Agenda

• Function implementation
• Evaluation strategies
Runtime organization

- The address space of a process is divided into segments:
  - code (aka text)
  - global data
  - heap
  - stack

- Details depend on the OS
An aside on bits

• Recent operating systems (e.g., Windows 7, OS X 10.6) are **64-bit**

• What are the advantages?

• What are the disadvantages?
Call stack

- Each thread in a process maintains a call stack
- Stack frame (aka activation record) contains:
  - function parameters
  - saved frame pointer of caller
  - saved return address of caller
  - local variables

- Stack frame pushed when calling a function
- Stack frame popped on return
**Stack frame**

**fp** - frame pointer
points to current frame

**sp** - stack pointer
points to current top of stack

**saved ip** - saved return address

**saved fp** - saved frame pointer of caller
aka *dynamic link*

sometimes also: **static link** - pointer to frame of lexically enclosing function (this is not used by most languages nowadays)

**callee frame** can access args from **caller frame**
Call

• Before call:
  • caller allocates callee stack frame
  • caller evaluates and stores parameters in registers or on stack
  • caller stores return address in register or on stack

• Prologue:
  • callee stores frame pointer in stack
  • callee set fp to be top of stack
  • callee allocates local variables (bumping sp)
Return

- Epilogue:
  - callee stores return value in register or on stack
  - callee restores fp
  - callee jumps to return address

- After return:
  - caller copies return value out
  - caller deallocates callee frame
def g(y, z) = y + z

def f(x, y) = {
    val a = 4
    val b = g(x+a, 2)
    b + y
}

val x = 3

val x = 3
f(x, 1)
def g(y, z) = y + z

def f(x, y) = {
    val a = 4
    val b = g(x+a, 2)
    b + y
}

val x = 3

⇒ f(x, 1)
def g(y, z) = y + z

def f(x, y) = {
    val a = 4
    val b = g(x+a, 2)
    b + y
}

val x = 3

f(x, 1)

| saved ip | - |
| saved fp | - |
| x         | 3 |
| param 1   | 3 |
| param 2   | 1 |
def g(y, z) = y + z

def f(x, y) = {
    val a = 4
    val b = g(x+a, 2)
    b + y
}

val x = 3

f(x, 1)
def g(y, z) = y + z

def f(x, y) = {
  val a = 4
  val b = g(x+a, 2)
  b + y
}

val x = 3
f(x, 1)

<table>
<thead>
<tr>
<th>saved ip</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>saved fp</td>
<td>-</td>
</tr>
<tr>
<td>x</td>
<td>3</td>
</tr>
<tr>
<td>param 1</td>
<td>3</td>
</tr>
<tr>
<td>param 2</td>
<td>1</td>
</tr>
<tr>
<td>saved ip</td>
<td>&lt;after f(x, 1)&gt;</td>
</tr>
<tr>
<td>saved fp</td>
<td>1284</td>
</tr>
<tr>
<td>a</td>
<td>-</td>
</tr>
<tr>
<td>b</td>
<td>-</td>
</tr>
</tbody>
</table>
def g(y, z) = y + z

