Overview

⇒ Programming
• Recursion
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
Numerical Calculations

- Might have a rule that needs numerical calculation
  \[ close(X, Y) \leftarrow subtract(X, Y, \text{Diff}) \]
  \[ \land \ absolute(\text{Diff}, D) \]
  \[ \land \ less(D, 5) \]

- \( subtract(X, Y, Z) \) is true for all \( X, Y, Z \) where \( Z = X - Y \)
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
Cheating on Numbers

• Silly to define arithmetic
• Why not just build it in
• \( is(X, Y) \)
  - 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

friend(tom,sally)
friend(tom,george)
friend(sally,bill)
...

friend(X,Y) ← friend(Y,X)

• With top-down resolution, could keep applying rule

friendSub(tom,sally)
friendSub(tom,george)
friendSub(sally,bill)
...

friend(X,Y) ← friendSub(X,Y)
friend(X,Y) ← friendSub(Y,X)

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

• Subtract definition:
  
  \[ \text{subtract}(X,0,X) \]
  
  \[ \text{subtract}(X,s(Y),Z) \leftarrow \text{subtract}(X,Y,s(Z)). \]

  - Z gets in head gets unified after atom in body is resolved
  - Can be easier to understand flow if we write it as: \( \text{subtract}(X,0,X) \).

    \[ \text{subtract}(X,s(Y),NewZ) \leftarrow \text{subtract}(X,Y,Z) \]
    
    \[ Z = s(NewZ) \]

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

• 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?

\[
\text{length(nil,0).} \\
\text{length(p(Top,Rest),NewL) } \leftarrow \\
\quad \text{length(Rest,L) } \land \\
\quad \text{NewL is } L + 1.
\]
Two ways of view Theorem Proving: Answer Clause

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

\text{?length(p(5,p(3,p(1,nil))),L)}
\text{yes(L) \leftarrow length(p(5,p(3,p(1,nil))),L)}
\quad \text{renamed: yes(L1) \leftarrow length(p(5,p(3,p(1,nil))),L)}
\text{yes(L1) \leftarrow length(p(3,p(1,nil)),L) \land \text{is(L1,L+1)}}
\quad \text{renamed: yes(L1) \leftarrow length(p(3,p(1,nil)),L2) \land \text{is(L1,L2+1)}}
\text{yes(L1) \leftarrow length(p(1,nil),L) \land \text{is(L2,L+1)} \land \text{is(L1,L2+1)}}
\quad \text{renamed: yes(L1) \leftarrow length(p(1,nil)),L3) \land \text{is(L2,L3+1)} \land \text{is(L1,L2+1)}}
\text{yes(L1) \leftarrow length(nil,L) \land \text{is(L3,L+1)} \land \text{is(L2,L3+1)} \land \text{is(L1,L2+1)}}
\text{yes(L1) \leftarrow is(L3,0+1) \land \text{is(L2,L3+1)} \land \text{is(L1,L2+1)}}
\text{yes(L1) \leftarrow is(L2,1+1) \land \text{is(L1,L2+1)}}
\text{yes(L1) \leftarrow is(L1,2+1)}
\text{yes(3)}
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: length(p(5,p(3,p(1,nil))),L)
  Use: length(p(5,p(3,p(1,nil))),NewL) <- length(p(3,p(1,nil)),L) ^ is(NewL,L+1)
    Solve: length(p(3,p(1,nil)),L)
      Use: length(p(3,p(1,nil)),NewL) <- length(p(1,nil),L) ^ is(NewL,L+1)
        Solve: length(p(1,nil),L)
          Use: length(p(1,nil),NewL) <- length(nil,L) ^ is(NewL,L+1)
            Solve: length(nil,L)
              Solved: length(nil,0)
            Solve: is(NewL,0+1)
              Solved: is(1,0+1)
              Solved: length(p(1,nil),1)
            Solve: is(NewL,1+1)
              Solved: is(2,1+1)
              Solved: length(p(3,p(1,nil)),2)
            Solve: is(NewL,2+1)
              Solved: is(3,2+1)
              Solved: length(p(5,p(3,p(1,nil))),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:
    + $\text{concat}(p(\text{tim}, \text{nil}), p(\text{john}, p(\text{phil}, p(\text{ted}, \text{nil})))), L_3)$
  - Or finding the prefix of a list of given ending
    + $\text{concat}(L_1, p(c, p(d, \text{nil})), p(a, p(b, p(c, p(d, \text{nil}))))))$

