Cool backend

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
- Cool runtime system

OPERATIONAL SEMANTICS OF DISPATCH

Cool evaluation rules

- The evaluation judgment is **so, E, S ⊢ 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})
  - **x_i** are the names of the formal arguments
  - **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

$so, E, S \vdash e_1 : v_1 , S_1$

$so, E, S_1 \vdash e_2 : v_2 , S_2$

…

$so, E, S_{n-1} \vdash e_n : v_n , S_n$

$so, E, S_n \vdash e_0 : v_0, S_{n+1}$

$v_0 = X(a_1 = l_1,\ldots, a_m = l_m)$

$impl(X, f) = (x_1,\ldots, x_n, e_{body})$

$l_{xi} = newloc(S_{n+1})$ for $i = 1,\ldots,n$

$E' = [x_1 : l_{x1}, \ldots, x_n : l_{xn}, a_1 : l_1,\ldots,a_m : l_m]$

$S_{n+2} = S_{n+1}[v_1/l_{x1},\ldots,v_n/l_{xn}]$

$v_0 , E', S_{n+2} \vdash e_{body} : v, S_{n+3}$

$so, E, S \vdash e_0.f(e_1,...,e_n) : v, S_{n+3}$

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?

$$\frac{\begin{array}{l}\ldots \\ so, E, Sn \vdash e0 : v0, Sn+1 \\ v0 = X(a1 = l1, \ldots, am = lm) \\ impl(X, f) = (x1, \ldots, xn, ebody) \\ \ldots\end{array}}{so, E, S \vdash e0.f(e1, \ldots, en) : v, Sn+3}$$

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

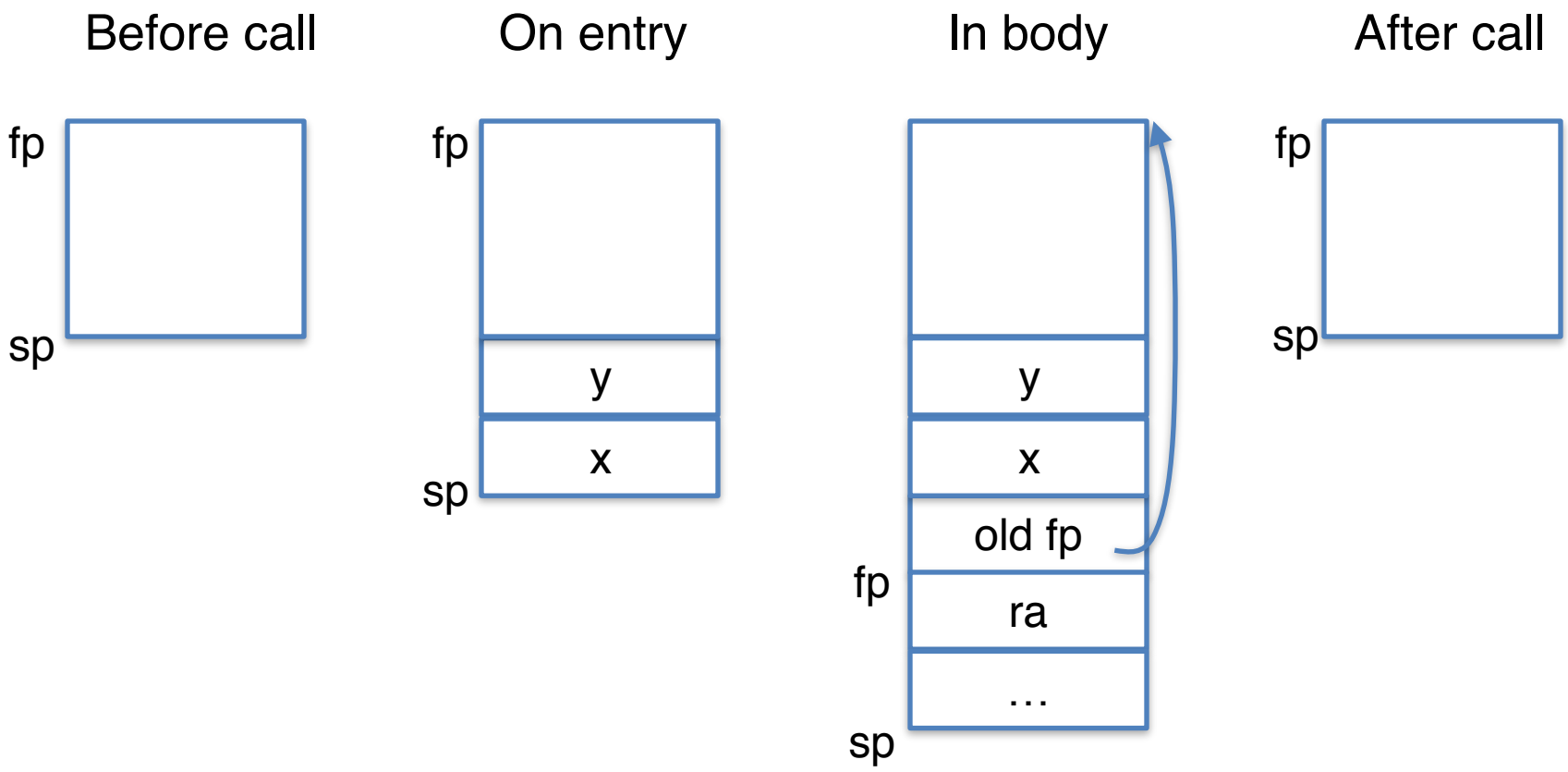
```
cgen(impl(f)=(x1,…,xn,e)) =
f_entry:
    push $fp
    move $fp $sp
    push $ra
    cgen(e)
    $ra := top
    lw $fp 4($fp)
    addiu $sp $sp z
    jr $ra
```

