Code generation for cool methods

Greta Yorsh

Admin

- PA1 code review
- PA3 handout and distro
- Lecture on 14/3: memory management (JN)
- Next week: check announcements on qm+
- Revision lecture on revision week

Today

- Code generation for methods
  - operation semantics of dispatch
  - cool stack frame layout

OPERATIONAL SEMANTICS OF DISPATCH

Cool evaluation rules

- The evaluation judgment is **so, E, S** $\vdash$ **e : v, S'**
  - **so** the current value of the self object
  - **E** the current variable environment
  - **S** the current store
- If the evaluation of **e** terminates then
  - the returned value is **v**
  - the new store is **S'**

More Notation

- For a class A and a method f of A
  impl(A, f) = (x_1, …, x_n, e_{body})
  - $\mathbf{x_i}$ are the names of the formal arguments
  - $\mathbf{e_{body}}$ is the body of the method
  - f can be inherited

Dispatch $\mathbf{e_0.f(e_1,\ldots,e_n)}$

- Evaluate the arguments in order $e_1,\ldots,e_n$
- Evaluate e_0 to the target object
- Let X be the dynamic type of the target object
- Find the definition of f for X
- Create n new locations
- Create an environment that maps formal arguments of f to those locations
- Initialize the locations with the actual arguments
- Set self to the target object
- Evaluate the body of f

Dispatch

$$\frac{\begin{array}{l} so, E, S \vdash e_1 : v_1, S_1 \\ so, E, S_1 \vdash e_2 : v_2, S_2 \\ \ldots \\ so, E, S_{n-1} \vdash e_n : v_n, S_n \\ so, E, S_n \vdash e_0 : v_0, S_{n+1} \\ v_0 = X(a_1 = l_1, \ldots, a_m = l_m) \\ impl(X, f) = (x_1, \ldots, x_n, e_{body}) \\ l_{xi} = newloc(S_{n+1}) \text{ for } i = 1, \ldots, n \\ E' = [x_1 : l_{x1}, \ldots, x_n : l_{xn}, a_1 : l_1, \ldots, a_m : l_m] \\ S_{n+2} = S_{n+1}[v_1/l_{x1}, \ldots, v_n/l_{xn}] \\ v_0, E', S_{n+2} \vdash e_{body} : v, S_{n+3} \end{array}}{so, E, S \vdash e_0.f(e_1, \ldots, e_n) : v, S_{n+3}}$$

Dispatch

- The body of the method is invoked with
  - E mapping formal arguments and self's attributes
  - S like the caller's except with actual arguments bound to the locations allocated for formals
- The notion of the frame is implicit: new locations are allocated for actual arguments
- The semantics of **static** dispatch is similar, except the implementation of f is taken from the specified class

Runtime errors in dispatch

- What happens if impl(X, f) is not defined?
- What happens if target object is void?

$$\frac{\begin{array}{l} \ldots \\ so, E, S_n \vdash e_0 : v_0, S_{n+1} \\ v_0 = X(a_1 = l_1, \ldots, a_m = l_m) \\ impl(X, f) = (x_1, \ldots, x_n, e_{body}) \\ \ldots \end{array}}{so, E, S \vdash e_0.f(e_1, \ldots, e_n) : v, S_{n+3}}$$

Runtime errors

- There are some runtime errors that the type checker does not try to prevent (can it ?)
  - dispatch on void
  - case on void
  - no matching branch in case
  - division by zero
  - substring out of range
  - heap overflow
- Execution must abort gracefully
  - with an error message not with segfault
- Operational rules do not cover these cases

In practice...

- Cool operational rules are **precise and detailed**

- Most languages do not have a well specified operational semantics
- When portability is important an operational semantics becomes essential
- Notation we used for Cool is very limited

COOL STACK FRAME LAYOUT

Cool stack frame

- Code for function calls and function definitions depends on the **layout of the stack frame**
  - return value: always in the accumulator $a0
  - actual parameters on the stack pushed by the caller
  - return address in $ra
  - start of the frame in $fp (callee saved)
  - address of the next location on the stack is in $sp
  - the top of the stack is at address $sp + 4
- Invariant: on exit $sp is the same as it was on function entry
  - no need to save $sp

Code generation for method calls

```
cgen(f(e1,…,en)) =
   cgen(en)
   push $a0
   …
   cgen(e1)
   push $a0
   jal f_entry
```

