CMSC 330: Organization of Programming Languages

Type Systems, More on Scoping, and Parameter Passing, con't.

Parameter Passing in OCaml

- Quiz: What value is bound to z?

```
let add x y = x + y

let z = add 3 4
```

7

```
let add x y = x + y

let z = add (add 3 1) (add 4 1)
```

9

```
let r = ref 0
let add x y = (!r) + x + y
let set_r () = r := 3; 1

let z = add (set_r ()) 2
```

Actuals evaluated before call

6

Call-by-Value

- In *call-by-value*, actual parameters to functions are fully evaluated before the function is invoked
  - Also in OCaml, in let x = e1 in e2, the expression e1 is fully evaluated before e2 is evaluated
- Java and C also use call-by-value

```
int r = 0;
int add(int x, int y) { return r + x + y; }
int set_r(void) {
  r = 3;
  return 1;
}
add(set_r(), 2);
```

Another Puzzle

- Quiz: What value is bound to z?

```
let r = ref 0
let add x y = x + y
let set_r () = r := 3; 1

let z = add (!r) (set_r ())
```

- It depends on the *order of evaluation*
  - Usually this is very explicit
    - e1; e2 (* evaluate e1 before e2 *)
  - Function arguments is one place it's confusing
    - May be specified in a language, or it may not be
    - May depend on optimization level
    - It's a bad habit to depend on it if you're not sure

Order of Evaluation

- Will OCaml raise a Division_by_zero exception?

```
let x = 0

if x != 0 && (y / x) > 100 then
  print_string "OCaml sure is fun!"

if x == 0 || (y / x) > 100 then
  print_string "Sure, OCaml is fun!"
```

  – No: && and || are *short-circuiting* in OCaml
    - e1 && e2 evaluates e1. If false, it returns false. Otherwise, it returns the result of evaluating e2
    - e1 || e2 evaluates e1. If true, it returns true. Otherwise, it returns the result of evaluating e2

Order of Evaluation, con't.

- Java, C, and Ruby all short-circuit &&, ||
- But some languages don’t, like Pascal:

```
x := 0;
...
if (x <> 0) and (y / x > 100) then
  writeln('Sure OCaml is fun');
```

  – So this would need to be written as

```
x := 0;
...
if x <> 0 then
  if y / x > 100 then
    writeln('Sure OCaml is fun');
```

Call-by-Value in Imperative Languages

- In Java and C, call-by-value has another feature
  – What does this program print?

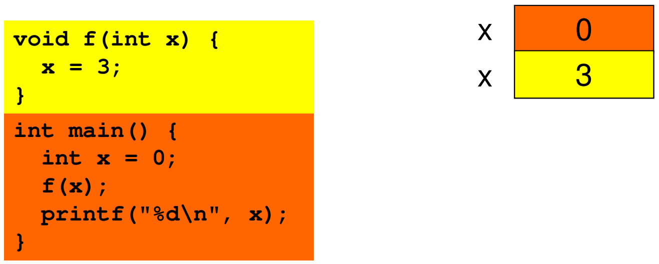
```
void f(int x) {
  x = 3;
}

int main() {
  int x = 0;
  f(x);
  printf("%d\n", x);
}
```

  – Prints 0

Call-by-Value in Imperative Languages, con't.

- The value of the actual parameter is copied to the stack location of the formal parameter



```
void f(int x) {
  x = 3;
}
int main() {
  int x = 0;
  f(x);
  printf("%d\n", x);
}
```

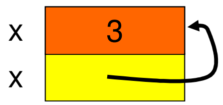
x 0

x 3

Call-by-Reference

- Alternative idea: Implicitly pass a *pointer* or *reference* to the actual parameter
  - If the function writes to it the actual parameter is modified

```
void f(int x) {
  x = 3;
}
int main() {
  int x = 0;
  f(x);
  printf("%d\n", x);
}
```



Call-by-Reference, con't.

- Advantages
  - The entire argument doesn't have to be copied to the called function
    - It's more efficient if you're passing a large (multi-word) argument
    - Can do this without explicit pointer manipulation
  - Allows a function to easily change several parameters ("return" more than one value)
- Disadvantages
  - Can you pass a non-variable (e.g., constant, function result) by reference?
  - It may be hard to tell if a function modifies an argument
  - What if you have *aliasing*?

Aliasing

- We say that two names are *aliased* if they refer to the same location in memory
  - C examples (this is what makes optimizing C hard)

```
int x;
int *p, *q;

p = &x;   /* *p and x are aliased */
q = p;    /* *q, *p, and x are aliased */
```

```
struct list { int x; struct list *next; }
struct list *p, *q;
...
q = p;    /* *q and *p are aliased */
          /* so are p->x and q->x */
          /* and p->next->x and q->next->x... */
```

Aliasing Example

- What happens in the following function?

```
void f(int *x, int *y, int n) {
  /* f is supposed to add 2 * y[i] to x[i] */
  int i;
  for (i = 0; i < n; i++) {
    x[i] += y[i];
    x[i] += y[i];
  }
}

int a[] = {1, 2, 3, 4, 5};
f(a, a, 5);
```

Call-by-Reference, con't.

