Chapter 9: Advanced Programming Techniques

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

Advanced Programming

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

Solver Extension Examples

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

\[
\begin{align*}
c_{\text{add}}(c(R1,I1), c(R2,I2), c(R3,I3)) & : - \\
& R3 = R1 + R2, I3 = I1 + I2.
\end{align*}
\]

\[
\begin{align*}
c_{\text{mult}}(c(R1,I1), c(R2,I2), c(R3,I3)) & : - \\
\end{align*}
\]

- Efficient in all modes of usage
- Only involves a fixed number of primitive constraints
**Solver Extension Examples**

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

\[
\text{not_empty}([\_|\_]).
\]

\[
\text{concat}([S1],S1).
\]

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

- \text{concat} is efficient only when all the sequences in the first argument are fixed in length

**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.

\[
\text{not_empty}(L2),\text{concat}([L1,L2],L),L = \text{[]}.
\]

No solution, but the goal runs forever.
Stronger Constraint Solvers

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

Newton-Raphson Method

From guess \(x_i\) determine where the line of slope \(f'(x_i)\) that passes through \((x_i,f(x_i))\) hits the \(x\) axis. This is the next guess \(x_{i+1}\)

Need user-defined constraints for the function and its derivative

\[
\begin{align*}
&f(X,F) :- F = (X+2) \cdot (X+2) \cdot (X+2).
&df(X,F) :- F = 3 \cdot (X+2) \cdot (X+2).
\end{align*}
\]
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,X0,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

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.
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 \[ E = X^2 + 3 \]

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)
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

Programming Optimization

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

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
Optimistic Partitioning

\[
\text{split}_\text{min}(\text{Data, Min0, Max0, JL0, JL}) :\quad \\
\quad \text{Mid} = (\text{Min0} + \text{Max0}) / 2, \\
\quad (\text{Min0} \leq \text{End}, \text{End} \leq \text{Mid}, \\
\quad \text{schedule} (\text{Data}, \text{End}, \text{JL1}), \text{indomain} (\text{End}) \rightarrow \\
\quad \text{Max} = \text{End} - 1, \\
\quad \text{split}_\text{min}(\text{Data, Min0, Max, JL1, JL}) \\
\quad ; (\text{Mid} + 1 \leq \text{End}, \text{End} \leq \text{Max0}, \\
\quad \text{schedule} (\text{Data}, \text{End}, \text{JL1}), \text{indomain} (\text{End}) \rightarrow \\
\quad \text{Min} = \text{Mid} + 1, \text{Max} = \text{End} - 1, \\
\quad \text{split}_\text{min}(\text{Data, Min, Max, JL1, JL}) \\
\quad ; \text{JL} = \text{JL0}).
\]

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

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.
  - \text{member}(X, L1), X = Y, \text{member}(Y, L2)
Call

- built-in literal \texttt{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
- \( X = \text{member}(A, [a,b]), \text{call}(X) \)
- has answers \( A = a \) and \( A = b \)
- \( \text{once}(G) :- (\text{call}(G) \rightarrow \text{true} ; \text{fail}). \)
- defines once in terms of if-then-else

Negation

- Important higher-order predicate \texttt{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
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

Negation

- Implementing disequality
  - \texttt{ne(X,Y)} :- not(X=Y).
- Goal \(X = 2, \ Y = 3\), \texttt{ne(X,Y)} succeeds
- Goal \(X = 2, \ Y = 2\), \texttt{ne(X,Y)} fails
- Goal \(X = 2, \ \texttt{ne(X,Y)}, \ Y = 3\) fails!
  
  \[
  \begin{align*}
  \langle \texttt{ne(X,Y)} | X = 2 \land Y = 3 \rangle & \Downarrow \langle \texttt{ne(X,Y), Y = 3} | X = 2 \rangle \\
  \langle \texttt{not(X = Y)} | X = 2 \land Y = 3 \rangle & \Downarrow \langle \texttt{not(X = Y), Y = 3} | X = 2 \rangle \\
  \langle [] | X = 2 \land Y = 3 \rangle & \Downarrow \langle [] | \texttt{false} \rangle \\
  \langle X = Y | X = 2 \land Y = 3 \rangle & \Downarrow \langle X = Y | X = 2 \rangle \\
  \langle [] | \texttt{false} \rangle & \Downarrow \langle [] | X = 2 \land X = Y \rangle
  \end{align*}
  \]
Safe Negation

- A negative literal is guaranteed to act right (as the negation of its argument) when the goal is fixed (has no variables)
- Otherwise problems with solver
  - $Y \cdot Y = 4, Y \geq 0, \neg (Y \geq 1)$ fails!
  - $X < 0, Y > 1, Z > 2, \neg (X = Y \cdot Z)$ fails!
- One other usage (testing compatibility)
  - $\text{is_compatible}(G) :- \neg \neg(G)$.
  - true if (non-fixed) $G$ is compatible with store

