Numerical Calculations

- Might have a rule that needs numerical calculation
  
  close(X,Y) ← subtract(X,Y,Diff) ∧ absolute(Diff,D) ∧ less(Diff,5)

  - `subtract(X,Y,Z)` is true for all $X, Y, Z$ where $Z = X - Y$.

Theorem Proving versus Programming

- Top-Down Theorem Proving is similar to program execution:
  - Multiple rules with same head similar to if-else
  - Resolution rule is really just procedure call, and does not need evaluation condition
  - Recursion like a while statement

Overview

- Recursion = Programming
Problems with Friends

With top-down resolution, could keep applying rule

\[ \text{friend}(X,Y) \leftarrow \text{friend}(Y,X) \]

• Reformulated rules to work with top-down resolution

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

Cheating on Numbers

We could define subtraction
• First, pick a simple representation for numbers
• Represent \( X \) as integers with the successor function
• \( 0 \) is a number
• If \( X \) is a number, so is \( s(X) \)
• \( \text{is}(X,Y) \) - \text{successor}\( X \)
• Define \text{successor} function
• \( X \) is a number
• \( s(X) \) is a number
• First, pick a simple representation for numbers
• We could define subtraction

Numbers
Overview

We can use an ordinary theorem prover.

Reasoning procedure is separated from knowledge about domain.

Very concise representation: just a few rules can capture a lot of

Reasoned knowledge about the domain with just rules and a

Why is Representation and Reasoning Neat?

Explicit Unification

We subtract the definition:

 subtract(X,0,X).
 subtract(X,s(Y),Z) ← subtract(X,Y ,s(Z)).

Z gets head gets unified after atom in body is resolved

Can be easier to understand the flow if we write it as:

 subtract(X,0,X).
 subtract(X,s(Y),NewZ) ← subtract(X,Y ,Z)

Z = s(NewZ)

'=' is for explicit unification

really just an infix version of the 2-ary predicate, =,

Is for explicit unification

(X/a)(Z/c) = Z

Z(X/c)\(\text{substitute}(X,a))

\(\text{substitute}(Z,c))\(\text{substitute}(X,a))

\text{can be cases in understanding how it work it as substitute}\n
\text{Z gets in head gets unified after atom in body is resolved,}

\text{cases in head gets unified after atom in body is resolved,}

\text{Subsequent definition:}
Concatenating lists
• Might want a predicate `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:
    
    ```
    +
    concat(p(tim, nil), p(john, p(phil, p(ted, nil)))), L_3)
    ```
  - Or finding the prefix of a list of given ending
    
    ```
    +
    concat(L_1, p(c, p(d, nil)), p(a, p(b, p(c, p(d, nil))))))
    ```

• How do we define `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

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, add top element to structure
• Example:
  - What is done on way into recursion?
  - What is done on way out?

Recursive Data Structure

<table>
<thead>
<tr>
<th>Recursion and Mathematical Induction</th>
</tr>
</thead>
<tbody>
<tr>
<td>How does recursion work by having the recursive function defined in terms of simpler functions?</td>
</tr>
</tbody>
</table>

Recursion works by having the recursive function defined in terms of simpler functions.

Example: Path through a Maze

- path of 1 2 6 10 could be represented
  - \( p(1, p(2, p(6, p(10, \text{nil})))) \)
  - \( p(10, p(6, p(2, p(1, \text{nil})))) \)
  - \( p \): 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
  
  \[
  \text{member}(S, p(S, X)) \leftarrow \text{member}(S, p(X, R)) \]
  
  \[
  \text{member}(S, p(R, X)) \leftarrow \text{member}(S, R) \]

- yes(X) \leftarrow \text{con}(p(a, p(b, \text{nil})), p(c, \text{nil}), X)

- use \( \text{con}(p(Top, Rest), Y, N\text{List}) \leftarrow \text{con}(Rest, Y, N\text{Rest}) \land N\text{List} = p(Top, N\text{Rest}) \)

- yes(X) \leftarrow \text{con}(p(b, \text{nil}), p(c, \text{nil}), N\text{Rest}1) \land X = p(a, N\text{Rest}1)

- use \( \text{concat}(\text{nil}, Z, Z) \)

- yes(X1) \leftarrow N\text{Rest}1 = p(b, p(c, \text{nil})) \land X1 = p(a, N\text{Rest}1)

- use \( = (X, X) \)

- yes(p(a, p(b, p(c, \text{nil})))

- More Concider
Depth first search will make sure any possible path is explored

\[ \text{depth-first search} \]

- Need to make sure we do not get stuck in an endless cycle

\[ \text{path}(2,16) \]

\[ \text{use: path}(X, Y) \]

\[ \text{connected}(X, Z) \land \text{path}(Z, Y) \]

\[ \text{sub: } X/3, Y/16 \]

\[ \text{yes} \]

\[ \text{connectedSub}(2, Z) \land \text{path}(Z, 16) \]

\[ \text{use: connectedSub}(2, 3) \]

\[ \text{sub: } Z/3 \]

\[ \text{yes} \]

\[ \text{path}(1, 16) \]

\[ \text{use: path}(X, Y) \]

\[ \text{connected}(X, Z) \land \text{path}(Z, Y) \]

\[ \text{sub: } X/1, Y/16 \]

\[ \text{yes} \]

\[ \text{connected}(1, Z) \land \text{path}(Z, 16) \]

\[ \text{use: connected}(X, Y) \]

\[ \text{connected}(X, Y) \leftarrow \text{connectedSub}(Y, X) \]

\[ \text{connected}(X, Y) \leftarrow \text{connectedSub}(X, Y) \]

\[ \text{proof: } \]

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

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

\[ ?- \text{path}(1, 16). \]
Why is this building the path on way out?

