Cool Typechecking and Runtime Organization

Martin Kellogg

Agenda

- Finish discussion of SELF_TYPE
- Object Lifetimes
- Activation Records
- Stack Frames

Agenda

- **Finish discussion of SELF_TYPE**
- Object Lifetimes
- Activation Records
- Stack Frames

Recall: Why Do We Want SELF_TYPE?

```
class Count {
  i : Int <- 0;
  inc() : Count {
    {
      i <- i + 1;
      self;
    }
  };
};
```

Recall: Why Do We Want SELF_TYPE?

```
class Count {
  i : Int <- 0;
  inc() : Count {
    {
      i <- i + 1;
      self;
    }
  };
};
```

```
class Stock inherits Count {
  name() : String { … };
};
```

Recall: Why Do We Want SELF_TYPE?

```
class Count {
  i : Int <- 0;
  inc() : Count {
    {
      i <- i + 1;
      self;
    }
  };
};
```

```
class Stock inherits Count {
  name() : String { … };
};

class Main {
  a : Stock <- (new Stock).inc();
  … a.name() …
};
```

Recall: Why Do We Want SELF_TYPE?

```
class Count {
  i : Int <- 0;
  inc() : Count {
    {
      i <- i + 1;
      self;
    }
  };
};
```

```
class Stock inherits Count {
  name() : String { … };
};

class Main {
  a : Stock <- (new Stock).inc();
  … a.name() …
};
```

without SELF_TYPE, the type rules will cause a typechecking error here, because inc() returns a Count (not a Stock)

Recall: SELF_TYPE to the Rescue

- We will **extend** the type system
  - That is, make it **more expressive**

Recall: SELF_TYPE to the Rescue

- We will **extend** the type system
    - That is, make it **more expressive**
- Insight:
    - inc returns "self"
    - therefore the return value will be the same type as "self"
    - which could be **Count** *or any subtype* of **Count**
    - In the case of `(new Stock).inc()`, the type is **Stock**

Recall: SELF_TYPE to the Rescue

- We will extend the type system
  - That is, make it **more expressive**
- Insight:
  - inc returns “self”
  - therefore the return value will be the same type as “self”
  - which could be **Count** *or any subtype* of **Count**
  - In the case of `(new Stock).inc()`, the type is **Stock**
- We introduce the keyword **SELF_TYPE** to use for the return value of such functions
  - We will need to modify the type rules to handle SELF_TYPE

Recall: Typechecking SELF_TYPE (properly)

- Recall the operations that we've defined over types:
  - subtyping: $T_1 \leq T_2$
  - least upper bound: $lub(T_1, T_2)$

Recall: Typechecking SELF_TYPE (properly)

- Recall the operations that we've defined over types:
  - subtyping: $T_1 \leq T_2$
  - least upper bound: $lub(T_1, T_2)$
- To handle **SELF_TYPE** properly, we need to **extend** these operations to handle it

Recall: Typechecking SELF_TYPE (properly)

- Recall the operations that we've defined over types:
  - subtyping: $T_1 \leq T_2$
  - least upper bound: $lub(T_1, T_2)$
- To handle **SELF_TYPE** properly, we need to extend these operations to handle it
  - need to consider all four combinations of **SELF_TYPE** and "normal" types (cf. Punnett squares)
  - see last lecture's slides for the details on how we did this

Type Rules for SELF_TYPE

Type Rules for SELF_TYPE

- Since occurrences of **SELF_TYPE** depend on the **enclosing class**, we need to carry **more context** during typechecking
    - In particular, we need to add the enclosing class!

Type Rules for SELF_TYPE

- Since occurrences of **SELF_TYPE** depend on the **enclosing class**, we need to carry **more context** during typechecking
  - In particular, we need to add the enclosing class!
- This leads to a new typing judgment form:

$$\Gamma, \mathbf{M}, C \vdash e : T$$

Type Rules for SELF_TYPE

- Since occurrences of **SELF_TYPE** depend on the **enclosing class**, we need to carry **more context** during typechecking
  - In particular, we need to add the enclosing class!
- This leads to a new typing judgment form:

$$\Gamma, \mathbf{M}, C \vdash e : T$$

- Read as "An expression **e** occurring in the body of C has static type **T** given a variable type environment Γ and method signatures $\mathbf{M}$"

Changing the Type Rules for SELF_TYPE

- The next step is to design type rules that account for **SELF_TYPE** for each language construct

Changing the Type Rules for SELF_TYPE

- The next step is to design type rules that account for **SELF_TYPE** for each language construct
- Most of these rules are the same as the rules without **SELF_TYPE**, except that ≤ and ***lub*** are the new versions with **SELF_TYPE** support; only change is to pass through the enclosing class

