How OCaml is compiled to a von Neumann machine

Speaker: Andrew Appel
COS 326
Princeton University
Two models for OCaml

### Interpreter

```ocaml
let rec eval (e:exp) : exp =
    match e with
    | Int_e i -> Int_e i
    | Op_e(e1,op,e2) ->
        eval_op (eval e1) op (eval e2)
    | Let_e(x,e1,e2) ->
        eval (substitute (eval e1) x e2)
    | Var_e x -> raise (UnboundVariable x)
    | Fun_e (x,e) -> Fun_e (x,e)
    | FunCall_e (e1,e2) ->
        (match eval e1
         | Fun_e (x,e) ->
             eval (Let_e (x,e2,e))
         | _ -> raise TypeError)
    | LetRec_e (x,e1,e2) ->
        (Rec_e (f,x,e)) as f_val ->
        let v = eval e2 in
        substitute f_val f
        (substitute v x e)
```

### Operational semantics

<table>
<thead>
<tr>
<th>i ∈ Z</th>
<th>i --&gt; i</th>
</tr>
</thead>
<tbody>
<tr>
<td>e1 --&gt; v1</td>
<td>e2 --&gt; v2</td>
</tr>
<tr>
<td>e1 op e2 --&gt; v</td>
<td></td>
</tr>
<tr>
<td>e1 --&gt; v1</td>
<td>e2 [v1/x] --&gt; v2</td>
</tr>
<tr>
<td>let x = e1 in e2 --&gt; v2</td>
<td></td>
</tr>
<tr>
<td>λx.e --&gt; λx.e</td>
<td></td>
</tr>
<tr>
<td>e1 --&gt; λx.e</td>
<td>e2 --&gt; v2</td>
</tr>
<tr>
<td>e1 e2 --&gt; v</td>
<td></td>
</tr>
<tr>
<td>e1 --&gt; rec f x = e</td>
<td>e2 --&gt; v2</td>
</tr>
<tr>
<td>e1 e2 --&gt; v3</td>
<td></td>
</tr>
</tbody>
</table>
Another model of computation

**computer**

/kəmˈpyooər/

*noun*

1. an electronic device for storing and processing data, typically in binary form, according to instructions given to it in a *variable program.*
John Von Neumann (1903-1957)

- Scientific achievements
  - Stored program computers
  - Cellular automata
  - Inventor of game theory
  - Nuclear physics

- Princeton Univ. & Princeton I.A.S. 1930-1957
- Known for “Von Neumann architecture” (1950)
  - In which programs are just data in the memory
Instructions are fetched from RAM
So is data
How OCaml is compiled to machine language

- Variables
- Integers
- Constant constructors
- Value-carrying constructors
- Pattern-matching
- Let x = exp in exp
- Function definition
- Function call
- Tail call

```ocaml
type t = A | B | C of int | D of t * t
```
Variables

Variables are kept in registers, just as in the translation of C programs to assembly language.

**Ocaml**

```ocaml
let x = 3 in ...
```

**Assembly language**

```assembly
move 3, r2
```

When you do a function call, variables whose values will still be needed after the call, will be stored into the stack frame, just as in the translation of C programs to assembly language.

If you have more active variables in your function than your machine has registers, some variables will be kept in the stack frame instead of registers.
Integers

The garbage collector needs to distinguish integers from pointers. OCaml does that by using the last bit of the word:

(Word-aligned) pointers end in 00 (binary)
Integers end in 1 (binary)

OCaml
let x = 3 in ...

Assembly language
move 7, r2

So, integer N is really stored as 2N+1

And, on a 64-bit-word machine, you really only get 63-bit integers
Constant constructors

```
type t =
  A | B
  | C of int | D of t*t
```

A is represented as 1 (the first odd number)
B is represented as 3 (the second odd number)

This is similar to how C programs represent NULL as 0
Value-carrying constructors

**OCaml**

```ocaml
let p = C 3 in
let q = D p p in ...```

This is similar to how C programs represent malloc’ed struct-pointers

