Might have a rule that needs numerical calculation

\[ A - X = Z \quad \text{where} \quad Z = \text{some function of } X, A \]

- \text{subtract}(X, Y, Z) \land \text{less}(Z, 5)
- \text{subtract}(X, Y, Z) \land \text{abs}(Z)
- \text{subtract}(X, Y, Z) \land \text{assoc}(Z)
- \text{subtract}(X, Y, Z) \land \text{assoc}(Z)

In theorem proving versus programming:

- Theorem proving is similar to program execution
- Top-down theorem proving is similar to program execution
- Resolution rule is really just procedure call
- Recursion like a while statement
- Unification is like setting variable values
- Resolution rule is really just procedure call
- Recursive rule is really just procedure call
- And done the while block
- Recursion rule is really just procedure call
- Multiple rules with same head similar to if-else
- Resolution rule is really just procedure call
Problems with Friends

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

With top-down resolution, could keep applying rule:
friendSub(tom,sally)
frendSub(tom,george)
frendSub(sally,bill)

Reformulated rules to work with top-down resolution:
friendSub(tom,sally)
frendSub(tom,george)
frendSub(sally,bill)

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 expr command
+ Tcl views everything as a list, even program code
- But will require that Y is a ground expression
- Need to be careful about when atom is evaluated

This is sort of cheating
If we just built it in
Silly to define arithmetic

Numbers

We could define a corresponding definition of subtrahend
Could just use a different notation
+ Defined by the successor function
+ First pick a simple representation for numbers
  + For Z, we use integers with the successor function
+ If X is a number, so is X + Y
+ It's a number if Y is a number

With top-down resolution, could keep applying rule:
friend(tom,sally)
frend(tom,george)
frend(sally,bill)

friendSub(tom,sally)
frendSub(tom,george)
frendSub(sally,bill)
Overview

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
  • Reasoning procedure is separated from knowledge about domain
  • We can use an ordinary theorem prover

Explicit Unification

• Subtract definition:
  \[ \text{subtract}(X, 0, X). \]
  \[ \text{subtract}(X, \text{s}(Y), Z) \leftarrow \text{subtract}(X, Y, Z). \]
  - Z gets in head gets unified after atom in body is resolved

- Can be easier to understand flow if we write it as:
  \[ \text{subtract}(X, 0, X). \]
  \[ \text{subtract}(X, \text{s}(Y), \text{NewZ}) \leftarrow \text{subtract}(X, Y, Z) \]
  \[ Z = \text{s}(\text{NewZ}) \]

• '=' is for explicit unification
  - Really just an infix version of the 2-ary predicate '='
  - Not adding any power to Datalog
  - Not saying that two different constants point to same domain object
  - No adding any power to Datalog
  - Can be easier to understand flow if we write it as: \( \text{subtract}(X, Y, Z) \)
  - Z gets in head gets unified after atom in body is resolved
  - Superficie 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 a list of elements of `L_1` followed by elements of `L_2`
- Useless 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 at each level of 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
- On way into recursion, what is done?

```
length(nil, 0).
length(p(Top, Rest), NewL) ← length(Rest, L) ∧ NewL is L + 1.
```

Recursion and Mathematical Induction

- Idea: define a predicate in terms of simpler instances of itself
- In the predicate—defined by a clause with no body
- Such a definition is derived in terms of elements lower in the data structure
- Such a definition is derived in terms of elements lower in the data structure
- Each such definition requires a definition on each level of the data structure
- Each such definition requires a definition on each level of the data structure
- Recursion works by having

```
length(nil, L) + 1.
length(p(Top, Rest), NewL) ← length(Rest, L) ∧ NewL is L + 1.
```
Example: Path through a Maze

• path of 1 2 6 10 could be represented
  - p(1, p(2, p(6, p(10, nil))))
  - p(10, p(6, p(2, p(1, nil))))
  - p: arbitrary function name to glue path together
  - 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))
  \land \text{member}(S, p(X, R))
  \leftarrow \text{member}(S, R)

Continued

What is done on way into recursion? (before base case)

What is done on way out of recursion? (after base case)

More Content
Does code really work?

... yes ← path(2,16)
use: path(X, Y) ← connected(X, Z) ∧ path(Z, Y)
sub: X/3 Y/16
... yes ← connectedSub(2, Z) ∧ path(Z, 16)
use: connectedSub(X, Y)
sub: X/1 Z/6
... yes ← path(3, 16)
...