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)

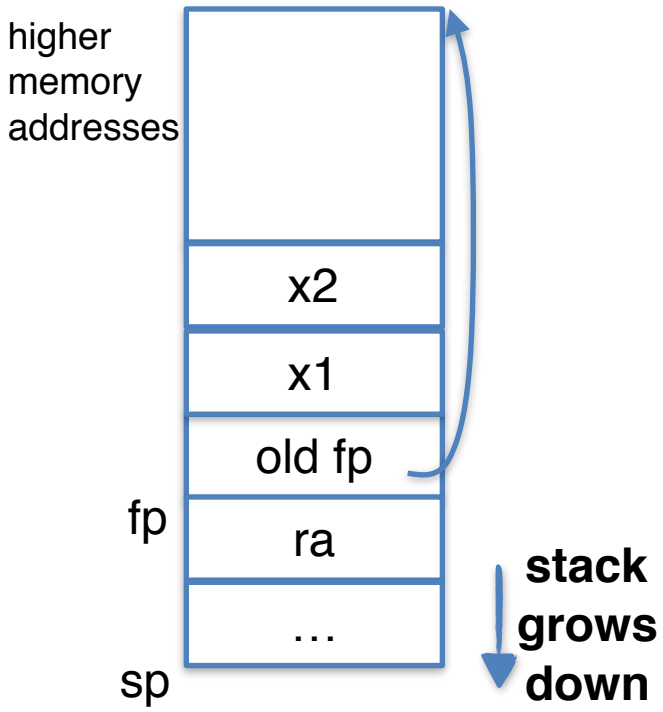


Why do we keep frame pointer?

- 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?

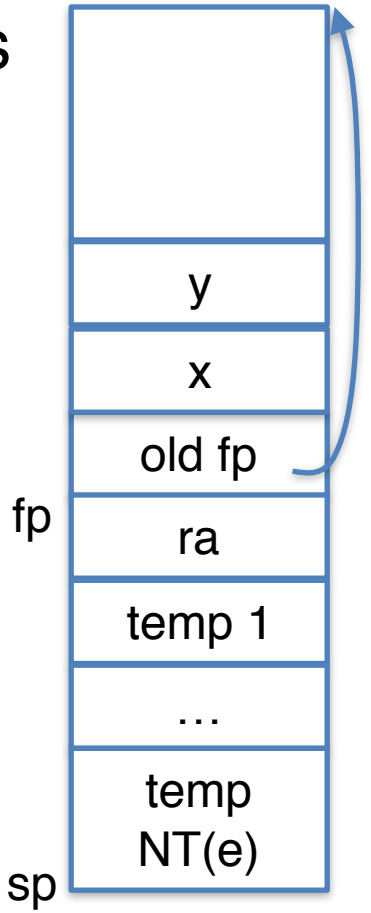
How many temporaries?

- 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

COOL RUNTIME SYSTEM

Cool Runtime System

- What does it provide ?
- What does it assume about our code ?

Runtime system: what does it **provide**?