def f(x, y) = {
  val a = 4
  val b = g(x+a, 2)
  b + y
}

val x = 3
f(x, 1)
def g(y, z) = y + z

def f(x, y) = {
    val a = 4
    val b = g(x+a, 2)
    b + y
}

val x = 3
f(x, 1)
def g(y, z) = y + z

def f(x, y) = {
    val a = 4
    val b = g(x+a, 2)
    b + y
}

val x = 3
f(x, 1)
def g(y, z) = y + z

def f(x, y) = {
  val a = 4
  val b = g(x+a, 2)
  b + y
}

val x = 3
f(x, 1)
def g(y, z) = y + z

def f(x, y) = {
    val a = 4
    val b = g(x+a, 2)
    b + y
}

val x = 3
f(x, 1)
def g(y, z) = y + z

def f(x, y) = {
    val a = 4
    val b = g(x+a, 2)
    b + y
}

val x = 3
f(x, 1)
def g(y, z) = y + z

def f(x, y) = {
  val a = 4
  val b = g(x+a, 2)
  b + y
}

val x = 3
f(x, 1)
def g(y, z) = y + z

def f(x, y) = {
    val a = 4
    val b = g(x+a, 2)
    b + y
}

val x = 3
f(x, 1)
Tail calls

- Recursive functions very common in functional languages

- Tail call:
  - call is last operation before return

- Can optimize:
  - before pushing new stack frame, pop the current frame
  - avoids explosive stack growth
  - essential in functional languages since they use lots of recursion
Tail calls

def fact(n) = {
    if (n <= 1) 1
    else fact(n-1)*n
}
?
Tail calls

def fact(n) = {
    if (n <= 1) 1
    else fact(n-1)*n
}

No.

after return from fact(n-1),
result is multiplied with n
Tail calls

def fact(n) = {
    if (n <= 1) 1
    else n*fact(n-1)
}

Tail calls

```python
def fact(n) = {
    if (n <= 1) 1
    else n*fact(n-1)
}
```

No.

after return from fact(n-1),
result is multiplied with n
Tail calls

def fact(n) = fact2(n, 1)

def fact2(n, acc) = {
    if (n <= 1) acc
    else fact2(n-1, acc*n)
}

Yes.
Tail calls

def fact(n) = fact2(n, 1)

def fact2(n, acc) = {
    if (n <= 1) acc
    else fact2(n-1, acc*n)
}

Compiler can rewrite as a loop:

def fact2(n, acc) = {
    while (n > 1) {
        n = n-1
        acc = acc*n
    }
    acc
}
Tail calls

```python
def even(n) = {
    if (n == 0) true
    else odd(n-1)
}

def odd(n) = {
    if (n == 1) false
    else even(n-1)
}
```

Yes.

Can’t (easily) rewrite as a loop, but can do *tail call optimization*. 
Tail call optimization

• Compiler can optimize tail calls as follows:
  • first pop the current stack frame
  • push the arguments to the call
  • push the return address (== the return address of the caller)
  • make the call

• If making a tail call to the same function, can just overwrite the arguments and jump to the function entrypoint
Without TCO

def fact(n) = fact2(n, 1)

def fact2(n, acc) = {
    if (n <= 1) acc
    else fact2(n-1, acc*n)
}

<table>
<thead>
<tr>
<th>param 1 (n)</th>
<th>param 2</th>
<th>saved ip</th>
<th>saved fp</th>
<th>param 1 (n)</th>
<th>param 2</th>
<th>saved ip</th>
<th>saved fp</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>-</td>
<td>-</td>
<td>1</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
def fact(n) = fact2(n, 1)
def fact2(n, acc) = {
    if (n <= 1) acc
    else fact2(n-1, acc*n)
}
def fact(n) = fact2(n, 1)

def fact2(n, acc) = {
    if (n <= 1) acc
    else fact2(n-1, acc*n)
}

<table>
<thead>
<tr>
<th></th>
<th>param 1 (n)</th>
<th>param 2</th>
<th>saved ip</th>
<th>saved fp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1288</td>
<td>2</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1284</td>
<td></td>
<td>3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1280</td>
<td></td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
def fact(n) = fact2(n, 1)
def fact2(n, acc) = {
    if (n <= 1) acc
    else fact2(n-1, acc*n)
}
TCO

- TCO is done in nearly all functional language implementations
- CLR (.NET) supports TCO
- JVM does not do TCO
### First-class functions

- **Scala**
  - `(x:Int) => x+1`
  - `(x:Int, y:Int) => x+y`
  - `(x:Int) => (y:Int) => x+y`

- **Scheme**
  - `(lambda (x) (+ x 1))`
  - `(lambda (x y) (+ x y))`
  - `(lambda (x) (lambda (y) (+ x y))`

