Theorem Proving versus Programming

• Top-Down Theorem Proving is similar to program execution
• Multiple rules with same head similar to if-else
  - Backtracking to next rule is like evaluating condition in an if-statement, and doing the else block
• Resolution rule is really just procedure call
  - Passing by unification
• Unification is like setting variable values
• Recursion like a while statement
• Basis of Prolog
Numbers

- We could define subtract
- First, pick a simple representation for numbers
  - Represent $X Y Z$ as integers with the successor function
    - $0$ is a number
    - if $X$ is a number, so is $s(X)$
- Define subtract by using successor function
  \[
  \text{subtract}(X,0,X),
  \text{subtract}(X,s(Y),Z) \leftarrow \text{subtract}(X,Y,s(Z))
  \]
- Could just as easily used some other notation
  - binary: $b(1,b(0,b(1,b(1,\text{null}))))$
  - Would need a corresponding definition of subtract

Numerical Calculations

- Might have a rule that needs numerical calculation
  \[
  \text{close}(X,Y) \leftarrow \text{subtract}(X,Y,\text{Diff})
  \land \text{absolute}(\text{Diff},D)
  \land \text{less}(D,5)
  \]
  - $\text{subtract}(X,Y,Z)$ is true for all $X, Y, Z$ where $Z = X - Y$
Cheating on Numbers

- Silly to define arithmetic
- Why not just build it in
- \texttt{is(X,Y)}
  - Evaluate expression \(Y\) and unify it with \(X\)
    - Really easy with Tcl with its \texttt{expr} command
    - Tcl views everything as a list, even program code
  - But will require that \(Y\) is a ground expression
    - Reasoning procedure needs to substitute values in before evaluating it using the \texttt{expr} command, as the variables are not Tcl variables
    - Need to be careful about when atom is evaluated
- This is sort of cheating
  - But very handy!

Problems with Friends

\texttt{friend(tom,sally)}
\texttt{friend(tom,george)}
\texttt{friend(sally,bill)}
...
\texttt{friend(X,Y) \leftarrow friend(Y,X)}

- With top-down resolution, could keep applying rule
  \texttt{friendSub(tom,sally)}
  \texttt{friendSub(tom,george)}
  \texttt{friendSub(sally,bill)}
  \texttt{friend(X,Y) \leftarrow friendSub(X,Y)}
  \texttt{friend(X,Y) \leftarrow friendSub(Y,X)}

- Reformulated rules to work with top-down resolution
Explicit Unification

- Subtract definition:
  
  \[
  \begin{align*}
  \text{subtract}(X,0,X). \\
  \text{subtract}(X,s(Y),Z) & \leftarrow \\
  \text{subtract}(X,Y,s(Z)).
  \end{align*}
  \]

- \(Z\) gets in head gets unified after atom in body is resolved
- Can be easier to understand flow if we write it as: \(\text{subtract}(X,0,X).\)
  
  \[
  \begin{align*}
  \text{subtract}(X,s(Y),NewZ) & \leftarrow \\
  \text{subtract}(X,Y,Z) \\
  Z = s(NewZ)
  \end{align*}
  \]

- ‘\(=\)’ is for explicit unification
  
  - Really just an infix version of the 2-ary predicate ‘\(=\)’,
    which can be defined as ‘\(=(X,X)\)’
  - Not adding any power to Datalog
    - Not saying that two different constants point to same domain object

Why is Representation and Reasoning Neat?

- Represented knowledge about the domain with just rules and a bunch of facts
- Very concise representation: just a few rules can capture a lot of different uses
- Reasoning procedure is separated from knowledge about domain
- We can use an ordinary theorem prover
Overview

• Programming
  ⇒ Recursion

Recursion and Mathematical Induction

• Idea: define a predicate in terms of simpler instances of itself
  \[ \text{length(nil,0)} \]
  \[ \text{length}(\text{p(Top|Rest)},\text{NewL}) \leftarrow \]
  \[ \text{length}(\text{Rest},\text{L}) \land \]
  \[ \text{NewL is L + 1}. \]

• Recursion works by having
  - a well-founded ordering of instances of relations
  - each element is defined in terms of elements lower in the ordering
  - and each decreasing chain eventually reaches an element that is simplest
    in the ordering – defined by a clause with no body
Recursive Data Structure

- Recursion is only 'looping construct' in Datalog
- Useful to have recursive data structure
  - Process top element of data structure on each level of the recursion and pass rest of the structure during recursion
  - Build up a data structure on way out of recursion on way out of recursion, add top element to structure
- Example:
  - What is done on way into recursion?
  - What is done on way out?