- Garbage collector
- Start-up code to invoke main
- Code for methods of basic classes
- Utilities
  - equality
  - runtime error reporting

Start-up in trap.handler

- Create copy of Main prototype object
- Call Main_init initialize Main's parent classes and attributes of Main
- Call Main.main
  - $a0 contains Main
  - $ra contains return
- If Main.main returns, execution halts with "COOL program successfully executed"

Methods of Object

Object.copy	input: $a0 address of prototype object return: $a0 contains address of newly allocated object
Object.abort	prints the name of object in $a0 terminates execution
Object.type_name	returns in $a0 address of the string object that contains the name of the class of object passed in $a0 uses class tag and the table class_nameTab

Other methods

equality_test	args: $t1 $t2 if args have same primitive type and same value return $a0 o/w $a1
_dispatch_abort	dispatch on void print $t1 line number and $a0 filename
_case_abort	case with no match prints name of object at $a0
_case_abort2	case on void print $t1 line number and $a0 filename

System calls

Service	Code	Arguments	Result
print integer	1	$a0=integer	Console print
print string	4	$a0=string address	Console print
read integer	5		$a0=result
read string	8	$a0=string address $a1=length limit	Console read
exit	10		end of program

Runtime system: what does it **require**?

- Object layout: header, GC tag
- Global data: class name table, class object table, prototype objects, constants
- Naming conventions for labels
- Register usage
- Calling conventions for methods of runtime system
  - use the same for user-defined methods

Global data

- **Class name table** is a mapping from class tags to class names
  - used for error reporting
- **Class object table** is a mapping from class tag to address of prototype object for the class
  - used for allocation
- Prototype objects
- Constants
  - String: class names, filenames
  - Int and Bool constants

Naming conventions for labels

- Dispatch table <classname>_dispTab
- Method entry point <classname>.<method>
- Class init code <classname>_init
- Prototype object <classname>_protObj
- Integer constant int_const<Symbol>
- String constant str_const<Symbol>

Cool object prototypes and dispatch tables

```
        .word   -1
A_protObj:
        .word   5
        .word   5
        .word   A_dispTab
        .word   int_const0
        .word   int_const0
        .word   -1
B_protObj:
        .word   6
        .word   6
        .word   B_dispTab
        .word   int_const0
        .word   int_const0
        .word   int_const0
        .word   -1
C_protObj:
        .word   7
        .word   6
        .word   C_dispTab
        .word   int_const0
        .word   int_const0
        .word   int_const0
```

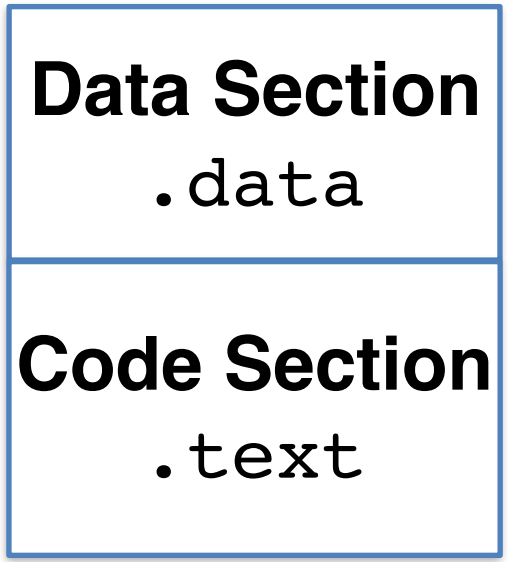
```
A_dispTab:
        .word   Object.abort
        .word   Object.type_name
        .word   Object.copy
        .word   A.f
B_dispTab:
        .word   Object.abort
        .word   Object.type_name
        .word   Object.copy
        .word   B.f
        .word   B.g
C_dispTab:
        .word   Object.abort
        .word   Object.type_name
        .word   Object.copy
        .word   A.f
        .word   C.h
```

Class A { a: Int ←0; d: Int ← 1; **f**(): Int { a ← a + d}; };
Class B inherits A { b: Int ←2; **f**(): Int {a}; **g**(): Int {a ← a - c}; };
Class C inherits A { c: Int ←3; **h**(): Int {a ← a * c}; };

Cool register conventions

- Zero $0
- ACC $a0
- SELF $s0 (callee saves)
- Stack pointer $sp
- Frame pointer $fp
- Return address $ra
- Arguments: self passed in $a0, others on the stack

Assembly file roadmap



Statically allocated data
- constants
- prototype objects
- dispatch tables

Code
- code for methods
- start-up code
- error handlers
- garbage collector

} trap.handler