Changing the Type Rules for SELF_TYPE

- The next step is to design type rules that account for **SELF_TYPE** for each language construct
- Most of these rules are the same as the rules without **SELF_TYPE**, except that ≤ and *lub* are the new versions with **SELF_TYPE** support; only change is to pass through the enclosing class
- E.g.,:

$$\frac{\Gamma, M, C \vdash e_1 : T_1 \quad \Gamma(id) = T_0 \quad T_1 \leq T_0}{\Gamma, M, C \vdash id \leftarrow e_1 : T_1} \text{[Assign]}$$

Changes to Dispatch Rules

- The rules for dispatch need to change. We modify the old dispatch rule:

Changes to Dispatch Rules

- The rules for dispatch need to change. We modify the old dispatch rule:

$$\frac{\begin{array}{ll} \Gamma, M, C \vdash e_0 : T_0 \quad \Gamma, & \\ M, C \vdash e_1 : T_1 & M(T_0, f) = (T_1', \ldots, T_n', T_{n+1}') \\ \ldots & \\ \Gamma, M, C \vdash e_n : T_n & \forall\ i \text{ in } (1 \ldots n),\ T_i \leq T_i' \end{array}}{\Gamma, M, C \vdash e_0.f(e_1, \ldots, e_n) : T_{n+1}'} \text{[Dispatch]}$$

Changes to Dispatch Rules

- The rules for dispatch need to change. We modify the old dispatch rule:

$$\frac{\begin{array}{ll} \Gamma, M, C \vdash e_0 : T_0 \quad \Gamma, & T_{n+1}' \neq \text{SELF\_TYPE} \\ M, C \vdash e_1 : T_1 & M(T_0, f) = (T_1', \ldots, T_n', T_{n+1}') \\ \ldots & \\ \Gamma, M, C \vdash e_n : T_n & \forall\ i \text{ in } (1 \ldots n),\ T_i \leq T_i' \end{array}}{\Gamma, M, C \vdash e_0.f(e_1, \ldots, e_n) : T_{n+1}'} \text{[Dispatch]}$$

Changes to Dispatch Rules

- Then, we add a **new rule** for the **SELF_TYPE** case:

Changes to Dispatch Rules

- Then, we add a new rule for the **SELF_TYPE** case:
  - (changes in pink)

$$\frac{\begin{array}{ll} \Gamma, M, C \vdash e_0 : T_0 \quad \Gamma, & \\ M, C \vdash e_1 : T_1 & M(T_0, f) = (T_1', \ldots, T_n', \text{SELF\_TYPE}) \\ \ldots & \\ \Gamma, M, C \vdash e_n : T_n & \forall\ i \text{ in } (1 \ldots n),\ T_i \leq T_i' \end{array}}{\Gamma, M, C \vdash e_0.f(e_1, \ldots, e_n) : T_0} \text{[Dispatch-Self]}$$

What's different about this rule?

$$\dfrac{\begin{array}{ll} \Gamma, M, C \vdash e_0 : T_0 & \\ \Gamma, M, C \vdash e_1 : T_1 & M(T_0, f) = (T_1', \ldots, T_n', \text{SELF\_TYPE}) \\ \ldots & \\ \Gamma, M, C \vdash e_n : T_n & \forall\ i \text{ in } (1 \ldots n),\ T_i \leq T_i' \end{array}}{\Gamma, M, C \vdash e_0.f(e_1, \ldots, e_n) : T_0}\ \text{[Dispatch-Self]}$$

What's different about this rule?

- It handles the **Stock** example

$$\frac{\begin{array}{ll} \Gamma, \mathbf{M}, \mathbf{C} \vdash e_0 : T_0 \quad \Gamma, & \\ \mathbf{M}, \mathbf{C} \vdash e_1 : T_1 & \mathbf{M}(T_0, f) = (T_1', \ldots, T_n', \text{SELF\_TYPE}) \\ \ldots & \\ \Gamma, \mathbf{M}, \mathbf{C} \vdash e_n : T_n & \forall\ i \text{ in } (1 \ldots n),\ T_i \leq T_i' \end{array}}{\Gamma, \mathbf{M}, \mathbf{C} \vdash e_0.f(e_1, \ldots, e_n) : T_0} \text{ [Dispatch-Self]}$$

What's different about this rule?

- It handles the **Stock** example
- Formal parameters **can't** be SELF_TYPE

$$\frac{\Gamma, M, C \vdash e_0 : T_0 \quad \Gamma, M, C \vdash e_1 : T_1 \quad \ldots \quad \Gamma, M, C \vdash e_n : T_n \quad M(T_0, f) = (T_1', \ldots, T_n', \mathrm{SELF\_TYPE}) \quad \forall\ i \text{ in } (1 \ldots n),\ T_i \le T_i'}{\Gamma, M, C \vdash e_0.f(e_1, \ldots, e_n) : T_0} \text{[Dispatch-Self]}$$

What's different about this rule?