• Depth first search will make sure any possible path is explored
• Need to make sure we do not visit a cell already in the path

Towards Building Recursive Structures

Example: Path through a maze
Why is this building the path on way out?

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

Building Recursive Structures

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

Path(X,X,p(X,nil)).

Path(X,Y,p(X,PathZtoY)) ← connected(X,Z) Path(Z,Y,PathZtoY).

...
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
\[
\text{rev}(\text{head}(\text{Tail}), R) \leftarrow
\]
# now to reverse the tail
\[
\text{rev}(\text{Tail}, \text{TR})
\]
# now to stick head after the reversal of the tail
\[
\text{append}(\text{TR}, \text{p}(\text{Head}, \text{nil}), R).
\]
\[
\text{rev}(\text{nil}, \text{nil}).
\]
- Not very efficient though, as we append after each step
- Number of steps in proof \( O(n^2) \)

• Can we do better?

---

Building Path on Way

• Build the current path before the recursive call
  - On way out
  \[
  \text{path}(X,Y, \text{Path}) \leftarrow \text{connected}(X,Z) \wedge \text{path}(Z,Y, \text{PathZtoY}) \wedge \text{Path} = \text{p}(X, \text{PathZtoY})
  \]
  - On way in
  \[
  \text{path}(X,Y, \text{PathToX}) \leftarrow \text{connected}(X,Z) \wedge \text{PathToZ} = \text{p}(Z, \text{PathToX}) \wedge \text{path}(Z,Y, \text{PathToZ})
  \]
  • For 'way in' version, how is answer returned to user?
  • Do both versions build the same path?

---

Proof Derivation (2nd Version)

?\text{path}(1,3, \text{Path})
yes(\text{Path}) \leftarrow \text{path}(1,3, \text{Path})
\]
use: \[
\text{path}(X_1, Y_1, P_1) \leftarrow \text{connected}(X_1, Z_1) \wedge \text{path}(Z_1, Y_1, \text{PathZY}_1) \wedge P_1 = \text{path}(X_1, \text{PathZY}_1)
\]
sub: \[
X_1/1 \ Y_1/3 \ P_1/\text{Path}
\]
\[
\text{yes}(\text{Path}) \leftarrow \text{connected}(1, Z_1) \wedge \text{path}(Z_1, 3, \text{PathZY}_1) \wedge \text{Path} = \text{path}(1, \text{PathZY}_1)
\]
use: \[
\text{connected}(X_2, Y_2) \leftarrow \text{connected}(X_2, Y_2)
\]
sub: \[
X_2/1 \ Y_2/3 \ P_1/\text{Path}
\]
\[
\text{yes}(\text{Path}) \leftarrow \text{connected}(1, Z_1) \wedge \text{path}(Z_1, 3, \text{PathZY}_1) \wedge \text{Path} = \text{path}(1, \text{PathZY}_1)
\]
use: \[
\text{connected}(1, 2)
\]
sub: \[
Z_1/2 \ P_1/\text{PathZY}_1
\]
\[
\text{yes}(\text{Path}) \leftarrow \text{connected}(2, Z_1) \wedge \text{path}(Z_1, 3, \text{PathZY}_1) \wedge \text{PathZY}_1 = \text{path}(1, \text{PathZY}_1) \wedge \text{Path} = \text{path}(1, \text{PathZY}_1)
\]...
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:
  \[
  \text{rev3}(p(\text{Head}, \text{Tail}), L_2, L_3) \leftarrow \text{rev3}(\text{Tail}, p(\text{Head}, L_2), L_3)
  \]
  \[
  \text{rev3}(\text{nil}, L_2, L_2).
  \]
- On way into recursion, tear off top element and put on top of new list
- Pass back completed list on way out

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

- \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}(p(\text{Head}, \text{Tail}), K, L) \leftarrow \text{rev3}(\text{Tail}, p(\text{Head}, K), L)

# now the base case

- \text{rev3}(\text{nil}, L, L).

Efficiently Reversing a List

- Take top off stack list, put on the top of the clean dishes
- The washing is a stack of dishes
- and pull items from the top off the stack into the recursion
- To reverse a list, we can
  - pull off elements of the list on way into recursion
  - work in reverse order of the recursion
  - let's make a list of which we learned from building paths on