The caller saves the actual arguments in reverse order

The stack frame so far is 4*n bytes long

The caller places the return address in register $ra

Code generation for methods

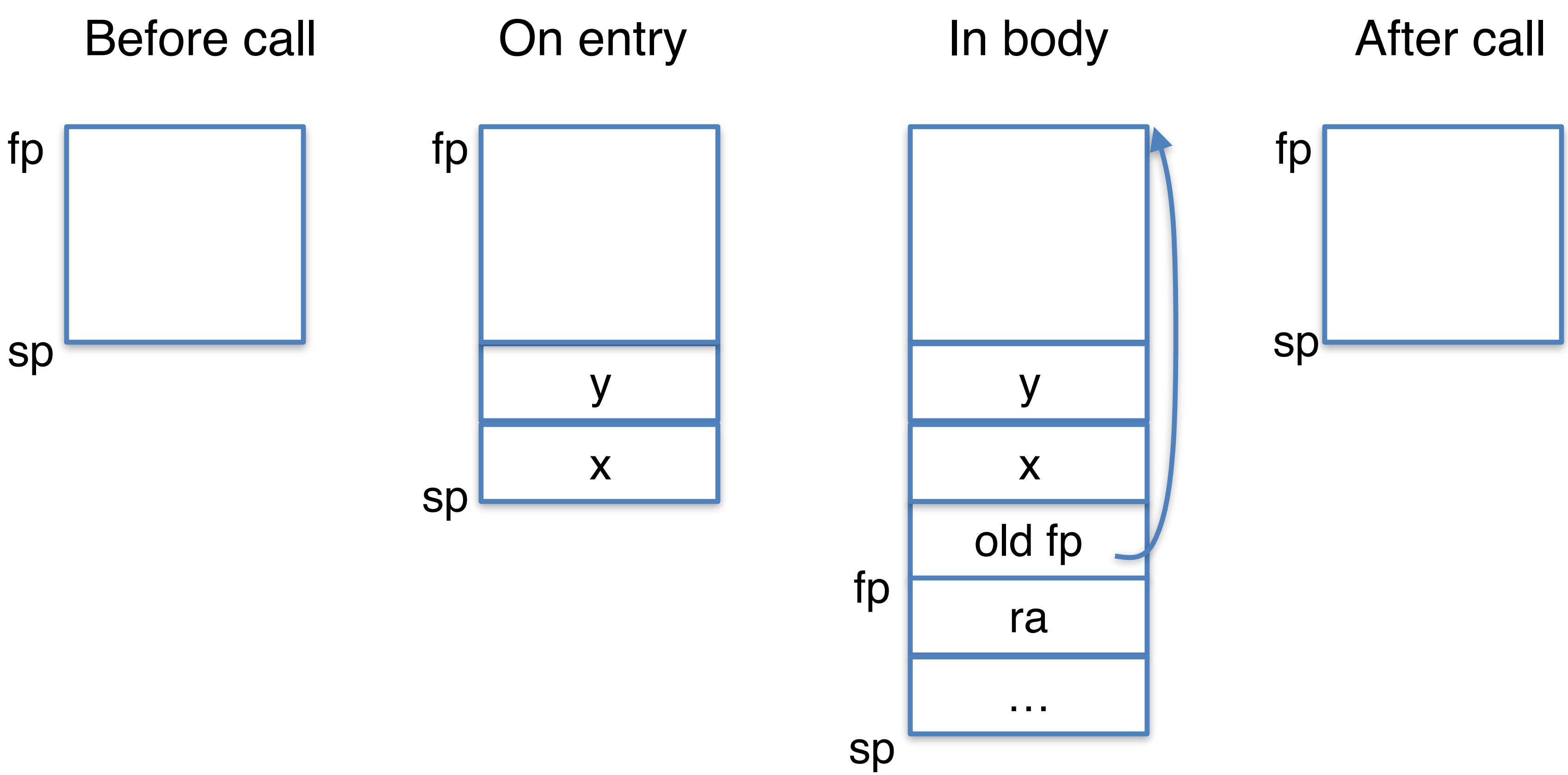
```
cgen(impl(f)=(x1,…,xn,e)) =
f_entry:
    push $fp
    move $fp $sp
    push $ra
    cgen(e)
    $ra := top
    lw $fp 4($fp)
    addiu $sp $sp z
    jr $ra
```

The frame pointer points to the top of the frame

The callee pops the return address, the saved value of the frame pointer and the actual arguments

z = 4*n + 8

Calling sequence for f(x,y)

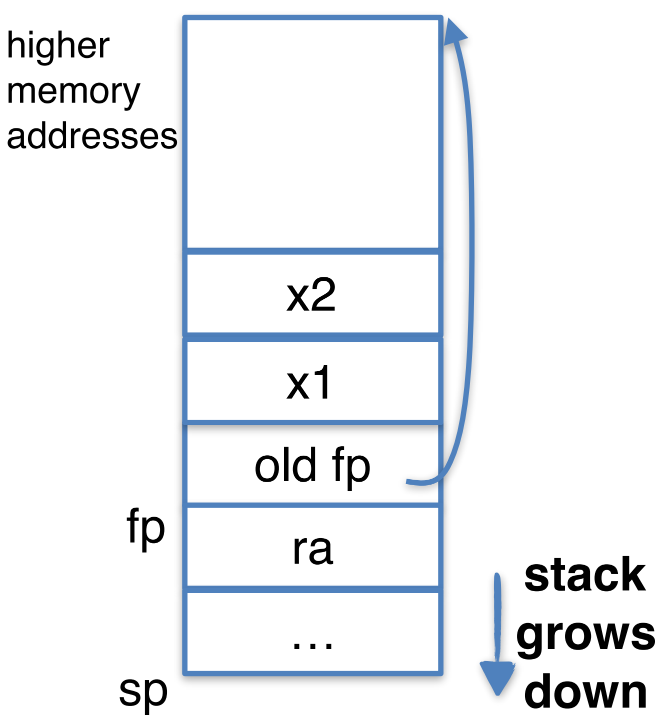


Why do we keep frame pointer?

