CMSC 330: Organization of Programming Languages

Parameter Passing

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
```

6

Actuals evaluated before call

Call-by-Value

- In call-by-value (*cbv*), arguments 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
- C, C++, and Java 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);
```


Call-by-Value in Imperative Languages

- In C, C++, and Java, call-by-value has another feature
  - What does this program print? 0

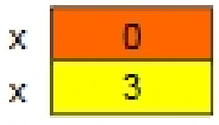
```
void f(int x) {
  x = 3;
}

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

  - Cbv protects function arguments against modifications

Call-by-Value (cont.)

- Actual parameter is copied to stack location of formal parameter

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



Call-by-Reference

- Alternative idea
  - Implicitly pass a pointer or reference to 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 (cont.)

- Advantages
  - Allows multiple return values
  - Avoid copying entire argument to called function
    - More efficient when passing large (multi-word) arguments
    - Can do this without explicit pointer manipulation
- Disadvantages
  - More work to pass non-variable arguments
    - Examples: constant, function result
  - May be hard to tell if function modifies arguments
  - Introduces aliasing

Aliasing

- We say that two names are aliased if they refer to the same object 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... */
```

Call-by-Reference (cont.)

- Call-by-reference is still around (e.g., C++)

```
int x = 0;                      // C++
void f(int& y) { y = 1; }       // y = reference var
f(x); printf("%d\n", x);        // prints 1
f(2);                           // error
```

- Seems to be less popular in newer languages
  - Older languages still use it
    - Examples: Fortran, Ada
  - Possible efficiency gains not worth the confusion
    - Pointer notation, if you've got it, seems easier
  - 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

- Cbv is 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
  - Call-by-reference can sometimes give different results
- A note about Java
  - Language is call by value
  - But most parameters are object references anyway
    - Still have aliasing, parameter modifications at object level

Call-by-Name (cont.)

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

```
let add x y = x + y

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

OCaml; cbv; arguments evaluated here

```
add (add 3 1) (add 4 1) ->
add 4 (add 4 1) ->
add 4 5 ->
4 + 5 -> 9
```

Haskell; cbn; arguments evaluated here

```
add x y = x + y

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

```
add (add 3 1) (add 4 1) ->
(add 3 1) + (add 4 1) ->
4 + (add 4 1) ->
4 + 5 -> 9
```


Call-by-Name (cont.)

- 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; cbv; 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; cbn; never evaluated because parameter is never used