- **OCaml**
  - `fun x -> x+1`
  - `fun (x, y) -> x+y`
  - `fun (x) -> fn(y) -> x+y`

- **Python**
  - `lambda x: x+1`
  - `lambda x, y: x+y`
  - `lambda x: lambda y: x+y`

*aka anonymous functions*
*aka closures*
*aka higher-order functions*
**Binding**

- Variables are either *bound* or *free* in a given expression

  - \( x \) is free
    - \( \text{fun } x \rightarrow x \)
      - \( x \) is bound
    - \( \text{fun } x \rightarrow x + y \)
      - \( x \) is bound, \( y \) is free

- A function can *capture* a free variable in its scope

  - \( \text{fun } x \rightarrow x + y \)
    - captures \( y \), but not \( x \)
  - \( \text{fun } y \rightarrow \text{fun } x \rightarrow x + y \)
    - does not capture \( y \) or \( x \)
Closures

• Aka “lambda”, “anonymous function”
• Object that represents a function at run time
  • first-class
  • does not necessarily have a name
    • \( \text{fun } x \rightarrow x + 1 \)
  • but can be bound to a name
    • \( \text{let add1 = fun } x \rightarrow x + 1 \)

• Closures:
  • \textit{capture} free variables in their context
  • closure is represented as a pair: (environment, code)
let x = 1
let y = 2
fun a -> a*x + y

Environment:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>y</td>
<td>2</td>
</tr>
</tbody>
</table>

Closure:

- env
- code

compiled("fun a -> a*env.x + env.y")

Note: since names refer to values, not locations, the environment is constant.
Closure implementation

- Environment is just a stack frame allocated on the heap.
- Must be on the heap since the closure might outlive the function that created it.
Lambda calculus

• Only 3 constructs:
  • variables \( x \)
  • functions (abstractions) \( \lambda x. e \)
    • a function that has a single formal parameter \( x \) and a body \( e \)
  • function application \( (e_1 e_2) \)
    • invokes function \( e_1 \) on argument \( e_2 \)

• The only values are functions
• Can encode all other languages in the lambda calculus
Function application

- **Beta reduction**
  - \((\lambda x. \; e_1) \; e_2 \longrightarrow e_1[e_2/x]\)

- Read \(e_1[e_2/x]\) as “\(e_1\) with \(e_2\) for \(x\)”
  - Replace free occurrences of \(x\) in \(e_1\) with \(e_2\)
  - (will define this more precisely next week)

- **Identity function**
  - \(\lambda x. \; x\)
  - \((\lambda x. \; x) \; e \longrightarrow x[e/x] = e\)
Evaluation strategies

- The lambda calculus does not define an evaluation strategy
  - evaluation order is *nondeterministic*

<table>
<thead>
<tr>
<th>(λ x. λ y. + x y) (/ 6 2) (* 4 2)</th>
<th>(λ x. λ y. + x y) (/ 6 2) 8</th>
<th>(λ y. + (/ 6 2) y) (* 4 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(λ x. λ y. + x y) 3 (* 4 2)</td>
<td>(λ x. λ y. + x y) (/ 6 2) 8</td>
<td>(λ y. + (/ 6 2) y) (* 4 2)</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>+ (/ 6 2) (* 4 2)</td>
</tr>
<tr>
<td>(λ y. + 3 y) 8</td>
<td>(λ x. λ y. + x y) 3</td>
<td>(λ y. + (/ 6 2) y) 8</td>
</tr>
<tr>
<td>(λ x. λ y. + x y) 3</td>
<td>(λ y. + 3 y) (* 4 2)</td>
<td>(λ y. + 3 y) 8</td>
</tr>
<tr>
<td>(λ y. + 3 y) 8</td>
<td>8</td>
<td>(λ y. + 3 y) 8</td>
</tr>
<tr>
<td>+ 3 (* 4 2)</td>
<td>(λ y. + 3 y) 8</td>
<td>(λ y. + 3 y) + 3 (* 4 2)</td>
</tr>
<tr>
<td>+ 3 (* 4 2)</td>
<td>(λ y. + 3 y) + 3 (* 4 2)</td>
<td>+ (/ 6 2) + 3 (* 4 2)</td>
</tr>
<tr>
<td>8</td>
<td>+ (/ 6 2) + 3 (* 4 2)</td>
<td>8</td>
</tr>
</tbody>
</table>