\[
\text{length}(\text{nil},0).
\]
\[
\text{length}(\text{p}(\text{Top},\text{Rest}),\text{NewL}) \leftarrow \text{length}(\text{Rest},L) \land \text{NewL is } L+1.
\]

Two ways of view Theorem Proving: Answer Clause

\[
\text{length}(\text{nil},0).
\]
\[
\text{length}(\text{p}(\text{Top},\text{Rest}),\text{NewL}) \leftarrow \text{length}(\text{Rest},L) \land \text{NewL is } L+1.
\]

?\text{length}(\text{p}(5,\text{p}(3,\text{p}(1,\text{nil}))),L)
\]
\[
\text{yes}(1) \leftarrow \text{length}(\text{p}(5,\text{p}(3,\text{p}(1,\text{nil}))),L)
\]
\[
\text{renamed: yes}(L1) \leftarrow \text{length}(\text{p}(5,\text{p}(3,\text{p}(1,\text{nil}))),L1)
\]

\[
\text{yes}(L1) \leftarrow \text{length}(\text{p}(3,\text{p}(1,\text{nil})),L) \land \text{is}(L1,L+1)
\]
\[
\text{renamed: yes}(L1) \leftarrow \text{length}(\text{p}(3,\text{p}(1,\text{nil})),L1) \land \text{is}(L1,L+1)
\]

\[
\text{yes}(L1) \leftarrow \text{length}(\text{p}(1,\text{nil})),L) \land \text{is}(L1,L+1)
\]
\[
\text{renamed: yes}(L1) \leftarrow \text{length}(\text{p}(1,\text{nil})),L) \land \text{is}(L1,L+1)
\]

\[
\text{yes}(L1) \leftarrow \text{is}(L1,2+1)
\]
\[
\text{yes}(3)
\]
Concatenating lists

- Might want a predicate $\text{concat}(L_1, L_2, L_3)$
  - This is similar to python’s `+` for lists
  - True if $L_3$ is list of elements of $L_1$ followed by elements of $L_2$
  - Useful for verifying whether $L_3$ is $L_1$ appended in front of $L_2$
  - But also should be useful for appending two lists together:
    - $\text{concat}([\text{tim}, \text{nil}], [\text{john}, \text{phil}, \text{ted}, \text{nil}])$, $L_3$
    - Or finding the prefix of a list of given ending
      - $\text{concat}([\text{L1}, \text{p(c, p(d, nil))}], \text{p(a, p(b, p(c, p(d, nil))))})$
  - How do we define $\text{concat}(L_1, L_2, L_3)$?
    - Only tricky part is which of $L_1$ and $L_2$ to attack
    - This is determined by how lists are represented

Two ways of view Theorem Proving: Recursive Calls

\[
\text{length(nil,0).} \\
\text{length(p(Top,Rest),NewL) ← length(Rest,L) ^ is(NewL,L+1)}
\]

Solve: $\text{length(p(5,p(3,p(1,nil)))),L}$
Use: $\text{length(p(5,p(3,p(1,nil)))),NewL) ← length(p(3,p(1,nil))),L ^ is(NewL,L+1)}$
Solve: $\text{length(p(3,p(1,nil))),L}$
Use: $\text{length(p(3,p(1,nil))),NewL) ← length(p(1,nil)),L ^ is(NewL,L+1)}$
Solve: $\text{length(p(1,nil)),L}$
Use: $\text{length(p(1,nil),NewL) ← length(nil,L) ^ is(NewL,L+1)}$
Solved: $\text{is(NewL,0+1)}$
Solved: $\text{is(1,0+1)}$
Solved: $\text{length(p(1,nil),1)}$
Solve: $\text{is(NewL,1+1)}$
Solved: $\text{is(2,1+1)}$
Solved: $\text{length(p(3,p(1,nil))),2}$
Solve: $\text{is(NewL,2+1)}$
Solved: $\text{is(3,2+1)}$
Solved: $\text{length(p(5,p(3,p(1,nil)))),3}$
• Since lists are defined with new elements on front, we need to do recursion on first part, so we can successively add to the front

• With/without explicit unification

\[
\text{concat}(p(Top,Rest),Second,NewList) \leftarrow \\
\text{concat}(Rest,Second,RestSecond), \\
\text{NewList} = p(Top,RestSecond)
\]

\[
\text{concat}(p(Top,Rest),Second,p(Top,RestSecond)) \leftarrow \\
\text{concat}(Rest,Second,RestSecond).
\]

• How does the following work with the above versions?

