Cool Type Checking
Cool Run-Time Organization

Run-Time Organization

Gentlemen, tonight we are going after the big prize. The Keeblers are paying us hansomly, but some of us might not make it back from Pepperidge farm tonight...

#1

One-Slide Summary

- We will use SELF_TYPE$_C$ for "C or any subtype of C". It shows off the subtlety of our type system and allows us to check methods that return self objects.
- The lifetime of an activation of (i.e., a call to) procedure P is all the steps to execute P plus all the steps in procedures that P calls.
- Lifetime is a run-time (dynamic) notion; we can model it with trees or stacks.

#2

Lecture Outline

- SELF_TYPE
- Object Lifetime
- Activation Records
- Stack Frames

#3

SELF_TYPE Dynamic Dispatch

- If the return type of the method is SELF_TYPE then the type of the dispatch is the type of the dispatch expression:

$$\frac{\begin{array}{c} O,M,C \vdash e_0 : T_0 \;\}A \\ \ldots \\ O,M,C \vdash e_n : T_n \;\}B \\ M(T_0, f) = (T_1',\ldots,T_n', \text{SELF\_TYPE}) \;\}C \\ T_i \leq T_i' \quad 1 \leq i \leq n \;\}D \end{array}}{O,M,C \vdash e_0.f(e_1,\ldots,e_n) : T_0}$$

#4

Where is SELF_TYPE Illegal in COOL?

m(x : T) : T' { … }

- Only T' can be SELF_TYPE! *Not T.*

What could go wrong if T were SELF_TYPE?

```
class A {  comp(x : SELF_TYPE) : Bool  {…};  };
class B inherits A {
    b() : int { … };
    comp(y : SELF_TYPE) : Bool { … y.b() …};  };
…
  let x : A ← new B in  … x.comp(new A); …
…
```

#5

Summary of SELF_TYPE

- The extended ≤ and lub operations can do a lot of the work. Implement them to handle SELF_TYPE
- SELF_TYPE can be used only in a few places. Be sure it isn't used anywhere else.
- A use of SELF_TYPE always refers to any subtype in the current class
  - The exception is the type checking of dispatch, where SELF_TYPE *as the return type* in an invoked method might have nothing to do with the current enclosing class

#6

Why Cover SELF_TYPE ?

- SELF_TYPE is a research idea
  - It adds more expressiveness to the type system
- SELF_TYPE is itself not so important
  - except for the project
- Rather, SELF_TYPE is meant to illustrate that type checking can be quite subtle
- In practice, there should be a balance between the complexity of the type system and its expressiveness

#7

Type Systems

- The rules in these lecture were Cool-specific
  - Other languages have very different rules
  - We'll survey a few more type systems later
- General themes
  - Type rules are defined on the **structure of expressions**
  - Types of variables are **modeled by an environment**
- Type systems tradeoff flexibility and safety

#8

Status

- We have covered the front-end phases
  - Lexical analysis
  - Parsing
  - Semantic analysis
- Next are the back-end phases
  - Optimization (optional)
  - Code execution (or code generation)
- We'll do **code execution** first . . .

#9

Run-time environments

- Before discussing code execution, we need to understand what we are trying to execute
- There are a number of standard techniques that are widely used for structuring executable code
- Standard Way:
  - Code
  - Stack
  - Heap

Run-Time Organization Outline

- Management of run-time resources
- Correspondence between static (compile-time) and dynamic (run-time) structures
  - "Compile-time" == "Interpret-time"
- Storage organization

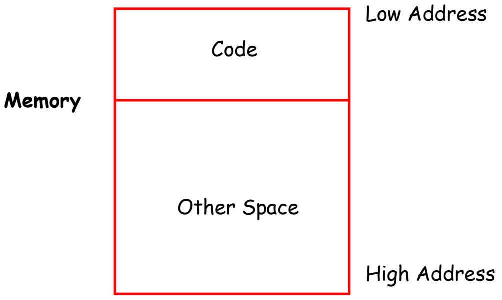
#11

Run-time Resources

- Execution of a program is initially under the control of the operating system
- When a program is invoked:
  - The OS allocates space for the program
  - The code is loaded into part of the space
  - The OS jumps to the entry point (i.e., "main")

#12

Memory Layout



#13

Notes

- Our pictures of machine organization have:
  - Low address at the top
  - High address at the bottom
  - Lines delimiting areas for different kinds of data
- These pictures are simplifications
  - e.g., not all memory need be contiguous
- In some textbooks lower addresses are at bottom

#14

What is Other Space?