+ 3 8

11

- **Note:** since all there are no side effects, the order of evaluation does not matter
CBV vs. CBN

• In lambda calculus: two strategies

  • *call-by-value*
    • evaluate function arguments before making a call
    • aka *eager evaluation*
    • aka *strict evaluation*

  • *call-by-name*
    • evaluate function call before evaluating arguments to the call
    • aka *lazy evaluation*
In practice

• Most languages support call-by-value

• Some languages also support call-by-name
  • Scala
  • Macros (actually call-by-denotation)

• Haskell (actually call-by-need)
C macros

- `#define F(a) (a*a)`
- `int x = 0;`
- `int y = F(++x);`
- `printf("%d %d", x, y); // prints 2 4, not 1 1`

- `F(++x)` expands to `(++x) * (++x)`

- Actually *call-by-denotation*
  - parameters are textually replaced
  - `F(a+b)` expands to `a+b*a+b == a+(b*a)+b`
    
    != `(a+b)*(a+b)`

- C macros are *unhygienic*. 
Can declare a call-by-name formal parameter using type => T

def repeat(n: Int)(body: => Unit) = {
    for (i <- 1 to n) {
        body
    }
}

repeat (10) {
    println("hello");
}

Can build your own control structures!
def withLock(body: => T): T {
    try {
        theLock.lock
        body
    }
    finally {
        theLock.unlock
    }
}

withLock { value = value + 1 } // atomic increment
def debug(msg: => String) = {
  if (debugEnabled)
    println(msg);
}

if debugEnabled is false, debug(bigArray.toString) will not call toString, which might be very expensive
def assert(test: => Boolean) = {
    if (assertionsEnabled)
        if (! test)
            throw new AssertionException();
}

if assertionsEnabled is false,
assert(x > 0) will not evaluate the test
An aside about assertions

In Java, assertions are guarded by a command-line option:

```java
java -ea MainClass
```

runs `MainClass` with assertions enabled

If assertions are not enabled the `assert` statement has no effect–the run-time compiler doesn’t generate any code for them

Useful idiom:

```java
boolean assertions = false;
assert assertions = true;
/* assertions == true only if running with -ea */
```
Call-by-need

- Languages with lazy evaluation implement call-by-need
- Semantics is the same as call-by-name, except each expression is evaluated at most once
- Example: Haskell

Summary:

<table>
<thead>
<tr>
<th></th>
<th>$(\lambda x. + x \ x)\ E$</th>
<th>$(\lambda x. 0)\ E$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call-by-value</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Call-by-name</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Call-by-need</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Call-by-need implementation

Represent each expression as a *thunk*

- a closure that takes no arguments and *memoizes* its result.

Expression $e$ implemented as:

```javascript
var result = null;
() => { if (result == null) result = e; result }
```

Variable reference $x$ implemented as:

```javascript
x()
```
Call-by-need implementation

• Optimization:

• When possible Haskell compiler will eagerly evaluate code as long as its behavior is the same as call-by-need.

• This eliminates much of the overhead of thunking.
Call-by-reference

Really orthogonal to CBV vs. CBN.
In most languages, call-by-reference is just call-by-value with a value that’s an address

```c
int x;
f(&x);  // passes address of stack variable x
        // letting f write until the caller’s
        // frame
```

In C, C++, can take address of variable on the stack.
In Java, C#, most other safe languages, cannot—all references are to objects in the heap.
Questions?