CMSC 330: Organization of Programming Languages

Parameter Passing
Programming Language Features (cont.)

- Names & binding
  - Namespaces
  - Static (lexical) scopes
  - Dynamic scopes

- Parameter passing
  - Call by value
  - Call by reference
  - Call by name
    - Eager vs. lazy evaluation

- Polymorphism
  - Parametric
  - Subtype
  - Ad-hoc

- Parallelism
  - Multithreading
  - Message passing
Parameter Passing in OCaml

What value is bound to \( z \)?

\[
\text{let } \text{add } x \ y = x + y \\
\text{let } z = \text{add } 3 \ 4
\]

\[
\text{let } \text{add } x \ y = x + y \\
\text{let } z = \text{add } (\text{add } 3 \ 1) \ (\text{add } 4 \ 1)
\]

\[
\text{let } r = \text{ref } 0 \\
\text{let } \text{add } x \ y = (!r) + x + y \\
\text{let } \text{set}_r () = r := 3; 1 \\
\text{let } z = \text{add } (\text{set}_r ()) \ 2
\]

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 \texttt{let x = e1 in e2}, the expression \texttt{e1} is fully evaluated before \texttt{e2} is evaluated
- C, C++, and Java also use call-by-value

```c
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

```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);
}
```

<table>
<thead>
<tr>
<th>x</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>3</td>
</tr>
</tbody>
</table>
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:
  
  Let \( add \ x \ y = x+y \)
  
  \( add \ (a*b) \ (c*d) \)

  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

Example:

\[
add \ (a*b) \ (c*d) = \ (a*b) + (c*d) \quad \rightarrow \quad \text{executed function}
\]
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)
```

**OCaml; cbv; arguments evaluated here**

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

**Haskell; cbn; arguments evaluated here**
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
Call by Name Examples

- P(x) {x = x + x;}
  What is: Y = 2;
  P(Y);
  write(Y)
  \[ \rightarrow \text{becomes } Y = Y + Y = 4 \]

- F(m) {m = m + 1; return m;}
  What is: int A[10];
  m = 1;
  P(A[F(m)])
  \[ \rightarrow \text{becomes } P(A[F(m)]) \]
  \[ \rightarrow A[F(m)] = A[F(m)] + A[F(m)] \]
  \[ \rightarrow A[m++] = A[m++] + A[m++] \]
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
    - `swap(A[m], m)`
    - `swap(m, A[m])`
  
  - One of these must fail
    - `swap(A[m], m) → t = A[m] ; A[m] = m; m = t;`  // fails!
    - `swap(m, A[m]) → t = m ; m = A[m]; A[m] = t;`
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

  ```
  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 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

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

```ocaml
let add1 x = x() + 1 in add1 (fun () -> (2+3))
```
Simulating CBN with CBV (cont.)

```
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...

```
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)
```

- Get 1st arg
- Return 2nd arg
- Never invoked

Parameters are now thunks
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[i], 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;
    a[i] = 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]);
}
```

<table>
<thead>
<tr>
<th>i</th>
<th>a[0]</th>
<th>a[1]</th>
<th>a[2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

```
2
2
```

The expression `a[i]` isn't evaluated until needed, in this case after `i` has changed.
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
How Function Calls Really Work

- Function calls are important
  - Usually have direct instruction support in hardware
  - Detail important for assembly language programming
    - See CMSC 216, 411, 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

![Stack Frame Diagram]
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

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

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

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.)

- Function is not tail recursive
  - Use stack frame store return value
  - Add 1 to return value, use as new return value

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

len [1; 2]
```

eax: 2
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

```ocaml
let rec len a l = match l with
  | []    -> a
  | _::_:t -> (len (a + 1) t)
len 0 [1; 2]
```
Short Circuiting

- Will OCaml raise a Division_by_zero exception?

```ocaml
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 "OCaml sure 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
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):

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

- So this would need to be written as

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