```
type t = A | B | C of int | D of t*t
```

<table>
<thead>
<tr>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 1</td>
</tr>
</tbody>
</table>

- header
- data word(s)

how many data words 0 for “first value-carrying constructor”
Not malloc/free!

• You may be familiar with how C’s malloc/free system works
  • Malloc is somewhat expensive:
    – function call
    – find right-size block in data structure
    – update data structure, initialize header and footer
  • Free is somewhat expensive:
    – function call
    – update data structure
    – test for coalescing (?)

• OCaml (and other functional languages) have a different system
The heap and the nursery

Machine registers (and stack)

Nursery

Older generation (much larger)

base

alloc

limit
How to allocate a constructed value

Machine registers (and stack)

let q = D p p in ...

Assembly language

if r5+3>r6 goto GC

Nursery

Older generation (much larger)
How to allocate a constructed value

Machine registers (and stack)

r1
r2

r5
r6

p
q
alloc
limit

let q = D p p in ...

Nursery

base
alloc

Older generation (much larger)

store (0|2|1), r5[0]

Assembly language

if r5+3>r6 goto GC
How to allocate a constructed value

Machine registers (and stack)

r1
r2
r5
r6

p
q
alloc
limit

Nursery

base

alloc

limit

Older generation (much larger)

let q = D p p in ...

Assembly language

if r5+3>r6 goto GC
store (0|2|1), r5[0]
store r2, r5[1]
How to allocate a constructed value

Machine registers (and stack)

r1
r2
p
q
alloc
limit
r5
r6
let q = D p p in ...

Assembly language

if r5+3>r6 goto GC
store (0|2|1), r5[0]
store r2, r5[1]
store r2, r5[2]

Nursery

base
alloc
limit

Older generation (much larger)
How to allocate a constructed value

Machine registers (and stack)

r1
r2
r5
r6
p
q
alloc
limit

Base

limit
alloc

Nursery

Older generation (much larger)

let q = D p p in ...

Assembly language

if r5+3>r6 goto GC
store (0 | 2 | 1), r5[0]
store r2, r5[1]
store r2, r5[2]
add r5+1 → r3
How to allocate a constructed value

Machine
registers
(and stack)

r1
r2
r5
r6

 alloc

limit

Nursery

Older generation
(much larger)

let q = D p p in ...

Assembly language

if r5+3 > r6 goto GC
store (0|2|1), r5[0]
store r2, r5[1]
store r2, r5[2]
add r5+1 → r3
add r5+3 → r5
How to allocate a constructed value

**Type Definitions**

```plaintext
type t =
    A | B
    | C of int | D of t*t
```

**Assembly Code**

```plaintext
if r5+3>r6 goto GC
store (0|2|1), r5[0]
store r2, r5[1]
store r2, r5[2]
add r5+1 → r3
add r5+3 → r5
```

**Steps for Allocation**

1. Test for space available
2. Store the header word
3. Store first field
4. Store second field
5. Assign the result (q)
6. Adjust the “alloc” pointer

**Total Instructions**

- Test for space available: 2 instructions
- Initialize the fields: 2 instructions
- Store header word: 2 instructions
- Store first field: 2 instructions
- Store second field: 2 instructions
- Assign result (q): 2 instructions
- Adjust “alloc” pointer: 2 instructions
What happens

WHEN THE NURSERY FILLS UP . . .

GARBAGE COLLECTION!
The nursery is full

Machine registers (and stack)

r1
r2

r5
r6

alloc
limit

Nursery

base

alloc
limit

2 1

Older generation (much larger)
Only these records are reachable

Machine registers (and stack)

r1
r2
r5
r6

Nursery

base

alloc limit

Older generation (much larger)
Move reachable records to older generation

(by breadth-first search)

Machine registers (and stack)

Nursery

Older generation (much larger)

alloc limit

base
Reset “alloc” pointer of Nursery

Machine registers (and stack)

r1
r2
r5
r6

Nursery

Older generation (much larger)

base alloc

limit

Reset "alloc" pointer of Nursery
How OCaml is compiled to machine language

- Variables
- Integers
- Constant constructors
- Value-carrying constructors
  - Pattern-matching
  - Let x = exp in exp
  - Function definition
  - Function call
  - Tail call

