Parameter Passing & Function Calls

Overview

- Parameter Passing
  - Call by value
  - Call by reference
  - Call by name
    - Eager vs. lazy evaluation
- Function calls
- Tail recursion
- Short circuiting

Parameter Passing in OCaml

What value is bound to z?

```ocaml
let add x y = x + y
let z = add 3 4
```

```ocaml
let add x y = x + y
let z = add (add 3 1) (add 4 1)
```

```ocaml
let z = ref 0;
let add x y = (!z) + x + y
let set_z () = z := 3; 1
let z = add (set_z ()) 2
```

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

```c
int r = 0;
int add(int x, int y) { return x + y; }
int set_x(void) {
    r = 3;
    return 1;
}
add(set_x(), 2);
```
Call-by-Value in Imperative Languages

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

```c
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

```c
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

```c
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)

```c
int x;
int *p, *q; /*Note that C uses pointers to simulate call by reference*/
p = &x; /* *p and x are aliased */
q = p; /* *q, *p, and x are aliased */
```

```c
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++)

```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, C with pointers
  • Possible efficiency gains not worth the confusion
    >> 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

Differences blurred for languages like Java

- Language is call-by-value
- But most parameters are object references anyway
  >> Still have aliasing, parameter modifications at object level

Call-by-Name

- Call-by-name (cbn)
  • First described in description of Algol (1960)
  • Generalization of Lambda expressions
  • Idea: In a function:
    Example: Let add x y = x+y
    add (a+b) (c+d) = (a+b) + (c+d) ◄ executed function

Then each use of x and y in the function definition is just a literal substitution of the actual arguments, (a+b) and (c+d), respectively

• Implementation: Highly complex, inefficient, and provides little improvement over other mechanisms
**Call-by-Name (cont.)**

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

```ocaml
let add x y = x + y
let z = add (add 3 1) (add 4 1)
```

```
add x y = x + y
z = add (add 3 1) (add 4 1)
```

**OCaml; cbv; arguments evaluated here**

**Haskell; cbn; arguments evaluated here**

What would be an example where this difference matters?

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

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

- **OCaml; cbv; infinite recursion at call**
- **Haskell; cbn; never evaluated because parameter is never used**

---

**Call by Name Examples**

- **P(x) {x = x + x;}**
  - What is: $Y = 2; \quad P(Y); \quad write(Y)$
  - $\Rightarrow$ becomes $Y = Y + Y = 4$

- **F(m) {m = m + 1; return m;}**
  - What is: $\int A[10]; \quad m = 1; \quad P(A[F(m)])$
  - becomes $P(A[F(m)]) = A[F(m)] + A[F(m)]$
  - $A[m++] = A[m++] + A[m++]$

---

**Call by Name Anomalies**

- Write a function to exchange values of X and Y

  - **Usual way - swap(x,y) { t=x; x=y; y=t; }**
    - Cannot do it with call by name!

  - **Reason**
    - Cannot handle both of following
      - $\rightarrow$ swap($A[m]; m$)
      - $\rightarrow$ swap($m, A[m]$)
    - One of these must fail
      - $\rightarrow$ swap($A[m]; m \rightarrow t = A[m]; A[m] = m; m = t$;)
      - $\rightarrow$ swap($m, A[m]; \rightarrow t = m; m = A[m]; A[m] = t; \quad //\textit{fails!}$
Two Cool Things to Do with CBN

- CBN is also called lazy evaluation
  - CBV is also known as eager evaluation

- Build control structures with functions
  ```ml
  let cond p x y = if p then x else y
  ```

- Build “infinite” data structures
  ```ml
  let integers n = n::(integers (n+1)) (* infinite loop in cbv *)
  ```

Simulating CBN with CBV

- Thunk
  - A function with no arguments

- Algorithm
  1. Replace arguments $a_1, \ldots, a_k$ by thunks $t_1, \ldots, t_k$
     - When called, $t_i$ evaluates and returns $a_i$
  2. Within body of the function
     - Replace formal argument with thunk invocations

```ml
let add1 x = x + 1 in add1 (2 + 3)
```

Simulating CBN with CBV (cont.)

- becomes...

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

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

```c
int i = 1;
void p(int f, int g) {
    g++;
    f = 5 * i;
}

int main() {
    int a[] = {0, 1, 2};
    p(a[1], i);
    printf("%d %d %d %d\n", i, a[0], a[1], a[2]);
}
```
Example: Call-by-Value

```c
int i = 1;
void p(int f, int g) {
    g++;
    f = 5 * i;
}
int main() {
    int a[] = {0, 1, 2};
p(a[i], i);
    printf("%d %d %d %d\n", i, a[0], a[1], a[2]);
}
```

Example: Call-by-Reference