• 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
• 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

\[
\begin{align*}
\text{concat}(p(Top, Rest), Second, NewList) & \leftarrow \\
& \text{concat}(Rest, Second, RestSecond), \\
& \text{NewList} = p(Top, RestSecond) \\
\end{align*}
\]

\[
\begin{align*}
\text{concat}(p(Top, Rest), Second, p(Top, RestSecond)) & \leftarrow \\
& \text{concat}(Rest, Second, RestSecond). \\
\end{align*}
\]

• How does the following work with the above versions?

\[
\text{concat}(X, Y, p(3, p(2, p(1, nil))))
\]
More Concat

\( \text{concat}(\text{nil}, Z, Z). \)
\( \text{concat}(p(T, \text{Rest}), \text{Second}, \text{NewList}) \leftarrow \text{concat}(\text{Rest}, \text{Second}, \text{RestSecond}) \land \text{NewList} = p(T, \text{RestSecond}) \)

- What is done on way into recursion? (before base case)
- What is done on way out of recursion? (after base case)
Continued

\[
\begin{align*}
yes(X) & \leftarrow con(p(a,p(b,nil)),p(c,nil),X) \\
& \quad \text{use } con(p(Top,Rest),Y,NList) \leftarrow con(Rest,Y,NRest) \land NList = p(Top,NRest) \\
& \quad \quad \quad Top/a \ Rest/p(b,nil) \ Y/p(c,nil) \ NList/X
\end{align*}
\]

\[
\begin{align*}
yes(X) & \leftarrow con(p(b,nil),p(c,nil),NRest1) \land X=p(a,NRest1) \\
& \quad \text{use } con(p(Top,Rest),Y,NList) \leftarrow con(Rest,Y,NRest) \land NList = p(Top,NRest) \\
& \quad \quad \quad Top/b \ Rest/nl Y/p(c,nil) \ NList/NRest1
\end{align*}
\]

\[
\begin{align*}
yes(X) & \leftarrow con(nil,p(c,nil),NRest) \land NRest1=p(b,NRest) \land X=p(a,NRest1) \\
& \quad \text{use } concat(nil,Z,Z) \\
& \quad \quad \quad Z/p(c,nil) \ NRest/p(c,nil)
\end{align*}
\]

\[
\begin{align*}
yes(X1) & \leftarrow NRest1=p(b,p(c,nil)) \land X1=p(a,NRest1) \\
& \quad \text{use } =(X,X) \\
& \quad \quad \quad X/p(b,p(c,nil)) \ NRest1/p(b,p(c,nil))
\end{align*}
\]

\[
\begin{align*}
yes(X1) & \leftarrow X1=p(a,p(b,p(c,nil))) \\
& \quad \text{use } =(X,X) \\
& \quad \quad \quad X1/p(a,p(b,p(c,nil))) \ X/p(a,p(b,p(c,nil)))
\end{align*}
\]

\[
\begin{align*}
yes(p(a,p(b,p(c,nil))))
\end{align*}
\]
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
  connectedSub(1,2).
  connectedSub(2,3).
  connectedSub(2,6).
  connectedSub(4,8).
  ...
  connected(X,Y) :- connectedSub(X,Y).
  connected(X,Y) :- connectedSub(Y,X).

• How do we find if there is a path?
Toward Building Recursive Structures

- Example: path through a maze
  connectedSub(1,2).
  connectedSub(2,3).
  connectedSub(2,6).
  connectedSub(4,8).
  
  connected(X,Y) :- connectedSub(X,Y).
  connected(X,Y) :- connectedSub(Y,X).

