Chapter 10: CLP Systems

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

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
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 \text{defn}(P,L) \text{ which returns rules defining } L \text{ in program } P \text{ in the order they occur, renamed to not contain any previous variables}
- simple_solve_goal(G)
  - \text{return \ } simpl(\text{vars}(G) , \text{simple_backtrack}(<G|true>))

simple_backtrack

- \text{simple_backtrack}(<G|C>)
  - \text{if } G \text{ is empty return } C
  - \text{let } G \text{ be of the form } L, G'
  - \text{case } L \text{ is a primitive constraint}
    - \text{if } \text{solv}(C \land L) = false \text{ return false}
    - \text{return \ } \text{simple_backtrack}(<G'|C \land L>)
  - \text{case } L \text{ is an atom } p(s1,...,sn)
    - \text{foreach } p(t1,...,tn) :- B \text{ in } \text{defn}(P,L)
      - \text{C1 = simple_backtrack(<s1=t1,...,sn=tn,B,G'|C>)}
      - \text{if } C1 != false \text{ return } C1
    - \text{return false}
Example execution \textit{sum}(1,S)

\begin{itemize}
\item (S1) \textit{sum}(0,0).
\item (S2) \textit{sum}(N,N+S) :- \textit{sum}(N-1,S).
\end{itemize}

\begin{verbatim}
simple_backtrack(<\textit{sum}(1,S)|true>)
simple_backtrack(<1=0,S=0|true>) rule S1
    returns false
simple_backtrack(<1=N',S=N'+S',\textit{sum}(N'-1,S')|1=N'>) rule S2
    simple_backtrack(<S=N'+S',\textit{sum}(N'-1,S')|1=N'>)
    simple_backtrack(<\textit{sum}(N'-1,S')|1=N'\land S=N'+S'>)
        simple_backtrack(<N'-1=0,S'=0|1=N'\land S=N'+S'>) rule S1
        simple_backtrack(<S'=0|1=N'\land S=N'+S'/\land N'-1=0>)
        simple_backtrack(<1|1=N'\land S=N'+S'/\land N'-1=0\land S'=0>)
    returns 1=N'\land S=N'+S'/\land N'-1=0\land S'=0
simpl(S,1=N'\land S=N'+S'/\land N'-1=0\land S'=0) = S=1
\end{verbatim}

Incremental Solving

- The simple backtracking evaluation is inefficient, consider calls to \textit{solv}
  - \textit{solv}(1=0)
  - \textit{solv}(1 = N')
  - \textit{solv}(1 = N' \land S = N + S')
  - \textit{solv}(1 = N' \land S = N + S' \land N'-1 = 0)
  - \textit{solv}(1 = N' \land S = N + S' \land N'-1 = 0 \land S' = 0)
- Repeated work
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`

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`
**Incremental GJ Example**

Solving \( I = N' \land S = N + S' \land N' - 1 = 0 \land S' = 0 \)

<table>
<thead>
<tr>
<th>( isolv(N'=1) )</th>
<th>( S )</th>
<th>( c )</th>
<th>( eliminate(c, S) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( isolv(S=N'+S') )</td>
<td>( N' = 1 )</td>
<td>( 1 = N' )</td>
<td>( I = N' )</td>
</tr>
<tr>
<td>( isolv(N'-1 = 0) )</td>
<td>( S = N'+S' )</td>
<td>( S = 1 + S' )</td>
<td></td>
</tr>
<tr>
<td>( isolv(S' = 0) )</td>
<td>( N'-1 = 0 )</td>
<td>( 0 = 0 )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( S' = 0 )</td>
<td>( S' = 0 )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( S = 1 \land S = 1 \land S' = 0 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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
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 isolv(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

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
**Example execution sum(1,S)**

