L19: Unification type systems (Preliminary Slides)

Sunday, November 07, 2010
6:30 PM

Refresh previous lesson

- Static vs dynamic types: are expressions or values typed?
- Strong vs weak typing?
- The matrix of the two axes. C looks good because it is static but weak

Runtime checks needed to reject reasonable programs:

```
var     s : array [1..10] of character;
    s := 'hello';                      { You wish }

    {----Thank you sir and may I have another! ----------}

    type string = array [1..40] of character;
    procedure error (c: string)
    begin
       write('ERROR: ');
       write(c);
       writeln('');
    end;

    error('File not found');  { In your dreams }
    error('File not found                      ');  { You have to do this
    error('Please just kill me Mr. Wirth       ')
```

Example of why C type checker misses a problem.

Previous type system were **nominal**. We gave types to values (when we created them, in dynamicall typed langauegs) or to expressions (eg when variable was declared).

We also annotated types of method arguments and return types.

Now for something completely different. Why give names to classes? Why give types of variables when the value can be inferred from leaves of the computations which must be literals.

Outline

Why annotate types when we can infer them and check them?

Simple example:

Example 1: Consider this factorial program.

```
def fact(n):
    if (n==0) { 1 } else { n * fact(n-1) }
```

Let's *type* this function. *Typing a function* includes type inference and type checking. We will want to ask three questions:

1) what is the type of the parameter n?
2) what is the return type of fact?

3) is the function type safe, ie will we perform only operations sanctioned by their type?

We will first write the type rules of the arithmetic:

```
type(0,int).                % 0 is an int value
type(1,int).
mult(int,int,int).          % E * E
mult(float,float,float).
sub(int,int,int).           % E - E
sub(float,float,float).
comparable(int,int).        % E == E
comparable(float,float).
```

Now we collect the type constraints present in fact. If all these constraints hold, fact is type safe.

```
fact(fun(I,O)) :-  % let's call I type of param n
    type(0,T0),
    type(1,T1),
    comparable(I,T0),
    T1=O,          % is n==0 legal?
    mult(I,O,O),   % type of 1 is the return type
    sub(I,T1,I).   % (2)
```

Now we can solve these constraints. If there is a solution, then fact is type safe if called with the parameter type I. We'll also learn that it will produce the result of type O.

```
?- fact(fun(I,O)).
I = int
O = int
```

If we relax the constraints and allow 0 and 1 to be floats, essentially saying that we can convert them to float if needed,

```
type(0,float).
type(1,float).
```

then there will be another solution.

```
I = float
O = float
```

Notes:

(1) fun(I,O) our way of denoting the function type I -> O

(2) How do we know that the return type of fact(n-1) is O? We have decided that fact has the same type in each invocation, hence the type fact(n-1) must be the same as that of fact(n), which we denote O.

We can often infer types even for open programs (functions, modules without types of inputs known). This is where unification especially helps us.

Example 2: Let's assume that we don't know what numeric types the programmer may define, yet we

want to type the function fact. ("type" means type check and produce its type, which is type of parameters and the return type)

Let's leave the program constraints the same (there is no need to change them) but let's make types of arithmetic more general.

```
type(0,A) :- num(A).
type(1,A) :- num(A).

mult(A,A,A) :- num(A).
sub(A,A,A) :- num(A).
comparable(A,A).

num(X).

% num(int).
% num(float).
```

Now the answer to fact(T) is

T = fun(X,X)

Well, instead of X, the system uses an "anonymous" variable _234 or something similar.