- Finding out if there is a path
  path(X,Y) :- connected(X,Y).
  path(X,Y) :- connected(X,Z), path(Z,Y).

?- path(1,16).

© P. Heeman, 2020
Proof Derivation

?path(1,16)

yes ← path(1,16)
  use: path(X₁,Y₁) ← connected(X₁,Z₁) ∧ path(Z₁,Y₁)
  sub: X₁/1 Y₁/16

gen(1,16)

yes ← connected(1,Z₁) ∧ path(Z₁,16)
  use: connected(X₂,Y₂) ← connectedSub(X₂,Y₂)
  sub: X₂/1 Y₂/Z₁

yes ← connectedSub(1,Z₁) ∧ path(Z₁,16)
  use: connectedSub(1,2)
  sub: Z₁/2

gen(1,16)

...
Does code really work?

... yes ← path(2,16)
    use: path(X₃,Y₃) ← connected(X₃,Z₃) ∧ path(Z₃,Y₃)
    sub: X₃/2 Y₃/16

yes ← connectedSub(2,Z₃) ∧ path(Z₃,16)
    use: connectedSub(2,3)
    sub: Z₃/3

yes ← path(3,16)
...

... yes ← path(2,16)
...

• 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
Building Recursive Structures

• How can we remember the path?

• Building path on way out of recursion

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

?- path(1, 16, Path) Path = p(1, p(2, p(6, p(10, p11, p(7, p(8, p(12, p(16, nil)))))))))
Proof Derivation

?path(1,3,Path)
yes(Path) ← path(1,3,Path)
  use: path(X_1,Y_1,p(X_1,PathZY_1)) ← connected(X_1,Z_1) ∧ path(Z_1,Y_1,PathZY_1)
  sub: X_1/1 Y_1/3 Path/p(1,PathZY_1)
yes(p(1,PathZY_1)) ← connected(1,Z_1) ∧ path(Z_1,3,PathZY_1)
  use: connected(X_2,Y_2)← connectedSub(X_2,Y_2)
  sub: X_2/1 Y_2/Z_1
yes(p(1,PathZY_1)) ← connectedSub(1,Z_1) ∧ path(Z_1,3,PathZY_1)
  use: connectedSub(1,2)
  sub: Z_1/2
yes(p(1,PathZY_1)) ← path(2,3,PathZY_1)
  use: path(X_3,Y_3,p(X_3,PathZY_3)) ← connected(X_3,Z_3) ∧ path(Z_3,Y_3,PathZY_3)
  sub: X_3/2 Y_3/3 PathZY_1/p(2,PathZY_3)
yes(p(1,p(2,PathZY_3))) ← connected(2,Z_3) ∧ path(Z_3,3,PathZY_3)
...
Why is this building the path on way out?

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

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

\[
\text{path}(X,Y,\text{Path}) \leftarrow \text{connected}(X,Z) \\
\text{path}(Z,Y,\text{PathZtoY}) \\
\text{Path} = p(X,\text{PathZtoY})
\]
Proof Derivation (2nd Version)

?path(1,3,Path)
yes(Path) ← path(1,3,Path)
  use: path(X₁,Y₁,P₁) ← connected(X₁,Z₁) \land path(Z₁,Y₁,PathZY₁) \land
  P₁=path(X₁,PathZY₁)
  sub: X₁/1 Y₁/3 P₁/Path
yes(Path) ← connected(1,Z₁) \land path(Z₁,3,PathZY₁) \land Path=path(1,PathZY₁)
  use: connected(X₂,Y₂)← connectedSub(X₂,Y₂)
  sub: X₂/1 Y₂/Z₁
yes(Path) ← connectedSub(1,Z₁) \land path(Z₁,3,PathZY₁) \land Path=path(1,PathZY₁)
  use: connectedSub(1,2)
  sub: Z₁/2
yes(Path) ← path(2,3,PathZY₁) \land Path=path(1,PathZY₁)
  use: path(X₃,Y₃,P₃) ← connected(X₃,Z₃) \land path(Z₃,Y₃,PathZY₃) \land
  P₃=path(X₃,PathZY₃)
  sub: X₃/2 Y₃/3 P₃/PathZY₁)
yes(Path) ← connected(2,Z₃) \land path(Z₃,3,PathZY₃) \land PathZY₁=path(X₃,PathZY₃
  \land Path=path(1,PathZY₁)
...
Building Path on Way In