- Holds all data for the program
- Other Space = Data Space
- A compiler is responsible for:
  - Generating code (that is run later)
  - Orchestrating use of the data area
- An interpreter is responsible for:
  - Executing the code directly (now)
  - Orchestrating use of the (run-time) data

#15

Code Execution Goals

- Two goals:
  - Correctness
  - Speed
- Most complications at this stage come from trying to be fast as well as correct

Assumptions about Execution

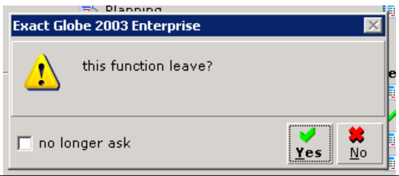
1. Execution is sequential; control moves from one point in a program to another in a well-defined order
2. When a procedure is called, control eventually returns to the point immediately after the call

Do these assumptions always hold?

#17

Activations

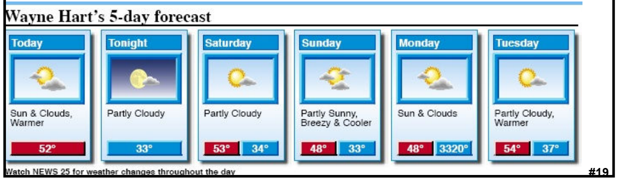
- An invocation of procedure P is an **activation** of P
- The **lifetime** of an activation of P is
  - All the steps to execute P
  - Including all the steps in procedures that P calls



#18

Lifetimes of Variables

- The **lifetime** of a variable x is the portion of execution during which x is defined
- Note that
  - Scope is a static concept
  - Lifetime is a dynamic (run-time) concept



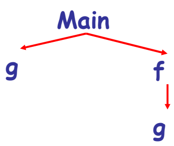
#19

Activation Trees

- Assumption (2) requires that when P calls Q, then Q returns before P does
- Lifetimes of procedure activations are properly nested
- Activation lifetimes can be depicted as a **tree**

Example

```
Class Main {
  g() : Int { 1 };
  f():  Int { g() };
  main(): Int {{ g(); f(); }};
}
```



#21

Example 2

```
Class Main {
  g() : Int { 1 };
  f(x:Int):  Int {
     if x = 0 then g() else f(x - 1) fi
  };
  main(): Int {{ f(3); }};
}
```

What is the activation tree for this example?

#22

Notes

- The activation tree depends on run-time behavior
- The activation tree may be different for every program input
- Since activations are properly nested, a **stack** can track currently active procedures
  - This is the **call stack**

#23

Example

```
Class Main {
  g() : Int { 1 };
  f():  Int { g() };
  main(): Int {{ g(); f(); }};
}
```

Main

Stack

Main

#24

Example

```
Class Main {
  g() : Int { 1 };
  f():  Int { g() };
  main(): Int {{ g(); f(); }};
}
```

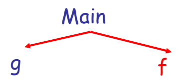
Stack

Main

g

Example

```
Class Main {
  g() : Int { 1 };
  f():  Int { g() };
  main(): Int {{ g(); f(); }};
}
```



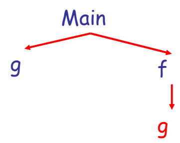
Stack

Main

f

Example

```
Class Main {
  g() : Int { 1 };
  f():  Int { g() };
  main(): Int {{ g(); f(); }};
}
```



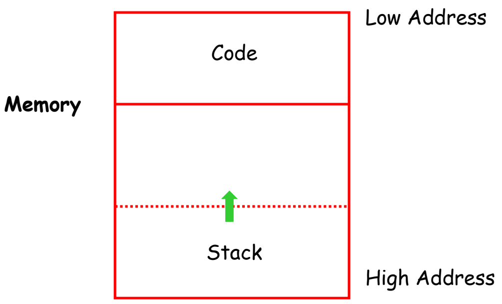
Stack

Main

f

g

Revised Memory Layout



Activation Records

- On many machines the stack starts at high-addresses and grows towards lower addresses
- The information needed to manage one procedure activation is called an **activation record** (AR) or **frame**
- If procedure F calls G, then G's activation record contains a mix of info about F and G.

What is in G's AR when F calls G?