- Stack machine with accumulator
  - the stack grows when intermediate results are saved
  - the variables are not at a fixed offset from $sp
- Frame pointer
  - always points to the return address on the stack
  - since it does not move it can be used to find the variables (function's formal parameters)

Code generation for variables

- for a function **impl(f)=(x1,x2,e)**
  frame and frame pointer are set up as follows:



x1 is at fp + 8
x2 is at fp + 12

cgen(xi) = lw $a0 z($fp)

where z = 4*(i+1)

Stack frame layout

- What intermediate values are placed on the stack?
- How many slots are needed in the frame to hold these values?

```
fib(x:Int):Int {
  if x = 1 then 0 else
     if x = 2 then 1 else
        fib(x - 1) + fib(x – 2)
     fi
  fi
};
```

How many temporaries?

- NT(e) is the number of temps needed to evaluate e
- NT(e1 + e2)
  - needs at least as many temporaries as NT(e1)
  - needs at least as many temporaries as NT(e2) + 1
- Space used for temporaries in e1 can be reused for temporaries in e2

How many temporaries?

- NT(e1 + e2) = max(NT(e1), 1 + NT(e2))
- NT(e1 - e2) = max(NT(e1), 1 + NT(e2))
- NT(if e1 then e2 else e3) = max(NT(e1), NT(e2), NT(e3))
- NT(while e1 loop e2 pool) = max(NT(e1),NT(e2))
- NT(id(e1,…,en) = max(NT(e1),…,NT(en))
- NT(int) = 0
- NT(id) = 0

Is this bottom-up or top-down?

At what point in code generation to determine the number of temporaries?

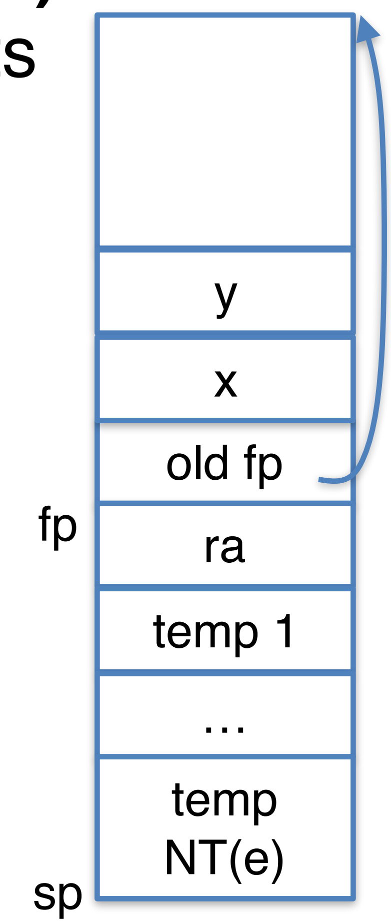
How many temporaries?

- Compute NT(e) at the beginning of code generation for each method
- What is NT(…code for fib…)?

```
fib(x:Int):Int {
  if x = 1 then 0 else
     if x = 2 then 1 else
        fib(x - 1) + fib(x – 2)
     fi
  fi
};
```

Revised stack frame

- For a function definition **impl(f)=(x1,…,xn,e)** the stack frame has **2 + n + NT(e)** elements
  - return address
  - old frame pointer
  - n arguments
  - NT(e) slots for intermediate results



Revised code generation

- How do we use the number of temporaries ?
- Code generation keeps track of how many temporaries are in use at each point
- Add a new argument **n** to code generation: the position of the next available temporary
- **cgen(e, n)** generate code for **e** and use temporaries whose address is **$fp – 4 – 4*n** or lower

Code generation for **add**

Original

```
cgen(e1 + e2) =
  cgen(e1)
  sw $a0 0($sp)
  addiu $sp $sp -4
  cgen(e2)
  lw $t1 4($sp)
  add $a0 $t1 $a0
  addiu $sp $sp 4
```

Revised

```
cgen(e1 + e2, nt) =
 cgen(e1, nt)
 sw $a0 -nt($fp)
 cgen(e2, nt + 4)
 lw $t1 -nt($fp)
 add $a0 $t1 $a0
```

The temporary area is used like a small, fixed-size stack

Code generation for methods

```
cgen(impl(f)=(x1,…,xn,e), nt) =
f_entry:
    addiu $sp $sp –(2+nt)
    sw $fp  (2+nt) ($sp)
    sw $ra (1+nt)($sp)
    addiu $fp $sp nt
    cgen(e)
    lw $ra (1+nt)($sp)
    addiu $sp $sp 4*n+8
    lw $fp 0($sp)
    j $ra
```

The frame pointer points to the top of the frame

The callee pops the return address, the saved value of the frame pointer and the actual arguments