- Build the current path before the recursive call

  - On way out
    \[ \text{path}(X,Y,\text{Path}) \leftarrow \text{connected}(X,Z) \]
    \[ \text{path}(Z,Y,\text{PathZtoY}) \]
    \[ \text{Path} = p(X,\text{PathZtoY}) \]

  - On way in
    \[ \text{path}(X,Y,\text{PathToX}) \leftarrow \text{connected}(X,Z) \]
    \[ \text{PathToZ} = p(Z,\text{PathToX}) \]

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

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

• $rev(L, R)$ true if $R$ is the reversal of list $L$

  # Good place to start is with list represented as head and tail
  $rev(p(Head, Tail), R) \leftarrow$
Reversing a list

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

  \[
  \text{# Good place to start is with list represented as head and tail}
  \]

  \[
  \text{rev}(p(\text{Head}, \text{Tail}), R) \leftarrow
  \]

  \[
  \text{# now to reverse the tail}
  \]
Reversing a list

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

\# Good place to start is with list represented as head and tail
\textit{rev}(\textit{p(Head, Tail)}, R) \leftarrow
\begin{align*}
\# & \textit{now to reverse the tail} \nonumber \\
\textit{rev}(\textit{Tail, TR}) & \nonumber
\end{align*}
Reversing a list

- \( \text{rev}(L, R) \) true if \( R \) 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
Reversing a list

- \( rev(L, R) \) true if \( R \) is the reversal of list \( L \)

  # Good place to start is with list represented as head and tail
  \[
  rev(p(Head, Tail), R) \leftarrow \\
  \quad \# \text{ now to reverse the tail} \\
  \quad rev(Tail, TR) \\
  \quad \# \text{ now to stick head after the reversal of the tail} \\
  \quad append(TR, p(Head, nil), R).
  \]

  \[
  rev(nil, nil).
  \]

- 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 in versus 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
Definition

• Define $rev3(J,K,L)$
  - 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:

  $reverse(J,L) \leftarrow rev3(J,nil,L)$.
Definition

• Define \( rev3(J,K,L) \)
  - 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:

\[
\text{reverse}(J,L) \leftarrow rev3(J,nil,L).
\]

# now define \( rev3 \) recursively in terms of a smaller version of itself
• 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:

$$\text{reverse}(J,L) ← \text{rev3}(J,\text{nil},L).$$

# now define $\text{rev3}$ recursively in terms of a smaller version of itself

$$\text{rev3}(p(\text{Head},\text{Tail}),K,L) ←$$
$$\text{rev3}(\text{Tail},p(\text{Head},K),L)$$

# now the base case
• Define $rev3(J,K,L)$
  - 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:
  
  \[
  reverse(J,L) \leftarrow rev3(J,nil,L).
  \]

  # now define $rev3$ recursively in terms of a smaller version of itself

  \[
  rev3(p(Head,Tail),K,L) \leftarrow
  \]

  \[
  rev3(Tail,p(Head,K),L)
  \]

  # now the base case

  \[
  rev3(nil,L,L).
  \]
Building Lists

- **Append:**
  
  \[
  \text{append}(p(\text{Head}, \text{Tail}), L, p(\text{Head}, R)) \leftarrow \\
  \text{append}(\text{Tail}, L, R) \\
  \text{append}(\text{nil}, L, L).
  \]

  - on way into recursion, tear off top element
  - on way out, put at top of list

- **rev3:**
  
  \[
  \text{rev3}(p(\text{Head}, \text{Tail}), L2, L3) \leftarrow \\
  \text{rev3}(\text{Tail}, p(\text{Head}, L2), L3) \\
  \text{rev3}(\text{nil}, L2, L2).
  \]

  - on way into recursion, tear off top element and put on top of new list
  - pass back completed list on way out