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
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).

\[
\begin{align*}
\langle \text{ne}(X,Y) | X = 2 \land Y = 3 \rangle & \Downarrow \langle \text{ne}(X,Y), Y = 3 | X = 2 \rangle \\
\langle \text{not}(X = Y) | X = 2 \land Y = 3 \rangle & \Downarrow \langle \text{not}(X = Y), X = 2 | Y = 3 \rangle \\
\langle \text{false} | X = 2 \land Y = 3 \rangle & \Downarrow \langle \text{false} | X = 2 \land Y = 3 \rangle \\
\end{align*}
\]

Delay Conditions

- takes a constraint and returns \textit{true} or \textit{false}, if \textit{true} it is said to \textbf{enable} the condition

- \textbf{primitive delay condition}:  
  - \text{ground}(X): X takes a fixed value  
  - \text{nonvar}(X): X cannot take all values  
  - \text{ask}(c): the constraint implies c

- \textbf{delay condition}: primitive delay or  
  - Cond1 and Cond2: both conditions hold  
  - Cond1 or Cond2: either condition holds
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)), ...

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
Selection Derivation

- A literal $L_i$ is selected for rewriting by a selection strategy
- **derivation step**: $G_1$ is $L_1,..., L_i, ..., L_m$
  - $L_i$ is a primitive constraint, $C_2$ is $C_1 \land L_i$
    - if $\text{solv}(C \land L_i) = \text{false}$ then $G_2 = []$
    - else $G_2 = L_1, ..., L_{i-1}, L_{i+1}, ..., L_m$
  - $L_i$ is a user-defined constraint, $C_2$ is $C_1$ and $G_2$ is the rewriting of $G_1$ at $L_i$ using some rule and renaming

Selection Derivation + Delay

- Literal selection strategy is **safe** if it only selects user-defined constraints $p(X)$ with a delay declaration
  - $:- \text{delay\textunderscore until}(\text{Cond}, p(X))$
- if the store enables $\text{Cond}$
- 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 $G \land C$
**Delaying Program**

The string constraint solver but where append is delayed

\[
\text{not\_empty}([|\textunderscore|\textunderscore]).
\]

\[
\text{concat}([S1],S1).
\]

\[
\text{concat}([S1,S2|Ss],S):=
\quad\text{append}(S1,T,S), \text{concat}([S2|Ss],T).
\]

\[
:= \text{delay\_until}(\text{nonvar}(X) \text{ or } \text{nonvar}(Z),
\quad\text{append}(X,Y,Z))
\]

\[
\text{append}([],Y,Y).
\]

\[
\text{append}([A|X],Y,[A|Z]):=\text{append}(X,Y,Z).
\]

---

**Derivation with Delay**

Goal:

\[
\text{not\_empty}(L2), \quad \text{concat}([L1,L2],L), \quad L = [].
\]

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

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

\[
\begin{align*}
\text{concat([L1,L2],L)} & \text{true} \\
\Rightarrow \text{append(L1,T,L), concat([L2],T)} & \text{true} \\
\Rightarrow \text{append(L1,T,L)} & \text{T = L2}
\end{align*}
\]

Successful derivation answer

\[
\text{append(L1,T,L)} \land T = L2
\]
or simplified \( \text{append(L1,L2,L)} \)

Delay for Writing Solvers

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

\[
\text{:- delay_until((ground(X) \land ground(Y)) or (ground(X) \land ground(Z)) or (ground(Y) \land ground(Z)), and(X,Y,Z))}
\]

\[
\begin{aligned}
&\text{and(0,0,0).} \\
&\text{and(0,1,0).} \\
&\text{and(1,0,0).} \\
&\text{and(1,1,1).}
\end{aligned}
\]
**Delay for Solvers**

Goal: \( \text{and}(A1,A2,0), \text{and}(A1,1,A3), \text{and}(1,0,A3) \)

has 13 states in the simplified derivation tree using delay

and 29 states without using delay

\[
\begin{align*}
\langle & \text{and}(A1,A2,0),\text{and}(A1,1,A3),\text{and}(1,0,A3)\rangle \\
\downarrow \\
\langle & \text{and}(A1,A2,0),\text{and}(A1,1,A3) \rangle \\
\downarrow \\
\langle & \text{and}(A1,A2,0) \rangle \\
\downarrow \\
\langle [ ] A3 = 0 \land A1 = 0 \land A2 = 0 \rangle
\end{align*}
\]

**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 in modelling but of restricted usefulness as provided in CLP