<table>
<thead>
<tr>
<th>Constraint store stack</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>inc_backtrack( sum(1,S) )</td>
<td>&lt;empty&gt;</td>
</tr>
<tr>
<td>inc_backtrack( 1=0, S = 0 )</td>
<td>true</td>
</tr>
<tr>
<td>return false</td>
<td>&lt;empty&gt;</td>
</tr>
<tr>
<td>inc_backtrack( 1=N', S = N' + S', sum(N' -1, S') )</td>
<td>true</td>
</tr>
<tr>
<td>inc_backtrack( S = N' + S', sum(N' -1, S') )</td>
<td>true</td>
</tr>
<tr>
<td>inc_backtrack( sum(N'-1, S') )</td>
<td>true</td>
</tr>
<tr>
<td>inc_backtrack( N'-1 = 0, S' = 0 )</td>
<td>true</td>
</tr>
<tr>
<td>inc_backtrack( S' = 0 )</td>
<td>true</td>
</tr>
<tr>
<td>inc_backtrack( [] )</td>
<td>true</td>
</tr>
</tbody>
</table>

simpl((S), N' = 1 \land S = 1 \land S' = 0) = S = 1

---

**Efficient saving and restoring**

- Incremental solver requires saving/restoring the constraint store
- Don't 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
Trailing

- Associate a timestamp with each primitive constraint
- At a choicepoint
  - store the current timestamp
- Backtracking
  - remove all constraints with a later stamp
- Doesn't handle when an old primitive constraint is modified

Trailing

- Whenever an old constraint (from before the last choicepoint) is modified
  - save the old value in the trail
- Note we don't have to trail the same constraint again if it is modified again before another choicepoint
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 $i \leq last$ then each modified coefficient is added to trail $<i,x,a>$ or $<i,constant,b>$

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
Semantic Backtracking Ex.

Imagine store is
1. \( X = Y + 2Z + 4 \)  
   Removing constraint
2. \( U = 3Y + Z - 1 \)
3. \( V = 3 \)

Adding constraint
\[ Y + 2V + X = 2 \]  
Add coefficient * \((Y+Z+4)\) to eqns 1,2 and remove 4

Eliminate vars
\[ Y = -Z - 4 \]
1. \( X = Z \)
2. \( U = -2Z - 13 \)
3. \( V = 3 \)
4. \( Y = -Z - 4 \)

Eliminate Y using equation and add.

Remember coeffs
\[ [(1,Y,1),(2,Y,3)] \]

Extra Constructs

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

**Cut Example**

Sum program for mode of usage: first arg fixed

\[
\text{sum}(N, SS) ::= \\
(N = 0 \rightarrow SS = 0) \\
; \quad \text{N} \geq 1, SS = N + S, \\
\text{sum}(N-1, S).
\]

Equivalent version with cut

\[
\text{sum}(N, SS) ::= \text{N} = 0, !, SS = 0. \\
\text{sum}(N, SS) ::= \text{N} \geq 1, SS = N + S, \\
\text{sum}(N-1, S).
\]
Cut Derivation Tree

When ! reached, other choices are pruned away.

Cut

- Cut commits to all choices made since when the atom which was rewritten that introduced the cut.
- Assume rewriting atom $A'$ using rule
  - $A ::= L_1, \ldots, L_i, !, L_{i+1}, \ldots, L_n$
- When ! reached all choices for rewriting $A'$ and all choices in evaluation $L_1, \ldots, L_i$ are removed.
**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

**Modifying inc_backtrack**

- **case** $L$ is an atom $p(s_1,...,s_n)$
- **foreach** $p(t_1,...,t_n)$ :- $L_1,...,L_n$ in defn($P,L$)
  - $i :=$ save_store()
  - **if** some $L_j = !$ then
    - **if** inc_backtrack($s_1=t_1,...,s_n=t_n,L_1,...,L_{j-1}$) **then**
      - remove_choicepoints(i)
      - **return** inc_backtrack($L_j+1,...,L_n,G'$)
    - **elseif** inc_backtrack($s_1=t_1,...,s_n=t_n,C'$) **then**
      - **return** true
      - restore_store()
    - **return** **false**
Cut Example 2

\[ h(X) : \neg X > 0, p(X), q(X) \]
\[ h(4) \]
\[ p(X) : \neg X < 4, r(X), !. \]
\[ p(3) \]
\[ r(1) \]
\[ r(2) \]
\[ q(2), q(3) \]

\[ \begin{align*}
\text{Cut Example 2} \\
\text{constraint storestack} \\
\text{inc_backtrack( } h(X) \text{ )} \\
\text{inc_backtrack( } X > 0, p(X), q(X) \text{ )} \\
\text{inc_backtrack( } p(X), q(X) \text{ )} \\
\text{inc_backtrack( } X < 4, r(X) \text{ ) (before cut)} \\
\text{inc_backtrack( } r(X) \text{ )} \\
\text{return true} \\
\text{return false} \\
\text{inc_backtrack( } q(X) \text{ ) (after cut)} \\
\text{return false} \\
\text{inc_backtrack( } X = 4 \text{ )} \\
\text{return true} \\
\end{align*} \]

