Programming Languages

Fall 2020 - COM SCI131-1 - EGGERT

UCLA CS 131 lecture 2020-12-09

admin stuff
- Friday is last day for assignments.
- Final exam like the midterm
  - Zoom session at the start (Registrar specifies time of start)
    to cover any last-minute details
  - You pick three hours within a 24-hour window.
  - Covers the whole class, emphasizing 2nd half
- review during discussion
  - I'll post past finals on CCLE

Parameter passing
- call by value
- call reference (motivated by efficiency)
  (has a problem: aliasing)

Here's what you see:	Here's what happens
int i = 10;	int i = 10;
int f (int &a)	int f (int *a)
{	{
i = a + 1;	i = *a + 1;
a = 3;	*a = 3;
return i;	return i;
}	}
caller:	
return f(i);	return f(&i);

Q. I am still a bit confused by the fist example, was the problem that the caller of the function was expecting the result with the latest i value? but instead it was getting the old value added to the return value?
A. A naive reading of left hand side would say "return arg plus one, and set i to that value, then set the argument to 3." But this particular call always returns 3.

Aliasing:
- * two names for the same variable
- * makes code more confusing to programmers
- * means compiler can't generate as good a code
  can't cache values into registers

One approach: give up
- Fortran: programs should not do aliasing,
  so compilers can generate fast code
- either implementation is allowed

Another approach: do what C does
- call by value only

Q. What is the example of using call by reference without aliasing?
A. Write code in which no parameters overlap with each other
or with global variables.

```
int i;
  int f (int &a, int &b)
{
  a = b + 1;
  b = i + 2;
}

int x, y;
  f(x, y);
```

Another approach (Ada)
call by result
caller doesn't evaluate the argument
callee determines the argument by assigning to it
when the callee returns that value is copied back to the caller

```
  bool read (FILE *input, char result input_char)
{
  ...
  input_char = input->buf[i++];
  return true;
}

  char v;
if (read (stdin, v))
  ...
```

Safer because underlying pointers are not accessible to
the programmer.

Q. For call by result, does that means we have to pass a
pointer that is not accessible, or pass by result is not
really helpful?
A. It can be done via pointer, but not necessarily.
"Results are in %rax, %rbx, %rcx, respectively."

call by value-result = call by value + call by result
values copied from caller to callee
callee computes, returns
values copied back from callee to the caller
less efficient
no aliasing problems

call by unification (Prolog)
pattern match between caller and callee
which makes the two arguments the "same" -- at least until failure

macro calls (C, C++, Scheme, ...)
arguments are pieces of your program
C, C++ - sequences of tokens
Scheme - data that represent program in the usual Lisp way
occur at compile time

do no 'evaluation' of the parameters, as opposed to any of the other call-by approaches?
A. Yes, but the parameters are macro-expanded.

m(1, n(2))

Expand n(2) to get a piece of a program x+2
Expand m(1, x+2) to get another piece of a program x+2+1

Q. Where does callback function fall into?
A. That's my next topic!

call by name.
call by name : functions :: call by reference : pointers

```
                                    thunk
                                    v
  int i;            int i;
  int sumup(int name e)       int sumup(int (*e) (void))
{                     {
    int a;                  int a;
 for (i=0; i<1000; i++)       for (i=0; i<1000; i++)
  a += e;                     a += e();
 return a;                   return a;
  }                     }


  int ar[1000];             int ar[1000];
  int myfun(void) {         int myfun(void) {
 return sumup(ar[i]*ar[i]);   return sumup(p);
  }                     }
                      int p(void)
          {
            return ar[i]*ar[i];
          }
```

Call by name can be more reliable

```
   int print_average (int n, int avg)
 {
   if (n == 0)
    printf("*\n");
   else
    printf("Average is %d\n", avg);
   }


   print_average (size, sum / size);
```

If size == 0, call by value will crash, but
call by name will work.

Lazy evaluation can be a win, if you put things off that don't actually need to be done.

There's a problem with all this - efficiency issue.
It could be that the actual parameter is an expensive call,

call by need = call by name + cache the result
callee invokes the thunk at most once
Functional programming languages like Haskell
use this to great effect - they push lazy evaluation.

let s = (... set of all prime numbers ...)
print (s[3]) // compute s[0], s[1], s[2], s[3] but then stop

Your program computes an enormous to-do list,
and it doesn't do any of it
until it absolutely must (e.g., print)

Q. But with the first example (sumup) the result returned by the thunk changes every time it is called, so caching could not work in that case, right?
A. Yes, the sumup example would not work with call by need.
If you lack side effects, call by need is pretty good.

