Chapter 6: Using Data Structures

Where we find how to use tree constraints to store and manipulate data

Using Data Structures

- Records
- Lists
- Association Lists
- Binary Trees
- Hierarchical Modelling
- Tree Layout
Records

- simplest type of data structure is a record
- **record** packages together a fixed number of items of information, often of different type
- e.g. *date*(3, Feb, 1997)
- e.g. complex numbers $X + Yi$ can be stored in a record $c(X, Y)$

Complex Numbers

Complex number $X + Yi$ is represented as $c(X,Y)$

Predicates for addition and multiplication

```
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)) :-
```

Note they can be used for subtraction/division
Example Adding $1 + 3i$ to $2 + Yi$

\[
\begin{align*}
C1 &= c(1,3), C2 = c(2,Y), c\_add(C1, C2, C3)|true \\
\downarrow \\
c\_add(C1, C2, C3)|C1 = c(1,3) \land C2 = c(2,Y) \\
\downarrow \\
\begin{align*}
C1 &= c(R1, I1), C2 = c(R2, I2), C3 = c(R3, I3), R3 = R1 + R2, I3 = I1 + I2 \\
&\quad C1 = c(1,3) \land C2 = c(2,Y) \\
\downarrow \\
C3 &= c(R3, I3), R3 = R1 + R2, I3 = I1 + I2 \\
&\quad C1 = c(1,3) \land C2 = c(2,Y) \land R1 = 1 \land I1 = 3 \land R2 = 2 \land I2 = Y \\
\end{align*}
\end{align*}
\]

Simplifying wrt $C3$ and $Y$ gives

\[C3 = c(3,3 + Y)\]
Records

- Term equation can
  - build a record \( C3 = c(R3, I3) \)
  - access a field \( C2 = c(R2, I2) \)
- underscore _ is used to denote an 
  **anonymous variable**, each occurrence is 
different. Useful for record access
  - \( D = date(_, M, _) \) in effect sets \( M \) to equal the 
    month field of \( D \)

Lists

- Lists store a variable number of objects
  usually of the same type.
- empty list [ ] (list)
- list constructor . (item x list -> list)
- special notation:
  
  \[
  \begin{align*}
  & [X|Y] . (X,Y) \\
  [X1, X2,\ldots, Xm|Y] . (X1, (X2, \ldots, (Xm,Y))) \\
  [X1, X2,\ldots, Xm] . (X1, (X2, \ldots, (Xm,[])))
  \end{align*}
  \]
List Programming

- Key: reason about two cases for \( L \)
  - the list is empty \( L = [\] \)
  - the list is non-empty \( L = [F|R] \)
- Example concatenating \( L_1 \) and \( L_2 \) giving \( L_3 \)
  - \( L_1 \) is empty, \( L_3 \) is just \( L_2 \)
  - \( L_1 \) is \( [F|R] \), if \( Z \) is \( R \) concatenated with \( L_2 \) then \( L_3 \) is just \( [F|Z] \)

\[
\text{append}([\], L_2, L_2). \tag{9}
\]
\[
\text{append}([F|R], L_2, [F|Z]) :- \text{append}(R, L_2, Z). \tag{9}
\]

Concatenation Examples

\[
\text{append}([\], L_2, L_2). \tag{9}
\]
\[
\text{append}([F|R], L_2, [F|Z]) :- \text{append}(R, L_2, Z). \tag{9}
\]
- concatenating lists \( \text{append}([1,2],[3,4],L) \)
- has answer \( L = [1,2,3,4] \)
- breaking up lists \( \text{append}(X,Y,[1,2]) \)
- ans \( X=[\] \) \( \land Y=[1,2], X=[1] \) \( \land Y=[2], X=[1,2] \) \( \land Y=[\] \)
- BUT is a list equal to itself plus \([1]\)
- \( \text{append}(L,[1],L) \) runs forever! \( \tag{10} \)
Alldifferent Example