```c
int i = 1;
void p(int f, int g) {
    g++;
    f = 5 * i;
}
int main() {
    int a[] = {0, 1, 2};
p(a[i], i);
    printf("%d %d %d %d\n", i, a[0], a[1], a[2]);
}
```

Example: Call-by-Name

```c
int i = 1;
void p(int f, int g) {
    g++;
    f = 5 * i;
}
int main() {
    int a[] = {0, 1, 2};
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 cbv)
  - 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
CBV versus CBN

- CBN is flexible-strictly more programs terminate
  - E.g., where we might have an infinite loop with cbv, we might avoid it with cbn by waiting to evaluate
- Order of evaluation is really hard to see in CBN
  - 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 (the unevaluated argument) 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)

How Function Calls Really Work

- Function calls are important
  - Usually have direct instruction support in hardware
    - Detail important for assembly language programming
  - See CMSC 212, 311, 412, or 430
- Will just provide quick overview here
- Key point to remember
  - Function calls generally require allocating stack frames

Stack Frame / Activation Record

- Machine-dependent data structure containing state information for each function invocation
  - Allocated on stack at function invocation
  - Freed upon function return (by popping stack)
- Contents may include
  - Local variables
  - Return address
  - Actual parameters
  - Return value
  - Address of frame of calling function
  - Address of frame of lexically enclosing function

Machine Model (Generic UNIX)

- The text segment contains the program’s source code
- The data segment contains global variables, static data (data that exists for the entire execution and whose size is known), and the heap
- The stack segment contains the activation records for functions
Machine Model (x86)

- The CPU has a fixed number of registers
  - Think of these as memory that's really fast to access
  - For a 32-bit machine, each can hold a 32-bit word

- Important x86 registers
  - eax: generic register for computing values
  - esp: pointer to the top of the stack
  - ebp: pointer to start of current stack frame
  - eip: the program counter (points to next instruction in text segment to execute)

The x86 Stack Frame/Activation Record

- The stack just after f transfers control to g
  - previous frames
  - return instruction ptr
  - ebp for caller of f
  - frame boundary
  - f's locals, saves
  - parameters for g
  - ebp
  - esp

Based on Fig 6-1 in Intel ia-32 manual

x86 Calling Convention

- To call a function
  - Push parameters for function onto stack
  - Invoke CALL instruction to
    - Push current value of eip onto stack
      - i.e., save the program counter
    - Start executing code for called function
  - Callee pushes ebp onto stack to save it

x86 Calling Convention (cont.)

- When a function returns
  - Put return value in eax
  - Invoke LEAVE to pop stack frame
    - Set esp to ebp
    - Restore ebp that was saved on stack and pop it off the stack
  - Invoke RET instruction to load return address into eip
    - i.e., start executing code where we left off at call
Lots More Details

- A whole lot more to say about calling functions
  - Local variables are allocated on stack by the callee as needed
    - This is usually the first thing a called function does
  - Saving registers
    - If the callee is going to use eax itself, you’d better save it to the stack before you call
  - Passing parameters in registers
    - More efficient than pushing/popping from the stack
  - Etc...
- Details covered in other courses

Tail Calls

- A tail call is a function call that is the last thing a function does before it returns
  - Not just function call in last line of code in function

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

```ocaml
let rec len = function
    | [] -> 0
    | (_::t) -> 1 + (len t) (* not tail call, performs +1 *)
```

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

Tail Recursion

- Recall that in OCaml, all looping is via recursion
  - Seems very inefficient
  - Needs one stack frame for each recursive call
- A function is tail recursive
  - If it is recursive
  - And recursive call is a tail call
- If function is tail recursive
  - Can reuse stack frame for each recursive call

Tail Recursion (cont.)

```ocaml
let rec len 1 = match 1 with
    | [] -> 0
    | (_,t) -> 1 + (len t)

len [1; 2]
```

```
| [1;2]  |
| [2]    |
| []     |
```

- Function is not tail recursive
  - Use stack frame store return value
  - Add 1 to return value, use as new return value
Tail Recursion (cont.)

Function is tail recursive
- Same stack frame can be reused for the next call
- Since we’d just pop it off and return anyway

let rec len a l = match l with
| [] -> a
| (x::t) -> (len (a + 1) t)
len 0 [1; 2]

eax: 2

Short Circuiting

Will OCaml raise a Division_by_zero exception?

- 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

Short Circuiting (cont.)

- C, C++, Java, and Ruby all short-circuit &&, ||
- But some languages don’t, like Pascal (although Turbo Pascal has an option for this):

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

- So this would need to be written as

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