Countdown Game, Type Inference and Program Synthesis

CS3100 Fall 2019

Review

Previously

- Cuts and Negation

This lecture

- Applications of Prolog
  - Solving the countdown game.
    - Concept of iterative deepening.
  - Type Inference for STLC
  - Program synthesis using iterative deepening.

Countdown game

- We have looked at a few generate and test puzzles before
  - Dutch national flag, N-Queens
  - Time for another one.
- This one doesn't use `perm`.
- Countdown is a TV show that was very popular in the 90s in the UK.

Rules

- Select 6 of 24 number tiles
  - large numbers: 25,50,75,100
  - small numbers: 1,2,3...10 (two of each)
- Contestant chooses how many large and small
- Randomly chosen 3-digit target number
- Get as close as possible using each of the 6 numbers at most once and the operations of addition, subtraction, multiplication and division
- No floats or fractions allowed

If you want to watch how the pros do it, highly recommend watching James Martin 952 (https://www.youtube.com/watch?v=6mCgiaAFCu8) on youtube.

Strategy – generate and test

- maintain a list of symbolic arithmetic terms
- initially this list consists of ground terms e.g.: `[25,50,75,100,6,3]`
- if the head of the list evaluates to the total then succeed
- otherwise pick two of the elements, combine them using one of the available arithmetic operations, put the result on the head of the list, and repeat

Prerequisites

- `eval(A,B)` – true if the symbolic expression A evaluates to B.
- `choose(N,L,R,S)` – true if R is the result of choosing N items from L and S is the remaining items left in L. The order of items in N does not matter.
- `arithop(A,B,C)` – true if C is a valid combination of A and B

Eval

In [1]:

```
eval(plus(A,B),C) :- !, eval(A,VA), eval(B,VB), C is VA + VB.
eval(mult(A,B),C) :- !, eval(A,VA), eval(B,VB), C is VA * VB.
eval(minus(A,B),C) :- !, eval(A,VA), eval(B,VB), C is VA - VB.
eval(div(A,B),C) :- !, eval(A,VA), eval(B,VB), C is VA div VB.
eval(A,A).
```

```
Added 5 clauses(s).
```

Choose

In [2]:

```
choose(0,L,[],L).
choose(N,[H|T],[H|R],S) :- N > 0, M is N-1, choose(M,T,R,S).
choose(N,[H|T],R,[H|S]) :- N > 0, choose(N,T,R,S).
```

```
Added 3 clauses(s).
```

In [3]:

```
?- choose(1,[1,2,3,4,5],X,Y).
```

```
Y = [ 2, 3, 4, 5 ], X = [ 1 ] ;
Y = [ 1, 3, 4, 5 ], X = [ 2 ] ;
Y = [ 1, 2, 4, 5 ], X = [ 3 ] ;
Y = [ 1, 2, 3, 5 ], X = [ 4 ] ;
Y = [ 1, 2, 3, 4 ], X = [ 5 ] .
```

In [4]:

```
?- choose(2,[1,2,3,4,5],X,Y).
```

```
Y = [ 3, 4, 5 ], X = [ 1, 2 ] ;
Y = [ 2, 4, 5 ], X = [ 1, 3 ] ;
Y = [ 2, 3, 5 ], X = [ 1, 4 ] ;
Y = [ 2, 3, 4 ], X = [ 1, 5 ] ;
Y = [ 1, 4, 5 ], X = [ 2, 3 ] ;
Y = [ 1, 3, 5 ], X = [ 2, 4 ] ;
Y = [ 1, 3, 4 ], X = [ 2, 5 ] ;
Y = [ 1, 2, 5 ], X = [ 3, 4 ] ;
Y = [ 1, 2, 4 ], X = [ 3, 5 ] ;
Y = [ 1, 2, 3 ], X = [ 4, 5 ] .
```

Helper predicates for ArithOp

In [5]:

```
isGreater(A,B) :- eval(A,Av), eval(B,Bv), Av > Bv.
notOne(A) :- eval(A,Av), Av =\= 1.
isFactor(A,B) :-  eval(A,Av), eval(B,Bv), 0 is Bv rem Av.
```

```
Added 3 clauses(s).
```

ArithOp

In [6]:

```
/* arithop(+A, +B, -C) */
/* unify C with a valid binary operation of expressions A and B */
arithop(A,B,plus(A,B)).
/* no negative numbers allowed */
arithop(A,B,minus(A,B)) :- isGreater(A,B).
arithop(A,B,minus(B,A)) :- isGreater(B,A).
/* don't allow mult by 1 */
arithop(A,B,mult(A,B)) :- notOne(A), notOne(B).
/* dont allow div by 1 and no fractions allowed */
arithop(A,B,div(A,B)) :- notOne(B), isFactor(B,A).
arithop(A,B,div(B,A)) :- notOne(A), isFactor(A,B).
```

```
Added 6 clauses(s).
```

