = 3 \rangle \\ \\ \langle X = Y \mid X = 2 \wedge Y = 3 \rangle \\ \Downarrow \\ \langle [] \mid false \rangle \end{array}$$

$$\begin{array}{c}\langle ne(X,Y), \underline{Y = 3} \mid X = 2 \rangle \\ \Downarrow \\ \langle \underline{ne(X,Y)} \mid X = 2 \wedge Y = 3 \rangle \\ \Downarrow \\ \langle \underline{not(X = Y)} \mid X = 2 \wedge Y = 3 \rangle \\ \Downarrow \\ \langle [] \mid X = 2 \wedge Y = 3 \rangle \\ \\ \langle X = Y \mid X = 2 \wedge Y = 3 \rangle \\ \Downarrow \\ \langle [] \mid false \rangle \end{array}$$

25

# Delay Conditions

- takes a constraint and returns *true* or *false*, if *true* it is said to **enable** the condition
- **primitive delay condition**:
  - `ground(X)`: *X* takes a fixed value
  - `nonvar(X)`: *X* cannot take all values
  - `ask(c)`: the constraint implies *c*
- **delay condition**: primitive delay or
  - `Cond1 and Cond2`: both conditions hold
  - `Cond1 or Cond2`: either condition holds

26

## *Delaying Literals*

- **delaying literal**: `delay_until(Cond,Goal)`
- Evaluation of *Goal* will delay until the constraint store enables *Cond*
- Two forms
  - **predicate-based**: for all user-defined constraints for predicate *p*
  - `:- delay_until(`*Cond*`,p(X))`
  - **goal-based**: for a particular user-defined constr.
  - `..., delay_until(`*Cond*`,p(X)), ...`

27

## *Delaying Literals*

- Can mimic goal-based with predicate based and vice-versa. Examine predicate-based
- How do delaying literals execute
- We need to slightly modify the execution strategy

28

## Selection Derivation

- A literal *Li* is selected for rewriting by a selection strategy
- **derivation step**: *G1* is *L1,..., Li, ..., Lm*
  - *Li* is a primitive constraint, *C2* is *C1 /\ Li*
    - **if** *solv(C /\ Li) = false* **then** *G2* = []
    - **else** *G2 = L1,...,Li-1,Li+1, ..., Lm*
  - *Li* is a user-defined constraint, *C2* is *C1* and *G2* is the rewriting of *G1* at *Li* using some rule and renaming

29

## Selection Derivation + Delay

- Literal selection strategy is **safe** if it only selects user-defined constraints `p(X)` with a delay declaration
  - `:- delay_until(`*Cond*`,p(X))`
- if the store enables *Cond*
- Sometimes in a state <*G*|*C*> no literal can be selected, the state is **floundered**
- A derivation with floundered final state is successful with answer G /\ C

30

# Delaying Program

The string constraint solver but where append is delayed

```
not_empty([_|_]).

concat([S1],S1).
concat([S1,S2|Ss],S) :-
     append(S1,T,S), concat([S2|Ss],T).

:- delay_until(nonvar(X) or nonvar(Z),
               append(X,Y,Z))
append([],Y,Y).
append([A|X],Y,[A|Z]) :- append(X,Y,Z).
```

31

# Derivation with Delay

Goal:

```
not_empty(L2),
concat([L1,L2],L),
L = [].
```

Note that the derivation runs forever if there is no delay condition.

$$\langle \underline{not\_empty(L2)}, concat([L1,L2],L), L=[] \mid true \rangle$$
$$\Downarrow$$
$$\langle \underline{concat([L1,L2],L)}, L=[] \mid L2=[\_|\_] \rangle$$
$$\Downarrow$$
$$\langle append(L1,T,L), \underline{concat([L2],T)}, L=[] \mid L2=[\_|\_] \rangle$$
$$\Downarrow$$
$$\langle append(L1,T,L), \underline{L=[]} \mid L2=[\_|\_] \wedge T=L2 \rangle$$
$$\Downarrow$$
$$\langle \underline{append(L1,T,L)} \mid L2=[\_|\_] \wedge T=L2 \wedge L=[] \rangle$$
$$\Downarrow$$
$$\langle [] \mid false \rangle$$

32

## *Floundered Derivation*

In the final step there is no literal that can be selected

$$\langle \underline{concat([L1, L2], L)} | true \rangle$$
$$\Downarrow$$
$$\langle append(L1, T, L), \underline{concat([L2], T)} | true \rangle$$
$$\Downarrow$$
$$\langle append(L1, T, L) | T = L2 \rangle$$

Successful derivation answer

*append(L1,T,L) /\ T = L2*

or simplified *append(L1,L2,L)*

33

## *Delay for Writing Solvers*

Boolean solving by local propagation. The constraint and(X,Y,Z) makes Z = X /\ Y it waits until two of the three are known before executing

```
:- delay_until((ground(X) and ground(Y)) or
               (ground(X) and ground(Z)) or
               (ground(Y) and ground(Z)),
               and(X,Y,Z))
and(0,0,0).
and(0,1,0).
and(1,0,0).
and(1,1,1).
```

34

## *Delay for Solvers*

Goal: `and(A1,A2,0), and(A1,1,A3), and(1,0,A3)` has 13 states in the simplified derivation tree using delay and 29 states without using delay

$$\langle and(A1, A2, 0), and(A1, 1, A3), \underline{and(1, 0, A3)} \mid true \rangle$$
$$\Downarrow$$
$$\langle and(A1, A2, 0), \underline{and(A1, 1, A3)} \mid A3 = 0 \rangle$$
$$\Downarrow$$
$$\langle \underline{and(A1, A2, 0)} \mid A3 = 0 \wedge A1 = 0 \rangle$$
$$\Downarrow$$
$$\langle [] \mid A3 = 0 \wedge A1 = 0 \wedge A2 = 0 \rangle$$

35

## *Advanced Programming Techniques Summary*

- Extending the constraint solver is straightforward in CLP, but usually they have restricted modes of usage
- Meta-programming and dynamic scheduling provide ways of making them more robust
- Similarly new optimization can be programmed
- Negation is useful is modelling but of restricted usefulness as provided in CLP

36