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

\[ \text{path}(X,Y,\text{Path}) \leq \text{connected}(X,Z) \land \text{path}(Z,Y,\text{PathZtoY}) \]

Path = p(X,\text{PathZtoY})

Proof Derivation

\[ \text{path}(1,3,\text{Path}) \] yes(\text{Path}) \leq \text{path}(1,3,\text{Path}) \]

use: \text{path}(X,1,\text{p}(X,\text{PathZY}1)) \leq \text{connected}(X,1) \land \text{path}(1,3,\text{PathZY}1) \]

sub: X1/1 Y1/3 Path/\text{p}(1,\text{PathZY}1) \]

\[ \text{yes}(\text{p}(1,\text{PathZY}1)) \leq \text{connected}(1,Z1) \land \text{path}(Z1,3,\text{PathZY}1) \]

use: \text{connected}(X,2) \leq \text{connectedSub}(X,2) \]

sub: X2/1 Y2/2 Z1/2 \]

\[ \text{yes}(\text{p}(1,\text{PathZY}1)) \leq \text{connectedSub}(1,Z1) \land \text{path}(Z1,3,\text{PathZY}1) \]

use: \text{connectedSub}(1,2) \]

sub: Z1/2 \]

\[ \text{yes}(\text{p}(1,\text{PathZY}1)) \leq \text{path}(2,3,\text{PathZY}1) \]

use: \text{path}(X,3,\text{p}(X,\text{PathZY}3)) \leq \text{connected}(X,3) \land \text{path}(3,Y,\text{PathZY}3) \]

sub: X3/2 Y3/3 PathZY1/\text{p}(2,\text{PathZY}3) \]

\[ \text{yes}(\text{p}(1,p(2,\text{PathZY}3))) \leq \text{connected}(2,Z3) \land \text{path}(Z3,3,\text{PathZY}3) \]

...
Reversing a List

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

# Good place to start is with list represented as head and tail
rev(Head, Tail, R) ←
# now to reverse the tail
rev(Tail, TR) ←
# now to stick head after the reversal of the tail
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(n²)

Can we do better?

Building Path on Way

- For 'way in' version, how is answer returned to user?
- Do both versions build the same path?

Proof Derivation (2nd Version)

?path(1, 3, Path) yes(Path) ←
use: path(X₁, Y₁, P₁) ←
connected(X₁, Z₁) ∧
path(Z₁, Y₁, PathZY₁) ∧
P₁ = path(X₁, PathZY₁)

sub: X₁/1 Y₁/3 P₁/Path

yes(Path) ←
connected(1, Z₁) ∧
path(Z₁, 3, PathZY₁) ∧
Path = path(1, PathZY₁)
use: connected(X₂, Y₂)

sub: X₂/1 Y₂/Z₁

yes(Path) ←
connectedSub(1, Z₁) ∧
path(Z₁, 3, PathZY₁) ∧
Path = path(1, PathZY₁)
use: connectedSub(1, 2)
sub: Z₁/2

yes(Path) ←
path(2, 3, PathZY₁) ∧
Path = path(1, PathZY₁)
use: path(X₃, Y₃, P₃) ←
connected(X₃, Z₃) ∧
path(Z₃, Y₃, PathZY₃) ∧
P₃ = path(X₃, PathZY₃)

sub: X₃/2 Y₃/3 P₃/PathZY₁

yes(Path) ←
connected(2, Z₃) ∧
path(Z₃, 3, PathZY₃) ∧
PathZY₁ = path(X₃, PathZY₃) ∧
Path = path(1, PathZY₁)
Building Lists

- Append:
  - append(p(Head,Tail),L,p(Head,R)) ← append(Tail,L,R)
  - append(nil,L,L).

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

- rev3:
  - rev3(p(Head,Tail),L2,L3) ← rev3(Tail,p(Head,L2),L3)
  - rev3(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

Definition

- Define reverse(J,L) ← rev3(J,nil,L).

- now define rev3 recursively in terms of a smaller version of itself

- rev3(p(Head,Tail),K,L) ← rev3(Tail,p(Head,K),L)

- now the base case

- rev3(nil,L,L).

Efficiently Reversing a List

- Take top off stack, wash it, and put it on the top of the clean dishes
- Take washing a stack of dishes
- and pass it down another list on way into recursion
- and pass elements of list on way into recursion

- To reverse a list, we can
- work in reverse order of the recursion

- Let's take advantage of when we learned from building paths on