We can program alldifferent using disequations

\[
\text{alldifferent}_\text{neq}([]).
\]

\[
\text{alldifferent}_\text{neq}([Y|Ys]) : -
    \text{not} \_ \text{member}(Y, Ys), \text{alldifferent}_\text{neq}(Ys).
\]

\[
\text{not} \_ \text{member}(\_ , [\]) .
\]

\[
\text{not} \_ \text{member}(X, [Y|Ys]) : -
    X \neq Y, \text{not} \_ \text{member}(X, Ys).
\]

The goal \text{alldifferent}_\text{neq}([A,B,C]) has one solution

\[
A \neq B \land A \neq C \land B \neq C
\]

Arrays

- Arrays can be represented as lists of lists
- e.g. a 6 x 7 finite element description of a metal plate 100C at top edge 0C other edges

\[
[[0, 100, 100, 100, 100, 100, 0],
[0, _, _, _, _, _, 0],
[0, _, _, _, _, _, 0],
[0, _, _, _, _, _, 0],
[0, 0, 0, 0, 0, 0]]
\]
Arrays Example

In a heated metal plate each point has the average temperature of its orthogonal neighbours

\[
\text{rows}([\_\_]).
\]
\[
\text{rows}([R_1,R_2,R_3|R_s]) :-
\]
\[
\text{cols}(R_1,R_2,R_3), \text{rows}([R_2,R_3|R_s]).
\]
\[
\text{cols}([\_\_], [\_\_], [\_\_]).
\]
\[
\text{cols}([T_L,T,T_R|T_s],[L,M,R|M_s],[B_L,B,R|B_s]):-
\]
\[
M = (T + L + R + B)/4,
\]
\[
\text{cols}([T,T_R|T_s],[M,R|M_s],[B,B_R|B_s]).
\]

Arrays Example

The goal \texttt{rows (plate)} constrains \textit{plate} to

\begin{align*}
\begin{bmatrix}
0, & 100, & 100, & 100, & 100, & 100, & 0 \\
0, & 46.6, & 62.5, & 66.4, & 62.5, & 46.6, & 0 \\
0, & 24.0, & 36.9, & 40.8, & 36.9, & 24.0, & 0 \\
0, & 12.4, & 20.3, & 22.9, & 20.3, & 12.4, & 0 \\
0, & 5.3, & 9.0, & 10.2, & 9.0, & 5.3, & 0 \\
0, & 0, & 0, & 0, & 0, & 0, & 0
\end{bmatrix}
\end{align*}
Association Lists

- A list of pairs is an association list
- we can access the pair using only one half of the information
- e.g. telephone book
  
  \[
  [p(peter,5551616),
p(kim, 5559282),
p(nicole, 5559282)]
  \]
- call this phonelist

```
peter 5551616
kim 5559282
nicole 5559282
```

List Membership

\[
\begin{align*}
\text{member}(X, \ [X|\_]). \\
\text{member}(X, \ [\_|R]) & : \neg \text{member}(X, R).
\end{align*}
\]

\(X\) is a member of a list if it is the first element or it is a member of the remainder \(R\)

We can use it to look up Kims phone number

\[
\text{member}(p(kim, N), \ phonelist)
\]

Unique answer: \(N = 5559282\)
Abstract Datatype: Dictionary

- **lookup** information associated with a key
- **newdic** build an empty association list
- **add key** and associated information
- **delete key** and information

lookup(D, Key, Info) :- member(p(Key, Info), D).
newdic([]).
addkey(D0, K, I, D) :- D = [p(K, I) | D0].
delkey([], _, []).
delkey([p(K,_) | D], K, D).
delkey([p(K0, I) | D0], K, [p(K0, I) | D]) :- K != K0, delkey(D0, K, D).
Modelling a Graph

- A directed graph can be thought of as an association of each node to its list of adjacent nodes.