```ocaml
type t = A | B | C of int | D of t*t
```
Pattern-matching

match x with
| A -> exp1
| B -> exp2
| C i -> exp3(i)
| D(i,j) -> exp4 i j

type t =
  A | B
  | C of int | D of t*t

Assembly language
(suppose x is in register r2)

andb r2,1 → r3
if r3=0 goto Boxed
   handle cases A,B
goto Done
Boxed:
   handle cases C,D
Done:

First, test whether the constructed value is “unboxed” (constant constructor) or “boxed” (value-carrying constructor)
Pattern-matching

match x with
| A -> exp1
| B -> exp2
| C i -> exp3(i)
| D(i,j) -> exp4 i j

type t =
    A | B
    | C of int | D of t*t

Assembly language
(suppose x is in register r2)

andb r2,1 → r3
if r3=0 goto Boxed
  (if r2=1 then exp1 else exp2)
goto Done
Boxed:
  handle cases C,D
Done:
match x with
| A -> exp1
| B -> exp2
| C i -> exp3(i)
| D(i,j) -> exp4 i j

type t =
  A | B
  | C of int | D of t*t

Assembly language
(suppose x is in register r2)

andb r2,1 → r3
if r3=0 goto Boxed
handle cases A,B
goto Done
Boxed:
load r2[-1] → r3
andb 127,r3 → r3
(if r3=0 then C else D)
Done:
Pattern-matching

match x with
  | A -> exp1
  | B -> exp2
  | C i -> exp3(i)
  | D(i,j) -> exp i j

Assembly language
(suppose x is in register r2)

D case:
  load r2[0] → r4  (fetch i)
  load r2[1] → r5  (fetch j)

type t =
  A | B
  | C of int | D of t*t
Summary of Pattern-matching

match x with
| A -> exp1
| B -> exp2
| C i -> exp3(i)
| D (i,j) -> exp4 i j

Conditional branches (or switch-statement)

Memory loads
How OCaml is compiled to machine language

- Variables
- Integers
- Constant constructors
- Value-carrying constructors
- Pattern-matching
  - Let x = exp in exp
  - Function definition
  - Function call
  - Tail call
Almost as simple as,

```
let x = y + z in ...
```

But remember, in order to make integers distinguishable from pointers, OCaml represents integers with low-order-bit 1, which is to say, \( r3 = 2y + 1 \)  \( r1 = 2z + 1 \)

and we need to compute \( r4 = 2(y+z) + 1 \)

```
Machine registers (and stack)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>r2</td>
<td>r3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>z</td>
</tr>
</tbody>
</table>
```

Assembly language

```
add r3+r1 → r4
sub r4-1 → r4
```
fun x -> x+1

More or less, a function is translated as a label in assembly language, which stands for an address in machine language, where some machine instructions implement the function:

Assembly language
f:
  add r0+2 → r0
  ret

But there is one important difference from the way C functions are compiled!
(fun w -> x+w+y)

Free variables! (in this case, x and y)

Assembly language
f:

*um, how do I know the values of x and y?*

ret
Function definitions

(fun w -> x+w+y)

Free variables! (in this case, x and y)

Assembly language

f_code:

get x and y from environment-pointer

ret
(fun w -&gt; x+w+y)

Evaluating “fun ... -&gt; ...” is like constructing two records on the heap...

... and will be garbage-collected when no longer in use.
let y = f(x) in ...

**Assembly language**

f

```
Assembly language
f_code:
get free vars from env
ret
```
Tail call

\[ f(x) \]

Assembly language

\[ f\_code: \]

\[ \text{get free vars from env} \]

\[ \text{ret} \]

**Assembly language**

\[
\begin{align*}
\text{move} & \ x \rightarrow r1 \quad \# \ \text{arg} \\
\text{load} & \ f[1] \rightarrow r2 \quad \# \ \text{env} \\
\text{load} & \ f[0] \rightarrow r3 \quad \# \ \text{code} \\
\text{jmp} & \ r3
\end{align*}
\]
Conclusion

• Each feature of the OCaml language is implemented in a few instructions of machine language

• Some of these features work just like their counterparts in C,

• What’s different:
  – garbage collection, instead of malloc/free
  – function closures
  – distinguishing integers from pointers, by low-order bit