ArithOp

In [7]:

```
?- arithop(3,6,X).
```

```
X = plus(3, 6) ;
X = minus(6, 3) ;
X = mult(3, 6) ;
X = div(6, 3) .
```

Countdown

In [8]:

```
countdown([Soln|_],Target,Soln) :-
  eval(Soln,Target).
countdown(L,Target,Soln) :-
  choose(2,L,[A,B],R),
  arithop(A,B,C),
  countdown([C|R],Target,Soln).
```

```
Added 2 clauses(s).
```

Here, the first clause is the **test** and the second clause is **generate**.

Countdown

Let's try this out on the same number that James Martin was given in 1997.

In [9]:

```
?- countdown([25,50,75,100,6,3],952,A) {20}.
```

```
A = plus(mult(plus(100, 3), div(mult(75, 6), 50)), 25) ;
A = plus(div(mult(plus(100, 3), mult(75, 6)), 50), 25) ;
A = plus(mult(div(mult(75, 6), 50), plus(100, 3)), 25) ;
A = div(minus(mult(plus(100, 6), mult(75, 3)), 50), 25) ;
A = div(minus(mult(mult(plus(100, 6), 75), 3), 50), 25) ;
A = div(minus(mult(mult(plus(100, 6), 3), 75), 50), 25) ;
A = div(minus(mult(mult(75, 3), plus(100, 6)), 50), 25) ;
A = plus(div(mult(mult(plus(100, 3), 75), 6), 50), 25) ;
A = plus(div(mult(mult(plus(100, 3), 6), 75), 50), 25) ;
A = plus(div(mult(mult(75, 6), plus(100, 3)), 50), 25) ;
A = plus(mult(div(mult(75, 6), 50), plus(100, 3)), 25) .
```

Closest solution

If there are no solutions, we want to find the closest solution.

Define a helper predicate `diff/2` .

In [10]:

```
diff(X,Y,D) :- D is X - Y.
diff(X,Y,D) :- D is Y - X.
```

```
Added 2 clauses(s).
```

In [11]:

```
?- diff(3,5,2).
```

```
true.
```

In [12]:

```
?- diff(5,3,2).
```

true.

Closest Solution

Define the function `gen/3` which generates values within the given bounds.

In [13]:

```
gen(S,E,S).
gen(S,E,P) :- S < E, S2 is S+1, gen(S2,E,P).
```

Added 2 clauses(s).

In [14]:

```
?- gen(0,5,X).
```

```
X = 0 ;
X = 1 ;
X = 2 ;
X = 3 ;
X = 4 ;
X = 5 .
```

Closest Solution

In [15]:

```
solve2([Soln|_],Target,Soln,D) :-
  eval(Soln,R), diff(Target,R,D).
solve2(L,Target,Soln,D) :-
  choose(2,L,[A,B],R),
  arithop(A,B,C),
  solve2([C|R],Target,Soln,D).
closest(L,Target,Soln,D) :-
  gen(0,100,D), solve2(L,Target,Soln,D).
```

Added 3 clauses(s).

This technique of searching for solution with diff 0, then diff 1, and so on is called **Iterative Deepening** in AI.

In [16]:

```
?- closest([25,50,75,100,6,3],959,A,D) {1}.
```

```
A = div(plus(mult(mult(plus(50, 3), 75), 6), 100), 25), D = 1 .
```

Type Inference for STLC

Let us develop a type inference procedure for Simply Typed Lambda Calculus (STLC).

Recall the tems in STLC:

$$\begin{array}{lcll} M, N & \coloneqq & x & \text{(variable)} \\ & \mid & M\ N & \text{(application)} \\ & \mid & \lambda x : A.\, M & \text{(abstraction)} \\ & \mid & \langle M, N \rangle & \text{(pair)} \\ & \mid & \mathsf{fst}\ M & \text{(project-1)} \\ & \mid & \mathsf{snd}\ M & \text{(project-2)} \\ & \mid & () & \text{(unit)} \end{array}$$

STLC Typing Rules

$$\frac{}{\Gamma, x : A \vdash x : A}\ (var) \qquad \frac{}{\Gamma \vdash () : 1}\ (unit)$$

$$\frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}\ (\rightarrow elim) \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.\, M : A \rightarrow B}\ (\rightarrow intro)$$

$$\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathsf{fst}\ M : A}\ (\times\, elim1) \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathsf{snd}\ M : B}\ (\times\, elim2)$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N \rangle : A \times B}\ (\times\, intro)$$

Type Checking to Type Inference

- STLC rules are presented in a way that you can easily do type checking.
- In the standard presentation of type inference algorithm for STLC, you will need
  - Type schemes (types with variables in them)
  - Unification of type schemes
  - Substituion for variables in type schemes.
- Luckily Prolog provides all of these
  - Type schemes -> Prolog terms with variables,
  - Unification -> Prolog unification
  - Substitution -> Prolog substitution.