\[
\text{concat}(X,Y,p(3,p(2,p(1,nil))))
\]
Example: Path through a Maze

- path of 1 2 6 10 could be represented
  - \( p(1, p(2, p(6, p(10, \text{nil}))) \)
  - or \( p(10, p(6, p(2, p(1, \text{nil})))) \)
- \( p \): arbitrary function name to glue path together
- \( \text{nil} \): arbitrary symbol to denote empty path

- To get first part of path and remaining, unify with \( p(\text{Top}, \text{Rest}) \)

- Recursive rule to check if a cell is used in the path
  \[ \text{member}(S, p(S, X)) \]
  \[ \text{member}(S, p(X, R)) \leftarrow \text{member}(S, R) \]
Toward Building Recursive Structures

- Example: path through a maze
  
  \[
  \begin{array}{cccc}
  1 & 2 & 3 & 4 \\
  5 & 6 & 7 & 8 \\
  9 & 10 & 11 & 12 \\
  13 & 14 & 15 & 16 \\
  \end{array}
  \]

- Finding out if there is a path
  
  \[
  \text{path}(X,Y) \iff \text{connected}(X,Y). \\
  \text{path}(X,Y) \iff \text{connected}(X,Z), \text{path}(Z,Y). \\
  \text{?- path}(1,16).
  \]

Proof Derivation

\[
\begin{align*}
? \text{path}(1,16) \\
\text{yes} & \leftarrow \text{path}(1,16) \\
& \text{use: } \text{path}(X_1,Y_1) \leftarrow \text{connected}(X_1,Z_1) \land \text{path}(Z_1,Y_1) \\
& \text{sub: } X_1/1, Y_1/16 \\
\text{yes} & \leftarrow \text{connected}(1,Z_1) \land \text{path}(Z_1,16) \\
& \text{use: } \text{connected}(X_2,Y_2) \leftarrow \text{connectedSub}(X_2,Y_2) \\
& \text{sub: } X_2/1, Y_2/Z_1 \\
\text{yes} & \leftarrow \text{connectedSub}(1,Z_1) \land \text{path}(Z_1,16) \\
& \text{use: } \text{connectedSub}(1,2) \\
& \text{sub: } Z_1/2 \\
\text{yes} & \leftarrow \text{path}(2,16) \\
\end{align*}
\]
Building Recursive Structures

• How can we remember the path?
• Building path on way out of recursion

\[\text{path}(X, X, p(X, nil)).\]
\[\text{path}(X, Y, p(X, PathZtoY)) \leftarrow \text{connected}(X, Z) \land \text{path}(Z, Y).\]

?- \text{path}(1, 16, \text{Path})
Path = p(1, p(2, p(6, p(10, p(11, p(7, p(8, p(12, p(16, nil))))))))).

Does code really work?

... yes \leftarrow \text{path}(2,16)
use: \text{path}(X_3,Y_3) \leftarrow \text{connected}(X_3,Z_3) \land \text{path}(Z_3,Y_3)
sub: \text{X}_3/2 \text{ Y}_3/16
yes \leftarrow \text{connectedSub}(2,Z_3) \land \text{path}(Z_3,16)
use: \text{connectedSub}(2,3)
sub: \text{Z}_3/3
yes \leftarrow \text{path}(3,16)
...

• Depth first search will make sure any possible path is explored
• But might end up in an endless cycle
  + Need to make sure we do not visit a cell already in the path
  + Will come back to this issue
Why is this building the path on way out?

Path is being accumulated in the answer atom, but not available to the path clause.

path(X,Y) ← connected(X,Y) ∧ path(X,Z) ∧ path(Z,Y)
Proof Derivation (2nd Version)

\[ \text{path}(1,3, \text{Path}) \]
\[ \text{yes}(\text{Path}) \leftarrow \text{path}(1,3, \text{Path}) \]
\[ \text{use: } \text{path}(X_1,Y_1,P_1) \leftarrow \text{connected}(X_1,Z_1) \land \text{path}(Z_1,Y_1, P_{YZ1}) \land \]
\[ P_1 = \text{path}(X_1, Path_{YZ1}) \]
\[ \text{sub: } X_1/1 Y_1/3 P_1/\text{Path} \]
\[ \text{yes}(\text{Path}) \leftarrow \text{connected}(1,Z_1) \land \text{path}(Z_1,3, P_{YZ1}) \land \text{Path}=\text{path}(1, Path_{YZ1}) \]
\[ \text{use: } \text{connected}(X_2,Y_2) \leftarrow \text{connectedSub}(X_2,Y_2) \]
\[ \text{sub: } X_2/1 Y_2/2 Z_1 \]
\[ \text{yes}(\text{Path}) \leftarrow \text{connectedSub}(1,Z_1) \land \text{path}(Z_1,3, P_{YZ1}) \land \text{Path}=\text{path}(1, Path_{YZ1}) \]
\[ \text{use: } \text{connectedSub}(1,2) \]
\[ \text{sub: } Z_1/2 \]
\[ \text{yes}(\text{Path}) \leftarrow \text{path}(2,3, P_{YZ1}) \land \text{Path}=\text{path}(1, Path_{YZ1}) \]
\[ \text{use: } \text{path}(X_3,Y_3,P_3) \leftarrow \text{connected}(X_3,Z_3) \land \text{path}(Z_3,Y_3, P_{YZ3}) \land \]
\[ P_3 = \text{path}(X_3, Path_{YZ3}) \]
\[ \text{sub: } X_3/2 Y_3/3 P_3/Path_{YZ3} \]
\[ \text{yes}(\text{Path}) \leftarrow \text{connected}(2,Z_1) \land \text{path}(Z_1,3, P_{YZ1}) \land \text{Path}_{YZ1}=\text{path}(X_3, Path_{YZ3}) \]
\[ \land \text{Path}=\text{path}(1, Path_{YZ1}) \]