Answer: \( X = 4 \)
If-Then-Else, Once+Negation

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

\[
\begin{align*}
\text{once}(G) & \leftarrow \text{call}(G), !. \\
\text{not}(G) & \leftarrow \text{call}(G)!, !, \text{fail}. \\
\text{not}(G). \\
G1 \rightarrow G2 \ ; G3 & \leftarrow \text{call}(G1), !, \text{call}(G2). \\
G1 \rightarrow G2 \ ; G3 & \leftarrow \text{call}(G3).
\end{align*}
\]

Optimization

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

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)
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 $E=m,G$

Backtracking Optimization

- case L is minimization literal minimize(G,E)
  - $i := \text{save}_\text{store}()$
  - $m := +\infty$
  - inc_backtrack($G, \text{catch}(m,E)$)
  - 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
    - $m := \text{minimize}_\text{store}(E)$
  - return false
Backtracking Example

\[
\text{inc\_backtrack(butterfly}(S,P),\text{catch}(+, -P))
\]
\[
\text{inc\_backtrack(}\text{catch}(+, -P))
\]
\[
\text{store: } P = -100 \land 0 \leq S \leq 1 \text{ sets } m := 100
\]
\[
\text{inc\_backtrack(}\text{catch}(100, -P))
\]
\[
\text{store: } P = 100S - 200 \land 1 \leq S \leq 3 \text{ sets } m := -100
\]
\[
\text{inc\_backtrack(}\text{catch}(-100, -P))
\]
\[
\text{store: } P = -100S + 400 \land 3 \leq S \leq 5
\]
\[
\text{isolv}(P < -100) \text{ fails no update}
\]
\[
\text{inc\_backtrack(}\text{catch}(100, -P))
\]
\[
\text{store: } P = -100 S >= 5 \text{ fails no update}
\]
\[
\text{inc\_backtrack(-P = -100, butterfly}(S,P))
\]
\[
\text{answers: } P = 100 \land S = 3 \text{ (twice)}
\]

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) \land R
    - return true
### Incremental Tree Solving Ex.

Constraints collected by goal \( \text{append([a], [b, c], L)} \)

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

<table>
<thead>
<tr>
<th>( c )</th>
<th>( S )</th>
<th>( \text{elim}(c) )</th>
<th>( \text{unify}(c) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( [a] = [F</td>
<td>R] )</td>
<td>true</td>
<td>( [a] = [F</td>
</tr>
<tr>
<td>( [b, c] = Y )</td>
<td>( F = a \land R = [] \land Y = [b, c] )</td>
<td>( [b, c] = Y )</td>
<td>( Y = [b, c] )</td>
</tr>
<tr>
<td>( L = [F</td>
<td>Z] )</td>
<td>( F = a \land R = [] \land Y = [b, c] \land L = [a</td>
<td>Z] )</td>
</tr>
<tr>
<td>( R = [] )</td>
<td>( F = a \land R = [] \land Y = [b, c] \land L = [a</td>
<td>Z] \land [b, c] = Z )</td>
<td>( [b, c] = Z )</td>
</tr>
</tbody>
</table>

### 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** \( f(t_1, \ldots, t_n) \): \( n+1 \) memory cells
    - first: constructor info \( f/1 \)
    - rest: pointers to \( t_1, \ldots, t_n \)
  - optimization: store subtrees with no children directly
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 $V$

Data Structures for Trees

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

Represents solved form: $V = g(a) \land W = g(a)$
**Occurs Check Revisited**

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

**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
Incremental Bounds Ex.

Smugglers knapsack, no whiskey

\[
\begin{align*}
\text{capacity} & : 4W + 3P + 2C \leq 9 \land 15W + 10P + 7C \geq 30 \\
D(W) &= [0..0], D(P) = [1..3], D(C) = [0..3]
\end{align*}
\]

Add first constraint
Add second constraint

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