My Secret Plan

was to teach Prolog was a way to teach you type inference.

Well, not really :-). But it works out well.

Remove simple types & add polymorphism

- Since we have type schemes (variables in terms), we can infer **polymorphic types**!
- Rather than writing $\lambda x : A.\, M$, we just write $\lambda x.\, M$.
  - We will infer the most general type for x.
- For fun, we will also integers, booleans, + and < on integers, if-then-else.

Occurs check

We will enable occurs check so that the term $\lambda x.\, x\ x$ will be ill-typed.

In [17]:

```
?- set_prolog_flag(occurs_check,true).
```

```
true.
```

Typing Judgement

We model the typing environment Γ as a list of variable and type pairs.

We implement the predicate `lookup/2` to lookup the type of a variable in the environment.

In [18]:

```
lookup([(X,A)|T],X,A).
lookup([(Y,_)|T],X,A) :- \+ X = Y, lookup(T,X,A).
```

```
Added 2 clauses(s).
```

Observe that we are using `\+ X = Y` which holds when `X` does not unify with `Y`.

Typing rules

Next we encode the typing rules as they are specified in the STLC typing rules.

In [19]:

```
/* unit */     type(G,u,unit).
/* -> elim */  type(G,app(M,N),B)           :- type(G,M,A -> B),type(G,N,A).
/* -> intro */ type(G,lam(var(X),M),A -> B) :- type([(X,A)|G],M,B).
/* X elim1 */  type(G,fst(M),A)             :- type(G,M,A * B).
/* X elim2 */  type(G,snd(M),B)             :- type(G,M,A * B).
/* X intro */  type(G,pair(M,N),A * B)      :- type(G,M,A), type(G,N,B).
/* var */      type(G,var(X),A)             :- lookup(G,X,A).
```

```
Added 7 clauses(s).
```

Typing rules

Add the new typing rules the additional terms and operators.

In [20]:

```
type(G, X, int)        :- integer(X).
type(G, true, bool).
type(G, false, bool).
type(G, A + B, int)    :- type(G,A,int), type(G,B,int).
type(G, A - B, int)    :- type(G,A,int), type(G,B,int).
type(G, A < B, bool)   :- type(G,A,int), type(G,B,int).
type(G, ite(A,B,C), T) :- type(G,A,bool), type(G,B,T), type(G,C,T).
```

```
Added 7 clauses(s).
```

In [21]:

```
type(Term,Type) :- type([],Term,Type).
```

```
Added 1 clauses(s).
```

Type inference

Now we can infer the type of programs written in STLC.

What is the type of $1 + 2$?

In [22]:

```
?- type(1+2,X).
```

```
X = int .
```

what is the type of $\lambda x. \lambda y. \text{if } x < y \text{ then } x + y \text{ else } x - y$?

In [23]:

```
?- X = var(x), Y = var(y), type(lam(X,lam(Y,ite(X < Y,X+Y,X-Y))),T).
```

```
Y = var(y), X = var(x), T = ->(int, ->(int, int)) .
```

It is `int -> int -> int` .

Type inference

We can also infer polymorphic types.

What is the type of $\lambda x. fst(x) + 1$?

In [24]:

```
?- X = var(x), type(lam(X,fst(X)+1),T).
```

```
X = var(x), T = ->(*(int, _1882), int) .
```

It is `int * 'a -> int` .

Type inference

What is the type of $\lambda f. \lambda x. f\ x$?

In [25]:

```
?- F = var(f), X = var(x), type(lam(F,lam(X,app(F,X))),T).
```

```
X = var(x), T = ->(->(_2024, _2026), ->(_2024, _2026)), F = var(f) .
```

It is `('a -> 'b) -> ('a -> 'b)` or equivalently `('a -> 'b) -> 'a -> 'b`

Type Inference

- We cannot infer types for every program.
  - such programs do not have a valid STLC type.

What is the type of $\lambda x. x\ x$?

In [26]:

```
?- X = var(x), type(lam(X,app(X,X)),T).
```

```
false.
```

What is the type of `if true then 0 else false` ?