\[
[p(fn,[]), p(iw,[fn]), p(ch,[fn]), p(ew,[fn]), p(rf,[ew]), p(wd,[ew]), p(tl,[ch.rf]), p(dr,[iw])]
\]
call this house

Finding Predecessors

The predecessors of a node are its immediate predecessors plus each of their predecessors

\[
\text{predecessors}(N,D,P) :- \\
\text{lookup}(D,N,NP), \\
\text{list_predecessors}(NP,D,LP), \\
\text{list_append}([NP \mid LP],P).
\]
\[
\text{list_predecessors}([],\_,[]).
\]
\[
\text{list_predecessors}([N|Ns],D,[NP\mid NPs]) :- \\
\text{predecessors}(N,D,NP), \\
\text{list_predecessors}(Ns,D,NPs).
\]
Finding Predecessors

```prolog
list_append([],[]).
list_append([L|Ls], All) :-
    list_append(Ls, A),
    append(L, A, All).
```

Appends a list of lists into one list.

We can determine the predecessors of tiles (tl) using:

```prolog
predecessors(tl, house, Pre)
```

The answer is $Pre = [ch, rf, fn, ew, fn]$

Note repeated discovery of $fn$

Accumulation

- Programs building an answer sometimes can use the list answer calculated so far to improve the computation
- Rather than one argument, the answer, use two arguments, the answer so far, and the final answer.
- This is an **accumulator pair**
**Finding Predecessors**

- A better approach *accumulate* the predcsrs.

```prolog
predecessors(N,D,P0,P) :-
    lookup(D,N,NP),
    cumul_predecessors(NP,D,P0,P).

cumul_predecessors([],_,P,P).

cumul_predecessors([N|Ns],D,P0,P) :-
    member(N,P0),
    cumul_predecessors(Ns,D,P0,P).

cumul_predecessors([N|Ns],D,P0,P) :-
    not_member(N,P0),
    predecessors(N,D,[N|P0],P1),
    cumul_predecessors(Ns,D,P1,P).
```

**Binary Trees**

- empty tree: *null*
- non-empty: *node(t1, i, t2)* where *t1* and *t2* are trees and *i* is the item in the node
- programs follow a pattern (as for lists)
  - a rule for empty trees
  - a recursive rule (or more) for non-empty trees
Binary Trees

\[ \text{node(node(null,p(k,282),null),p(n,282),node(null,p(p,616),null))} \]

A binary tree storing the same info as \textit{phonelist} denote it by \textit{ptree}

Binary Trees

\[
\text{traverse(null,[]).} \\
\text{traverse(node(T1,I,T2),L) :-} \\
\text{traverse(T1,L1),} \\
\text{traverse(T2,L2),} \\
\text{append(L1,[I|L2],L).}
\]

Program to traverse a binary tree collecting items

\[
\text{traverse(}\textit{ptree},L) \\
\text{has unique answer } L = \{p(k,282),p(n,282),p(p,616)\}
\]
**Binary Search Tree**

- **binary search tree** (BST): A binary tree with an order on the items such that for each node$(t1,i,t2)$, each item in $t1$ is less than $i$, and each item in $t2$ is greater than $i$
- previous example is a bst with right order
- another implementation of a dictionary!

---

**Binary Search Tree**

Finding an element in a binary search tree

\[
\text{find(node}(\_\_\_,I\_\_\_),E) :- E = I.
\]
\[
\text{find(node}(L,I\_\_\_),E):-\text{less\_than}(E,I),\text{find}(L,E).
\]
\[
\text{find(node}(\_\_\_,I\_\_\_),E):-\text{less\_than}(I,E),\text{find}(R,E).
\]