- F is "suspended" until G completes, at which point F resumes. G's AR contains information needed to resume execution of F.
- G's AR may also contain:
  - Actual parameters to G (supplied by F)
  - G's return value (needed by F)
  - Space for G's local variables

The Contents of a Typical AR for G

- Space for G's return value
- Actual parameters
- Pointer to the previous activation record
  - The **control link** points to AR of F (caller of G)
- Machine status prior to calling G
  - Local variables
  - (Compiler: register & program counter contents)
- Other temporary values

#31

Example 2, Revisited

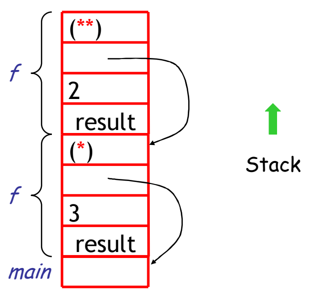
```
Class Main {
  g() : Int { 1 };
  f(x:Int):Int {
      if x=0 then g() else f(x - 1) (**) fi
  };
  main(): Int {{f(3); (*) }};}
```

AR for f:

return address
control link
argument
space for result

#32

Stack After Two Calls to f



#33

Notes

- main has no argument or local variables and its result is “never” used; its AR is uninteresting
- (*) and (**) are return addresses of the invocations of f
  - The return address is where execution resumes after a procedure call finishes
- This is only one of many possible AR designs
  - Would also work for C, Pascal, FORTRAN, etc.

#34

The Main Point

The interpreter must determine, at compile-time, the layout of activation records and execute code that correctly accesses locations in the activation record

Thus, the AR layout and the interpreter must be designed together!

#35

Discussion

- The advantage of placing the return value 1st in a frame is that the caller can find it at a fixed offset from its own frame
  - The caller must write the return address there
- There is nothing magic about this organization
  - Can rearrange order of frame elements
  - Can divide caller/callee responsibilities differently
  - An organization is better if it improves execution speed or simplifies code generation

#36

Discussion (Cont.)

- Real compilers hold as much of the frame as possible in registers
  - Especially the method result and arguments
- Why?

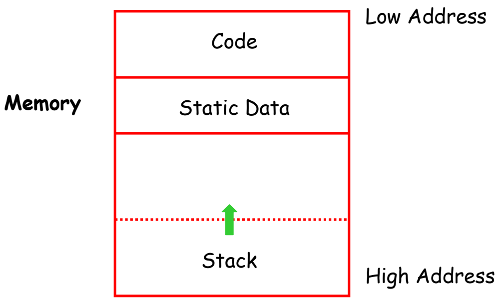
#37

Globals

- All references to a global variable point to the same object
  - Can't store a global in an activation record
    - Is this true?
- Globals are assigned a fixed address once
  - Variables with fixed address are "statically allocated"
- Depending on the language, there may be other statically allocated values

#38

Memory Layout with Static Data



#39

Heap Storage

- A value that outlives the procedure that creates it cannot be kept in the AR

method foo() { new Bar }

The Bar value must survive deallocation of foo's AR

- Languages with dynamically allocated data use a **heap** to store dynamic data

#40

Notes

- The code area contains object code
  - For most languages, fixed size and read only
- The static area contains data (not code) with fixed addresses (e.g., global data)
  - Fixed size, may be readable or writable
- The stack contains an AR for each currently active procedure
  - Each AR usually fixed size, contains locals
- Heap contains all other data
  - In C, heap is managed by *malloc* and *free*

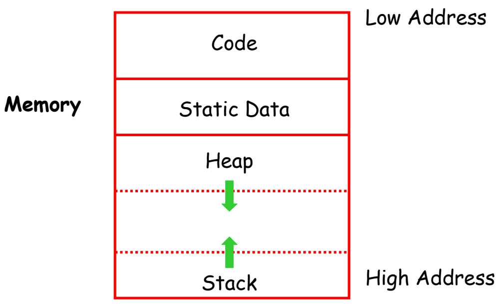
#41

Notes (Cont.)

- Both the heap and the stack grow
- Compilers must take care that they don't grow into each other
- Solution: start heap and stack at opposite ends of memory and let the grow towards each other

#42

Memory Layout with Heap



#43

Why Am I Telling You This?

- You will have to implement "something like a heap" and "something like a call stack" for your interpreter.
- You can re-use the Python/Ruby/OCaml call stack
  - No explicit return address or control link
  - Mutually-recursive procedures like "eval_exp" and "eval_method" call each other

#44

Your Own Heap

- We must support code like:
  - let x = new Counter(5) in
  - let y = x in {
  - x.increment(1);
  - print(y.getCount()); // what does this print?
  - }
- You'll need an explicit heap (as described today and also next week). A heap maps addresses (integers) to values.

#45

Homework

- WA4 due this FRIDAY at Midnight
- PA4 due Friday March 30th (17 days)
- For Thursday: Read Chapters 7.3, 9-9.3
  - Optional Stroustrup article
  - This article is often loved by students

#46