Building Path on Way **In**

- Build the current path before the recursive call
  - On way out
    \[ \text{path}(X,Y, \text{Path}) \leftarrow \text{connected}(X,Z) \land \text{path}(Z,Y, Path_{ZtoY}) \land \text{Path} = p(X, Path_{ZtoY}) \]
  - On way in
    \[ \text{path}(X,Y, \text{PathToX}) \leftarrow \text{connected}(X,Z) \land \text{PathToZ} = p(Z, PathToX) \land \text{path}(Z,Y, PathToZ) \]

- For ‘way in’ version, how is answer returned to user?

- Do both versions build the same path?
Reversing a list

• \textit{rev}(L, R) \text{ true if } R \text{ is the reversal of list } L

\# Good place to start is with list represented as head and tail
\text{rev}(p(\text{Head}, \text{Tail}), R) \leftarrow

\# now to reverse the tail
\text{rev}(\text{Tail}, TR)

\# now to stick head after the reversal of the tail
\text{append}(TR, p(\text{Head}, \text{nil}), R).

\text{rev}(\text{nil}, \text{nil}).

- Not very efficient though, as we append after each step
- Number of steps in proof $O(n^2)$

• Can we do better?

Efficiently Reversing a List

• Let’s take advantage of what we learned from building paths on way \textit{in} versus \textit{out} of the recursion

• To reverse a list, we can
  - pull off elements of the list on way into recursion
  - and put them onto another list on way into recursion

• Like washing a stack of dishes
  - Take top one off, wash it, and put it on the top of the clean dishes
Building Lists

• Define $\text{rev3}(J,K,L)$
  - where $J$ is remainder of list to be reversed
  - $K$ is the reversal of the list that has been reversed so far
  - $L$ will be used to return the list at the bottom of the recursion

• Definition:
  $\text{reverse}(J,L) \leftarrow \text{rev3}(J,\text{nil},L)$.
  \# now define $\text{rev3}$ recursively in terms of a smaller version of itself
  $\text{rev3}(\text{p(Head,Tail)},K,L) \leftarrow$
  $\text{rev3}(\text{Tail, p(Head,K)}, L)$
  \# now the base case
  $\text{rev3}(\text{nil, L, L})$.

Definition

• Define $\text{rev3}(J,K,L)$
  - where $J$ is remainder of list to be reversed
  - $K$ is the reversal of the list that has been reversed so far
  - $L$ will be used to return the list at the bottom of the recursion

• Definition:
  $\text{reverse}(J,L) \leftarrow \text{rev3}(J,\text{nil},L)$.
  \# now define $\text{rev3}$ recursively in terms of a smaller version of itself
  $\text{rev3}(\text{p(Head,Tail)},K,L) \leftarrow$
  $\text{rev3}(\text{Tail, p(Head,K)}, L)$
  \# now the base case
  $\text{rev3}(\text{nil, L, L})$. 

Append:

- $\text{append}(\text{p(Head,Tail)}, L, \text{p(Head,R)}) \leftarrow$
  $\text{append}(\text{Tail, L, R})$
  $\text{append}(\text{nil, L, L})$.
  - on way into recursion, tear off top element
  - on way out, put at top of list

• $\text{rev3}$:
  $\text{rev3}(\text{p(Head,Tail)}, L2,L3) \leftarrow$
  $\text{rev3}(\text{Tail, p(Head,L2)}, L3)$
  $\text{rev3}(\text{nil, L2,L2})$.
  - on way into recursion, tear off top element and put on top of new list
  - pass back completed list on way out