Consider the goal \text{find}(ptree, p(k,N)) with definition of \text{less\_than} given below

\[
\text{less\_than}(p(k,\_\_),p(n,\_\_)).
\]
\[
\text{less\_than}(p(k,\_\_),p(p,\_\_)).
\]
\[
\text{less\_than}(p(n,\_\_),p(p,\_\_)).
\]
Binary Search Tree

\[
\{ \text{find}( \text{ptree}, p(k,N)) | \text{true} \} \\
\downarrow \\
\{ \text{less\_than}(p(k,N), p(n,282)), \text{find}(\text{node}(\text{null}, p(k,282), \text{null}), p(k,N)) | \text{true} \} \\
\downarrow \\
\{ \text{find}(p(k,N), \text{node}(\text{null}, p(k,282), \text{null})) | \text{true} \} \\
\downarrow \\
\{ n \mid N = 282 \}
\]

The binary search tree implements a disctionary with logarithmic average time to lookup and add and delete.

Hierarchical Modelling

- Many problems are hierarchical in nature
- Complex objects are made up of collections of simpler objects
- Modelling can reflect the hierarchy of the problem
Hierarchical Modelling Ex.

- steady-state RLC electrical circuits
- sinusoidal voltages and currents are modelled by complex numbers:
- individual circuit elements are modelled in terms of voltages and current:
- circuits are modelled by combining circuit components

Hierarchical Modelling Ex

- Represent voltages and currents by complex numbers: \( V = c(X,Y) \)
- Represent circuit elements by tree with component value: \( E = \text{resistor}(100), \ E = \text{capacitor}(0.1), \ E = \text{inductor}(2) \)
- Represent circuits as combinations or single elements: \( C = \text{parallel}(E1,E2), \ C = \text{series}(E1,E2), \ C = E \)
Hierarchical Modelling Ex.

resistor(R,V,I,_) :- c_mult(I,c(R,0),V).
inductor(L,V,I,W) :- c_mult(c(0,W*L),I,V).
capacitor(C,V,I,W) :- c_mult(c(0,W*C),V,I).
circ(inductor(L),V,I,W) :- inductor(L,V,I,W).
circ(capacitor(C),V,I,W) :- capacitor(C,V,I,W).
circ(parallel(C1,C2),V,I,W) :- c_add(I1,I2,I),
circ(C1,V,I1,W),circ(C2,V,I2,W).
circ(series(C1,C2),V,I,W) :- c_add(V1,V2,V),
circ(C1,V,I1,W),circ(C2,V,I2,W).

The goal
circ(series(parallel(resistor(100),capacitor(0.0001)),
parallel(parallel(inductor(2),resistor(50))),V,I,60).
gives answer
I=c(_t23,_t24)
V=c(-103.8*_t24+52.7*_t23,52.7*_t24+103.8*_t23)
Tree Layout Example

- Drawing a good tree layout is difficult by hand. One approach is using constraints
  - Nodes at the same level are aligned horizontal
  - Different levels are spaced 10 apart
  - Minimum gap 10 between adjacent nodes on the same level
  - Parent node is above and midway between children
  - Width of the tree is minimized

We can write a CLP program that given a tree finds a layout that satisfies these constraints
  - a association list to map a node to coords
  - predicates for building the constraints
  - predicate to calculate width
  - a minimization goal
**Tree Layout Example**

```
node(node(node(node(null,kangaroo,null),marsupial,node(null,koala,null)),mammal,node(null,monotreme,node(null,platypus,null))
),animal,node(node(node(null,cockatoo,null),parrot(node(null,lorikeet,null)),bird,node(null,raptor,node(null,eagle,null))))
```

A good tree layout

```
animal
  mammal
    marsupial
      kangaroo
    monotreme
      platypus
      cockatoo
    parrot
      lorikeet
    raptor
      eagle
```

**Data Structures Summary**

- Tree constraints provide data structures
  - accessing and building in the same manner
- Records, lists and trees are straightforward
- Programs reflect the form of the data struct.
- Association lists are useful data structure for attaching information to objects
- Hierarchical modelling