- Call-by-reference is still around (one popular language that has call-by-reference is C++), but seems to be less popular
  - Possible efficiency gains not worth the confusion
  - There are other ways to achieve the same results
    - If you've got pointers, use those instead
    - Or, in Java, pass in an object whose field gets set
  - "The hardware" is basically call-by-value
    - Although call by reference is not hard to implement and there may be some support for it

Call-by-Value Discussion

- Call-by-value is the standard for languages with side effects
  - When we have side effects, we need to know the order in which things are evaluated, otherwise programs have unpredictable behavior
  - Call-by-value specifies the order at function calls
- Most languages you'll see use call-by-value
- But there are alternatives...

Call-by-Name

- In *call-by-name*, arguments to functions are evaluated at the last possible moment, just before they're needed

```
let add x y = x + y
let mult x y = x * y

let z = mult (add 3 1) (add 4 1)
```

OCaml; call-by-value; arguments evaluated here

Haskell; call-by-name; arguments evaluated here

```
add x y = x + y
mult x y = x * y

z = mult (add 3 1) (add 4 1)
```

Call-by-Name, con't.

- What would be an example where this difference matters?

```
let cond p x y = if p then x else y
let rec loop n = loop n
let z = cond true 42 (loop 0)
```

OCaml; call-by-value; infinite recursion at call

```
cond p x y = if p then x else y
loop n = loop n
z = cond True 42 (loop 0)
```

Haskell; call-by-name; never evaluated because parameter is never used

Two Cool Things to Do with Call-by-Name

- Build control structures with functions

```
let cond p x y = if p then x else y
```

- Build “infinite” data structures

```
integers n = n:(integers (n + 1))
take 10 (integers 0)  (* infinite recursion in call-by-
                         value *)
```

- Call-by-name is also called *lazy evaluation*
  - (Call-by-value is also known as *eager evaluation*)

Three-Way Comparison

- Consider the following program under the three calling conventions
  - For each, determine i's value and which a[i] (if any) is modified

```
int i = 1;

void p(int f, int g) {
  g++;
  f = 3 * i;
}

int main() {
  int a[] = {7, 5, 3};
  p(a[i], i);
  printf("%d %d %d %d\n", i, a[0], a[1], a[2]);
}
```

Other Calling Mechanisms

- *Call-by-result*
  - Actual argument passed by reference, but not initialized
  - Written to in function body (and since passed by reference, affects actual argument)
- *Call-by-value-result*
  - Actual argument copied in on call (like call-by-value)
  - Mutated within function, but does not affect actual yet
  - At end of function body, copied back out to actual
- These calling mechanisms didn't really catch on
  - They can be confusing in cases
  - Recent languages don’t use them

Simulating Call-by-Name with Call-by-Value

- Call-by-name is implemented by passing in a *thunk* that, when called, evaluates to the actual parameter
  - Within the body, formal argument thunks are invoked to get actuals
  - A thunk is a compiler-generated function with no arguments, which returns the actual parameter

Simulating Call-by-Name with Call-by-Value, con't.

```
let cond p x y = if p then x else y
let rec loop n = loop n
let z = cond true 42 (loop 0)
```

– becomes...

Get 1st argument

Return 2nd argument

```
let cond p x y = if (p ()) then (x ()) else (y ())
let rec loop n = loop n  (* didn’t transform... *)
let z = cond (fun () -> true)
             (fun () -> 42)
             (fun () -> loop 0)
```

Never invoked

Call-by-Value versus Call-by-Name

- Call-by-name is flexible- strictly more programs terminate
  - E.g., where we might have an infinite loop with call-by-value, we might avoid it with call-by-name by waiting to evaluate
- Order of evaluation is really hard to see in call-by-name
  - Call-by-name doesn't mix well with side effects (assignments, print statements, etc.)
- Call-by-name is more expensive since:
  - Functions have to be passed around
  - If you use a parameter twice in a function body, its thunk will be called twice
    - Haskell actually uses *call-by-need* (each formal parameter is evaluated only once, where it's first used in a function)

Call-by-Value versus Call-by-Name, con't.

- Call-by-name isn't very “mainstream”
  - Haskell solves these issues by not having side effects
  - But then someone invented “monads” so you can have side effects in a lazy language
- Call-by-name's benefits may not be worth its cost

Tail Calls

- A *tail call* is a function call that is the last thing a function does before it returns

```
let add x y = x + y
let f z = add z z (* tail call *)
```

```
let rec length = function
  [] -> 0
| (_::t) -> 1 + (length t) (* not a tail call *)
```

```
let rec length a = function
  [] -> a
| (_::t) -> length (a + 1) t (* tail call *)
```

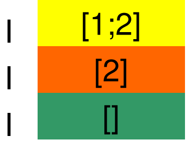
Tail Recursion

- Recall that in OCaml, repetition is typically done via recursion
  - Seems very inefficient
  - Needs one stack frame for recursive call
- A function is *tail recursive* if it is recursive and the recursive call is a tail call
- Suppose a program is running on the x86 architecture, which uses the eax register to store a function's return value when the function exits

Tail Recursion, con't.

```
let rec length l = match l with
    [] -> 0
  | (_::t) -> 1 + (length t)

length [1; 2]
```



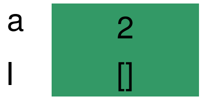
eax: 2

- Tail recursion can be implemented efficiently because we can reuse the stack frame for each recursive call

Tail Recursion, con't.

```
let rec length a l = match l with
    [] -> a
  | (_::t) -> (length (a + 1) t)

length 0 [1; 2]
```



eax: 2

- The same stack frame is reused for the next call, since we'd just pop it off and return anyway