Programming in Logic: Prolog

Accumulator Pairs

& Meta-Programming

Readings: 8.5.4, 23.1 - 23.3
Review

- Ease of understanding and obviousness of correctness are very important.
- `write(Term), nl, tab(N)`: communication with user
- `consult/1 & compile/1`: ways of loading programs into KB
- Term type tests: `var/1, nonvar/1, ...`
Review cont’d

• *name/2*: making and breaking atoms
• *=.*/2, functor/2, arg/3*: making & breaking clauses
• Using constructed clauses: *call/1* & meta-variable facility => data <-> code
• Modifying KB: *assert/retract*
  – Keeping track of values across calls
  – Learning from experience
Difference Lists Revisited

- \( L_1 - L_2, L_2 = [3|L_3] \)
- \( addDL(X, L_1 - L_2, L_1 - L_3) :- L_2 = [X|L_3]. \)
- For \( L_1-L_2 \) to be a difference list, \( L_2 \) must be a tail of \( L_1 \).
Difference Lists Revisited cont’d

For \( \text{addDL}(3, L1-L2, L1-L3) \) to succeed, \( L2 \) must match \([3/L3]\).

Thus, \( \text{addDL}(3, [1,2,4]-[4], [1,2,3]-[]) \), will fail!

While \( \text{addDL}(3, [1,2,3,4]-[3,4], [1,2,3,4]-[4]) \), will succeed!

Even though \([1,2,4]-[4]\) and \([1,2,3,4]-[3,4]\) represent the same list!
Difference Lists Revisited cont’d

• Normally, the 2nd argument of a difference list is an uninstantiated variable, i.e., \([1,2|L] - L\)

• When this is the case, then it is an incomplete data structure that can be further instantiated in an indefinite number of ways.

• It is this combination of difference lists and incomplete data structures that allows us to add to the end of a list in constant time.
Difference Lists Revisited cont’d

• When figuring out what’s happening with difference lists, draw difference diagrams.

• For definition of reverseDL, draw difference diagram for recursive case.
Detecting “Empty” Difference List

• The empty difference list is represented as \( L-L \).
• Let the \( emptyDL/1 \) relation test for empty DLs.
• Consider the definition: \( emptyDL(L - L) \).
• \( emptyDL([1,2] - [2]) \) & \( emptyDL([1,2] - [1,2]) \) both work as expected but what about \( emptyDL([1,2\mid L] - L) \)?
• Depends upon occurs check being implemented.
Recursion on Difference Lists

- Recursive definitions on list usually have the empty list as a base case.
- So DL versions would too!
- But can’t rely on being able to correctly detect empty list if DL’s 2nd argument is unbound.
- Therefore DL arguments which are being recursed on should be fully instantiated.
Pure Prolog Revisited

• Pure Prolog is Prolog code that can be understood as just logic without needing to understand Prolog’s execution of that code.

• Have seen when code no longer Pure Prolog:
  – Uses cut or (a goal using cut or (a goal …))
  – Uses a relation which requires one/more arguments to be instantiated/unbound.

• Now see that code that requires occurs check to work properly is also not pure Prolog.
Recursion vs Iteration

• Recursive definitions are often easier to understand than ones involving iteration because latter requires one to think about control flow and loop invariants.

• Recursive definitions are often more computationally expensive than iterative ones because former usually requires new stack frame per recursive call.
Tail Recursion:
Recursion => Iteration

• A recursive relation is *tail recursive* only if:
  – There is only one recursive call in relation.
  – That call is the last goal of the last clause.
  – All goals preceding last goal in last clause are deterministic.

• Tail recursion => no info to pass up recursive call chain.

• When Prolog finds a tail recursive relation, it transforms the recursion into iteration.
Accumulator Pairs
=> Tail Recursive

• Often possible to transform a non-tail recursive definition into tail recursive one.
• Usual transformation is to build answer on way down rather than on way up recursion.
• Often use auxiliary relation & accumulator pairs to accomplish this.
Accumulator Pair Example

Consider: \( \text{sumList}(\text{+\textit{List}}, \ ?\textit{Sum}) \) where \textit{List} is a list of numbers and 

\textit{Sum} is the sum of those numbers. One definition is:

\[
\text{sumList}([\ ], 0).
\]

\[
\text{sumList}[X|Xs], \textit{Sum} \) :-
\]

\[
    \text{sumList}(Xs, \textit{Sum0}),
    \textit{Sum} \text{ is } X + \textit{Sum0}.
\]

This definition is not tail recursive. It computes the sum on the way up the recursive call chain.
Accumulator Pair Ex. Cont’d

We can transform this into a tail-recursive definition by using accumulator pairs to compute the sum on the way down:

\[
\text{sumList}(\text{List}, \text{Sum}) :- \text{sumList}(\text{List}, 0, \text{Sum}).
\]

\[
\text{sumList}([\ ], \text{Sum}, \text{Sum}).
\]

\[
\text{sumList}([X|\text{Xs}], \text{Accum}, \text{Sum}) :- \text{NewAccum is Accum + X, sumList(}X\text{s, NewAccum, Sum}).
\]

Since \text{is/2} is a deterministic relation, this definition of \text{sumList/3} is tail recursive.
Another Example: \texttt{reverse/3}

Naïve version of \texttt{reverse(?List, ?ReversedList)}:

\begin{verbatim}
reverse([], []).  
reverse([X|Xs], ReversedList) :-
    reverse(Xs, ReversedXs),
    conc(ReversedXs, [X], ReversedList).
\end{verbatim}

Tail recursive version:

\begin{verbatim}
reverse(List, ReversedList) :- reverse(List, [], ReversedList).
reverse([], ReversedList, ReversedList).
reverse([X|Xs], Accum, ReversedList) :-
    reverse(Xs, [X|Accum], ReversedList).
\end{verbatim}
Meta-Programming

• Programming: writing a program to do X.

• Meta-programming: writing a program that writes the program to do X.

• Meta-programs operate on programs, i.e., programs are their data.
Learning as Meta-Programming

• For us, learning = automatic programming.

• Learning modifies the KB to make the “program” more competent.

• A learning program is one that rewrites itself.
clause(+Head, ?Body)

• `clause/2` “retrieves” clauses from KB whose head match `Head`.
• If clause is fact then `Body` is `true`.
• If rule body is only one goal, `Body` is that goal.
• Else `Body` is `(FirstGoal, OtherGoals)`, where “,” is infix operator: ,(FirstGoal, OtherGoals)
• `OtherGoals` may again be pair of goal and other goals.
Prolog Interpreting Prolog

• Basic Prolog meta-interpreter:

prove(true).

prove((Goal1, Goal2)) :- prove(Goal1), prove(Goal2).

prove(Goal) :- clause(Goal, Body), prove(Body).