Tutorium to Introduction to AI, 2nd week - Nicolas Höning

Nicolas Höning

April 27, 2006

induction vs deduction

- deduction is what you did in the logic course

induction vs deduction

- deduction is what you did in the logic course
- deductive reasoning infers no conclusion that is more general than the premises a (famous) example:
  All men are mortal.
  Socrates is a man.
  Therefore Socrates is mortal.

induction vs deduction

- deduction is what you did in the logic course
- deductive reasoning infers no conclusion that is more general than the premises a (famous) example:
  All men are mortal.
  Socrates is a man.
  Therefore Socrates is mortal.
- inductive reasoning infers the universal from the particular An example:
  All observed crows are black.
  therefore
  All crows are black.

Now what does Prolog do?

- I cite the lecture:
  "Recursive (Prolog) programs can be viewed as their own correctness proof!"

Now what does Prolog do?

- I cite the lecture:
  "Recursive (Prolog) programs can be viewed as their own correctness proof!"
- this says the following:
  you formulate your program inductively, and Prolog will try to proof it deductively.
  You make an assumption about the search Space, and Prolog will explore it.

Now what does Prolog do?

- I cite the lecture:
  "Recursive (Prolog) programs can be viewed as their own correctness proof!"
- this says the following:
  you formulate your program inductively, and Prolog will try to proof it deductively.
  You make an assumption about the search Space, and Prolog will explore it.
- so let's think inductively.

gauss(X,Y): An example

- As a little kid, Carl Friedrich Gauss was asked to add up all numbers from 1 to hundred (so the teacher could read the newspaper)

gauss(X,Y): An example

- As a little kid, Carl Friedrich Gauss was asked to add up all numbers from 1 to hundred (so the teacher could read the newspaper)
- After a few minutes he came up with this formula:

$$\sum_{i=0}^{x} i = \frac{x}{2}(x+1)$$

gauss(X,Y): An example

- As a little kid, Carl Friedrich Gauss was asked to add up all numbers from 1 to hundred (so the teacher could read the newspaper)
- After a few minutes he came up with this formula:

$$\sum_{i=0}^{x} i = \frac{x}{2}(x+1)$$

- some of you might know how to proof this formula by induction:
  1. state a base case: for example: it holds for $x = 0$
  2. assume it holds for all $x > 0$
  3. proof that it holds for $x + 1$ IF it holds for x

recursion

- this is what we do (and what induction is all about):
  - think of a base case which is the most simple case imaginable (i.e. $x = 0$)

recursion

- this is what we do (and what induction is all about):
  - think of a base case which is the most simple case imaginable (i.e. $x = 0$)
  - specify the transition from some other case to the next simpler one
- Prolog will then try to *deductively* proof that each case can be reduced to that most simple case then. It does this by applying the transition *recursively* until the simple case is reached.

gauss(X,Y) in Prolog code

- So let's do this for Prolog. We' ll make up a predicate *gauss(+X,+Y)*. What is the most simple case?

gauss(X,Y) in Prolog code

- So let's do this for Prolog. We' ll make up a predicate *gauss(+X,+Y)*. What is the most simple case?
- gauss(0,0).

gauss(X,Y) in Prolog code

- So let's do this for Prolog. We' ll make up a predicate *gauss(+X,+Y)*. What is the most simple case?
- gauss(0,0).
- And now the transition. What do we need to do if we have a case that is just a little more difficult and we want to reduce it to our base case?

gauss(X,Y) in Prolog code

- So let's do this for Prolog. We' ll make up a predicate *gauss(+X,+Y)*. What is the most simple case?
- gauss(0,0).
- And now the transition. What do we need to do if we have a case that is just a little more difficult and we want to reduce it to our base case?
- simply substract X from Y and then decrement X

gauss(X,Y) in Prolog code

- So let's do this for Prolog. We' ll make up a predicate *gauss(+X,+Y)*. What is the most simple case?
- gauss(0,0).
- And now the transition. What do we need to do if we have a case that is just a little more difficult and we want to reduce it to our base case?
- simply substract X from Y and then decrement X
-
```
gauss(X,Y) :-
   X1 is X - 1,
   Y1 is Y - X,
   gauss(X1,Y1).
```
  important: this program works only when both parameters are instantiated (that is said by the "+" in the declaration)!

some more stuff to take care of

- the program needs all parameters due to the calculations: When you do not know X, and you don't know X1, the term $X1 is X - 1$ has infinitely many solutions.
  So all this problem is useful for (besides discussing induction) is checking that some X indeed yields some Y. We'll extend it next week.

some more stuff to take care of

- the program needs all parameters due to the calculations: When you do not know X, and you don't know X1, the term $X1 is X - 1$ has infinitely many solutions.
  So all this problem is useful for (besides discussing induction) is checking that some X indeed yields some Y. We'll extend it next week.
- to stop Prolog from running to negative infinity, we add another line on top of the program (you don't need to do stuff like that for now):

```
gauss(-1,_) :- !, fail.
gauss(0,0).
gauss(X,Y) :-
  X1 is X - 1,
  Y1 is Y - X, gauss(X1,Y1).
```

some more stuff to take care of

- Avoid left recursion. You should use right recursion in almost every case.
  That just means: You call yourself again as the last step.
- Try to insert writeln() - predicates at different points of the recursive gauss - predicate and try to imagine when they are called.

another example: filter()

- let's write a function that filters a specific Item "Out" from a list "A" and returns the result.
  (we do not care if the order is right)

another example: filter()