In [27]:

```
?- type(ite(true,0,false),T).
```

```
false.
```

Program synthesis

- Program synthesis is generating programs according to a given specification.
- Our specifications are types!
- Let's generate lambda calculus programs that correspond to a particular type.
  - We will use iterative deepening to guide our search.
  - Otherwise, Prolog starts to explore down infinite paths
    - programs have no bounded length and Prolog uses DFS.
- Let's use the depth of the AST in order to iteratively search starting from depth of 0.

Bounded predecessor

Defines the predecessor for numbers >= 0.

In [28]:

```
pred(D,DD) :- D >= 0, DD is D - 1.
```

Added 1 clauses(s).

Add depth to the type checking rules

In [29]:

```
type(_,u,unit,D) :-
  pred(D,_).
type(G,app(M,N),B,D) :-
  pred(D,DD), type(G,M,A -> B,DD),type(G,N,A,DD).
type(G,lam(var(X),M),A -> B, D) :-
  pred(D,DD), type([(X,A)|G],M,B, DD).
type(G,fst(M),A,D) :-
  pred(D,DD), type(G,M,A * _,DD).
type(G,snd(M),B,D) :-
  pred(D,DD), type(G,M,_ * B,DD).
type(G,pair(M,N),A * B,D) :-
  pred(D,DD), type(G,M,A,DD), type(G,N,B,DD).
type(G,var(X),A,D) :-
  pred(D,_), lookup(G,X,A).
```

Added 7 clauses(s).

Add depth to the type checking rules.

In [30]:

```
type(_,X,int,D) :-
  pred(D,_), integer(X).
type(_,D,int,D) :-
  pred(D,_).
type(_,true,bool,D) :-
  pred(D,_).
type(_,false,bool,D) :-
  pred(D,_).
type(G,A + B,int,D) :-
  pred(D,DD), type(G,A,int,DD), type(G,B,int,DD).
type(G,A < B,bool,D) :-
  pred(D,DD), type(G,A,int,DD), type(G,B,int,DD).
type(G,ite(A,B,C),T,D) :-
  pred(D,DD), type(G,A,bool,DD), type(G,B,T,DD), type(G,C,T,DD).
```

Added 7 clauses(s).

Iteratively search for candidate programs

In [31]:

```
synthesize(P,T) :-
  gen(0,10,D), type([],P,T,D).
```

```
Added 1 clauses(s).
```

Synthesis

Get me those programs whose type is `int` .

In [32]:

```
?- synthesize(P,int).
```

```
P = 0 ;
P = 1 ;
P = +(0, 0) ;
P = ite(true, 0, 0) ;
P = ite(false, 0, 0) ;
P = app(lam(var(_1734), var(_1734)), 1) ;
P = app(lam(var(_1734), var(_1734)), +(0, 0)) ;
P = app(lam(var(_1734), var(_1734)), ite(true, 0, 0)) ;
P = app(lam(var(_1734), var(_1734)), ite(false, 0, 0)) ;
P = app(lam(var(_1734), 0), u) .
```

Synthesis

Let's ask for something more interesting.

Get me the program whose type is `A * B -> A` .

In [33]:

```
?- synthesize(P,(A*B)->A).
```

```
A = unit, P = lam(var(_1782), u), B = _1724 ;
A = int, P = lam(var(_1782), 0), B = _1724 ;
A = bool, P = lam(var(_1782), true), B = _1724 ;
A = bool, P = lam(var(_1782), false), B = _1724 ;
A = unit, P = app(lam(var(_1800), var(_1800)), lam(var(_1838), u)), B
= _1724 ;
A = int, P = app(lam(var(_1800), var(_1800)), lam(var(_1838), 0)), B =
_1724 ;
A = bool, P = app(lam(var(_1800), var(_1800)), lam(var(_1838), true)),
B = _1724 ;
A = bool, P = app(lam(var(_1800), var(_1800)), lam(var(_1838), fals
e)), B = _1724 ;
A = unit, P = lam(var(_1782), u), B = _1724 ;
A = ->(_1814, unit), P = lam(var(_1782), lam(var(_1810), u)), B = _172
4 .
```

Lots of valid programs, but not the program that we are looking for. i.e) the program that doesn't specialise `A` and `B` .

Synthesize

Get me the program whose type is `A * B -> A`, where `A` and `B` remain polymorphic, and `A` and `B` do not unify.

In [34]:

```
?- synthesize(P,(A*B)->A), var(A), var(B), dif(A,B) {1}.
```

```
A = Variable(255), P = lam(var(_1976), fst(var(_1976))), B = Variable
(258) .
```

That's the program we are looking for: $\lambda p.\, \mathsf{fst}\ p$.

Fin.