Numerical Calculations

A - \( X = Z \) where \( Z \neq X \) if

\[ \begin{align*}
&\text{less}(Diff) \\
&\text{absolute}(Diff) \\
&\text{close}(X,Y) \rightarrow \text{subtract}(X,Y,D) \\
&\text{subtract}(X,Y,Z) \\
&\text{is true for all} \ X, Y, Z \text{ where} \ Z = X - Y.
\end{align*} \]

Theorem Proving versus Programming

- Theorem proving is similar to program execution
- Recursion
- Backtracks are code that is tried but not used
- Resolution rule is like procedure call
- Unification is like setting variable values
- Basis of Prolog

Recursion

Programming
Problems with Friends

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

With top-down resolution, could keep applying rule

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

Reformulated rules to work with top-down resolution

Cheating on Numbers

Silly to define arithmetic

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

Reasoning procedure needs to substitute values in before evaluating it using the expr command, as the variables are not Tcl variables

This is sort of cheating

But very handy!

Numbers

We could define subtraction of numbers

- We could define a corresponding definition of successor

- Could just as easily use some other notation

- Define subtraction by using successor function

- Final code is a simple expression for numbers
Recursion and Mathematical Induction

Idea: define a predicate in terms of simpler instances of itself

\[ \text{live}(X) \leftarrow \text{connected to}(X,Y) \land \text{live}(Y). \]

\[ \text{live}(a). \]

Recursion works by having
- a well-founded ordering of instances of relations
- such that each expression is defined in terms of earlier ones in the ordering
- a well-founded ordering of instances of relations
- a decreasing chain eventually reaches the simplest instance—defined by a clause with no body

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 domain knowledge
- Separation of knowledge from the core procedure
- We can use an ordinary theorem prover
Example: path through a maze

- path(X,Y) :- connected(X,Y).
- path(X,Y) :- connected(X,Z), path(Z,Y).

Finding out if there is a path

Toward Building Recursive Structures

Recursive rule to check if a cell is used in the path

member(S,p(S,X))
member(S,p(X,R)) ← member(S,R)

Recursive Data Structure

- Recursion is only looping construct in Datalog
- Build up data structure on way out of recursion
- Process for recursion starts at the root of the recursion and
- Delivery to base case in the recursion
Building Recursive Structures

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

path(X,Y,p(X,PathZtoY)) ← connected(X,Z)
path(Z,Y,PathZtoY)

?- path(1,16,Path) Path = p(1,p(2,p(6,p(10,p(11,p(7,p(8,p(12,p(16,nil))))))))

Proof Derivation

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

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)

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
+ Need to make sure we do not visit a cell already in the path

Does code really work?
Proof Derivation (2nd Version)

Why is this building the path on way out?

Path = (path, path)
path = (p(X), path(Z to Y))

Path is being accumulated in the answer atom.

Proof Derivation (2nd Version)

why is this building the path on way out?

Path = (path, path)
path = (p(X), path(Z to Y))

Path is being accumulated in the answer atom.
Appending lists

- Might want a predicate `append(L1, L2, L3)`
  - This is similar to Tcl's `concat` command, not its `lappend`
  - True if `L3` is list of elements of `L1` followed by elements of `L2`
  - Useful for verifying whether `L3` is `L1` appended in front of `L2`
  - But also should be useful for appending two lists together:

```lisp
(+ append (p (tim, nil)) (p (john, (p (phil, (p (ted, nil))))) L3)
```

- How do we define `append(L1,L2,L3)`?
  - Only tricky part is which of `L1` and `L2` to attack
  - This is determined by how lists are represented

---

More Recursion

- `length(List,Len)`
  - `true` if `List` has length `Len`
  - `length(null,0)`.
  - `length(p(E,Rest),Len) ← length(Rest,RestLen)` is (`Len,RestLen + 1`).

---

Building Path on Way

- Build the current path before the recursive call
- On way out

```prolog
path(X,Y,Path) ← connected(X,Z) path(Z,Y,PathZtoY)
Path = p(X,PathZtoY)
```

- On way in

```prolog
path(X,Y,PathToX) ← connected(X,Z) path(Z,Y,PathToZ)
PathToZ = p(Z,PathToX)
```

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

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

# now the base case

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

Efficiently Reversing a List

• Let's take advantage of what we learned from building paths on way in versus way 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

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

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

# now to reverse the tail

\[ \text{rev}(\text{Tail},\text{TR}) \leftarrow \]

# now to stick head after the reversal of the tail

\[ \text{append}(\text{TR},\text{p(Head,nil)},R). \]

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

• Not very efficient though, as we append after each step

• Number of steps in proof is \( \Theta(n^2) \)

• Can we do better?
Building Lists

- Append:
  \[ \text{append}(\text{Head}, \text{Tail}, \text{List}) \leftarrow \text{append}(\text{Tail}, \text{List}, \text{Result}) \]
  \[ \text{append}(\text{nil}, \text{List}, \text{List}) \]

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

- Rev3:
  \[ \text{rev3}(\text{Head}, \text{Tail}, \text{List1}, \text{List2}) \leftarrow \text{rev3}(\text{Tail}, \text{List2}, \text{List1}) \]
  \[ \text{rev3}(\text{nil}, \text{List2}, \text{List2}) \]

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