- let's write a function that filters a specific Item "Out" from a list "A" and returns the result.
  (we do not care if the order is right)
- we could do this in a procedural language:

```
function filter (Item Out, List A)
   List B
   for every Item I in A:
      if I != Out B.push I
   end for
   return B
end function
```

filter() functionally

- and, we could do it more prolog-stylish, that means functional:

filter() functionally

- and, we could do it more prolog-stylish, that means functional:
- function filter (Item Out, List A, List B)

```
function filter (Item Out, List A, List B)
  if A.empty return B
  else
    Item I = A.pop()
    if I != Out B.push I
    return filter (Out, A, B)
  end else
end function
```

filter() functionally

- and, we could do it more prolog-stylish, that means functional:
- function filter (Item Out, List A, List B)
```
function filter (Item Out, List A, List B)
  if A.empty return B
  else
    Item I = A.pop()
    if I != Out B.push I
    return filter (Out, A, B)
  end else
end function
```
- ahh, recursion! and we see:
  - a simple base case (A is empty)
  - a transition step (pop A, push B and try again)

filter in Prolog

- now, how to translate this to Prolog? we need to translate

filter in Prolog

- now, how to translate this to Prolog? we need to translate
- an if/else construct

filter in Prolog

- now, how to translate this to Prolog? we need to translate
- an if/else construct
- and use only Prolog datatypes.
  lists are the only real datatype in Prolog. You can use them for a lot of things.

filter in Prolog

- first: the base case. ideas?

filter in Prolog

- first: the base case. ideas?
- filter(_,[],[]).
  this is the if of the if/else statement. The else will just be another predicate where the parameters are different. That's how it is done in Prolog. Now we want to handle the case where A's first item is not to be filtered.

filter in Prolog

- first: the base case. ideas?
- filter(_,[],[]).
  this is the if of the if/else statement. The else will just be another predicate where the parameters are different. That's how it is done in Prolog. Now we want to handle the case where A's first item is not to be filtered.
- ```
  filter(Out,[Head|RestA],[Head|RestB]) :-
    not(Out=Head),
    filter(Out,RestA,RestB).
  ```
  ```

filter in Prolog

- first: the base case. ideas?
- filter(_,[],[]).
  this is the if of the if/else statement. The else will just be another predicate where the parameters are different. That's how it is done in Prolog. Now we want to handle the case where A's first item is not to be filtered.
- ```
  filter(Out,[Head|RestA],[Head|RestB]) :-
    not(Out=Head),
    filter(Out,RestA,RestB).
  ```
- that leaves us with only one more case: What to do if we indeed want to filter?
  ```

filter in Prolog

- first: the base case. ideas?
- filter(_,[],[]).
  this is the if of the if/else statement. The else will just be another predicate where the parameters are different. That's how it is done in Prolog. Now we want to handle the case where A's first item is not to be filtered.
- ```
  filter(Out,[Head|RestA],[Head|RestB]) :-
     not(Out=Head),
     filter(Out,RestA,RestB).
  ```
- that leaves us with only one more case: What to do if we indeed want to filter?
- ```
  filter(Out,[Out|RestA],ListB) :-
     filter(Out,RestA,ListB).
  ```

it's really important how you specify the parameters because by that you specify your cases!
So make use of that head/tail notation. In this predicate you see that we pass an item from A to B just by giving it the same name:

```
filter(Out,[Head|RestA],[Head|RestB]) :-
   not(Out=Head),
   filter(Out,RestA,RestB).
```

Unification

- Unification is Prolog's way to prove some things are equal. You basically need to know one thing:

Unification

- Unification is Prolog's way to prove some things are equal. You basically need to know one thing:
- Variables are written in capital letters, atoms in small letters. You can assign the latter to the first, but not the other way round (I guess you knew that)

Unification

- Unification is Prolog's way to prove some things are equal. You basically need to know one thing:
- Variables are written in capital letters, atoms in small letters. You can assign the latter to the first, but not the other way round (I guess you knew that)
- The rest is details: Treat predicate names as atoms. Unification is associative [$t \ominus s = t(\ominus\ s) = (t\ominus)s$]. A substitution has a funny symbol (like $\ominus$), but is basically just a possible model for your universe.

Unification - the algorithm

If t1 is a variable then t1$\longrightarrow$t2
If t2 is a variable then t2$\longrightarrow$t1
If t1 and t2 are predicates, decompose them into predicate name and arguments. If predicate names are equal unify the argument lists.
If t1 and t2 are lists, unify element by element.
else: FAIL!

Unification - examples

- p(X,a,Y) =?= p(b,Z,T)

Unification - examples

- p(X,a,Y) =?= p(b,Z,T)
- Z→a, X→b, Y=T

Unification - examples

- p(X,a,Y) =?= p(b,Z,T)
- Z→a, X→b, Y=T
- p(b,a,Y) =?= p(b,Z,T)

Unification - examples

- p(X,a,Y) =?= p(b,Z,T)
- Z→a, X→b, Y=T
- p(b,a,Y) =?= p(b,Z,T)
- Z→a, Y=T

Unification - examples

- p(X,a,Y) =?= p(b,Z,T)
- Z→a, X→b, Y=T
- p(b,a,Y) =?= p(b,Z,T)
- Z→a, Y=T
- p(b,a,Y) =?= p(b,a,T)

Unification - examples

- p(X,a,Y) =?= p(b,Z,T)
- Z→a, X→b, Y=T
- p(b,a,Y) =?= p(b,Z,T)
- Z→a, Y=T
- p(b,a,Y) =?= p(b,a,T)
- Y=T

Unification - examples

- p(X,a,Y) =?= p(b,Z,T)
- Z→a, X→b, Y=T
- p(b,a,Y) =?= p(b,Z,T)
- Z→a, Y=T
- p(b,a,Y) =?= p(b,a,T)
- Y=T
- p(b,a,T) === p(b,a,T)