Cost models
It's important to have a mental model of how your program works.
For efficiency's sake.
E.g., heap management

Downside of cost models:
They become obsolete.
They might not be portable.

E.g., cost model for Python dictionaries.
d['abc'] = 27
d['defgh'] = 97
underlying cost model is that of a hash table.
return d['abc']
O(1)? O(len(d))?
~ O(1) amortized is common for hashing
Main Google search web server in Python
Many common phrases occur in web searches
"Gene Block", say
This dictionary d caches results of common searches
d["Gene Block"] = list of answers
d["Len Kleinrock"] = another list
...
Suppose the hashing function is a bad one.
Suppose a Google competitor is trying to bring Google down.
look at the hash function
issue query "14sd(&*&*", "c497qwer.xchbvpasdgf7"
h("14sd(&*&*") = 3343224
h("c497qwer.xchbvpasdgf7") = 3343224
h("Gene Block") = 3343224
(Aside: how to solve this?
trie is better for this, but slower for normal case
so: use a "secret" hashing function
or use "salt")

(Aside: C++, C have a "register" keyword
for performance - it was a mistake)

Semantics
What does a program *mean*?
syntax (first 2 weeks) - form of a program - "solved" problem
semantics - meaning of a program, irrespective of form
static semantics - what you can easily deduce before the program runs
what a compiler knows
easier
dynamic semantics - you need to run the program to know what it *means*
by observing what it *does*
harder - the Halting Problem provides a limit here

static semantics example:
attribute grammars - Knuth
basic idea:
associate with each grammar rule (about the syntax)
semantic rules that give us info about the meaning

Expr1 -> Expr2 + Expr3
type(Expr1) <- if type(Expr2) = 'int' & type(Expr3) = 'int'
then 'int'
else 'float'
symtab(Expr2) <- symtab(Expr1)
symtab(Expr3) <- symtab(Expr1)

dynamic semantics
operational
give an interpreter for the language
"Run the the program on the interpreter."
correspond to imperative languages

axiomatic
give axioms and rules of inference for the language
"Prove whatever properties you like."
correspond to logic languages

denotational
provide a function from programs to meanings
"Apply the meaning function."
correspond to functional languages

Webber provides an operational semantics for a subset of ML.
written in Prolog.
Idea is: assumption is you know Prolog,
so you can understand what an ML program means
when do we stop? philosophically this is unsatisfactory
English Dictionary:
the: the definite article
The classic operational semantics for Lisp
is an interpreter written in Lisp!

```
% Meaning of E, in a context C, is the value R.
% A context C is a list of name-value pairs, e.g., [v=5, w=10, rr=11]
%    m(E, C, R).

   m(E, _, E) :- integer(E).
```

```
m(E1+E2, C, R) :-
  m(E1, C, V1),
  m(E2, C, V2),
  R is V1+V2.

% let Var=Val in E
m(let(Var,Val,E), C, R) :-
  m(Val, C, V1),
  m(E, [Var=V1 | C], R).

--------------
% fun X->E
m(fun(X,E), C, fun(X,E,C)).

% F A
m(call(F,A), C, R) :-
  m(F, C, fun(X,E,Cf)),
  m(A, C, Val),
  m(E, [X=Val|Cf], R).  % static scoping, not dynamic scoping.
--------------
% fun X->E
m(fun(X,E), _, fun(X,E)).

% F A
m(call(F,A), C, R) :-
  m(F, C, fun(X,E)),
  m(A, C, Val),
  m(E, [X=Val|C], R).  % dynamic scoping, not static scoping.
```

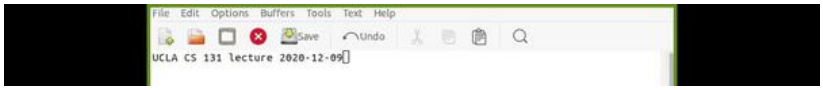

More about this in CS 130
  axiomatic semantics are a big deal there.
  It lets you prove properties of programs,
    you prove that gcd(a,b) actually computes G.C.D.
  Q. Prove invariants?
  A. Yes! Loop invariants - true each time through the loop
       "For 0<=i<n, a[i] is positive".
     for (n=0; n < 1000; n++)
       ...
       Trivial at the start of the loop
     If true at the end of the loop, you have a constraint
     on the state of the program.
    You can also do this with recursion.

    Doing these proofs can be expensive.

Last chapter!

A history of programming languages.

  1960s paper "The Next 700 Programming Languages"



◄ UCLA CS 131 lecture 2020-12-07

Jump to...

Discussion 1A Zoom Link ►