- It handles the **Stock** example
- Formal parameters **can't** be SELF_TYPE
- Actual arguments **can** be SELF_TYPE
  - extended ≤ handles this case

$$\frac{\Gamma, M, C \vdash e_0 : T_0 \quad \Gamma, M, C \vdash e_1 : T_1 \quad \ldots \quad \Gamma, M, C \vdash e_n : T_n \quad M(T_0, f) = (T_1', \ldots, T_n', \text{SELF\_TYPE}) \quad \forall\, i \text{ in } (1 \ldots n),\ T_i \leq T_i'}{\Gamma, M, C \vdash e_0.f(e_1, \ldots, e_n) : T_0} \text{[Dispatch-Self]}$$

What's different about this rule?

- It handles the **Stock** example
- Formal parameters **can't** be SELF_TYPE
- Actual arguments **can** be SELF_TYPE
  - extended ≤ handles this case
- The type T_0 of the dispatch expression *could* be SELF_TYPE

$$\frac{\begin{array}{ll} \Gamma, M, C \vdash e_0 : T_0 \quad \Gamma, & \\ M, C \vdash e_1 : T_1 & M(T_0, f) = (T_1', \ldots, T_n', \text{SELF\_TYPE}) \\ \ldots & \\ \Gamma, M, C \vdash e_n : T_n & \forall\ i \text{ in } (1 \ldots n),\ T_i \leq T_i' \end{array}}{\Gamma, M, C \vdash e_0.f(e_1, \ldots, e_n) : T_0} \text{[Dispatch-Self]}$$

Changes to Dispatch Rules

- What about **static dispatch**? Does it need changes?

Changes to Dispatch Rules

- What about **static dispatch**? Does it need changes? Yes...

$$\frac{\begin{array}{ll} \Gamma, \mathbf{M}, \mathbf{C} \vdash e_0 : T_0 & \Gamma, T_0 \le T \\ \mathbf{M}, \mathbf{C} \vdash e_1 : T_1 & \mathbf{M}(T, f) = (T_1', \ldots, T_n', T_{n+1}') \\ \ldots & \\ \Gamma, \mathbf{M}, \mathbf{C} \vdash e_n : T_n & \forall\ i \text{ in } (1 \ldots n),\ T_i \le T_i' \end{array}}{\Gamma, \mathbf{M}, \mathbf{C} \vdash e_0@T.f(e_1, \ldots, e_n) : T_{n+1}'} \text{ [Static Dispatch]}$$

Changes to Dispatch Rules

- What about static dispatch? Does it need changes? Yes...

$$\frac{\begin{array}{ll} \Gamma, M, C \vdash e_0 : T_0 \quad \Gamma, T_0 \leq T & T_{n+1}' \neq \text{SELF\_TYPE} \\ M, C \vdash e_1 : T_1 & \\ & M(T, f) = (T_1', \ldots, T_n', T_{n+1}') \\ \ldots & \\ \Gamma, M, C \vdash e_n : T_n & \forall\ i \text{ in } (1 \ldots n), T_i \leq T_i' \end{array}}{\Gamma, M, C \vdash e_0@T.f(e_1, \ldots, e_n) : T_{n+1}'} \text{ [Static Dispatch]}$$

Changes to Dispatch Rules

- And again we need a special rule for when the method's return type is **SELF_TYPE**:

Changes to Dispatch Rules

- And again we need a special rule for when the method's return type is **SELF_TYPE**: (changes again in **pink**)

$$\frac{\begin{array}{ll} \Gamma, M, C \vdash e_0 : T_0 \quad \Gamma, M, C \vdash e_1 : T_1 & T_0 \leq T \\ \ldots & M(T, f) = (T_1', \ldots, T_n', \text{SELF\_TYPE}) \\ \Gamma, M, C \vdash e_n : T_n & \forall\, i \text{ in } (1 \ldots n),\ T_i \leq T_i' \end{array}}{\Gamma, M, C \vdash e_0@T.f(e_1, \ldots, e_n) : T_0} \quad \text{[St.-Dispatch-Self]}$$

Static Dispatch Notes

Static Dispatch Notes

- Why is the rule on the previous slide correct?
  - If we dispatch a method returning **SELF_TYPE** in some class T, don't we get back a T?

Static Dispatch Notes

- Why is the rule on the previous slide correct?
    - If we dispatch a method returning **SELF_TYPE** in some class **T**, don't we get back a **T**?
- **No**. SELF_TYPE is the type of "self", which may be a **subclass** of the class in which the **method body** appears

Static Dispatch Notes

- Why is the rule on the previous slide correct?
    - If we dispatch a method returning **SELF_TYPE** in some class **T**, don't we get back a **T**?
- **No**. SELF_TYPE is the type of "self", which may be a **subclass** of the class in which the **method body** appears
    - Note: ***not*** the class in which the **call site** appears!

Static Dispatch Notes

- Why is the rule on the previous slide correct?
  - If we dispatch a method returning **SELF_TYPE** in some class **T**, don't we get back a **T**?
- **No**. SELF_TYPE is the type of "self", which may be a **subclass** of the class in which the **method body** appears
  - Note: ***not*** the class in which the **call site** appears!
- The static dispatch class cannot be SELF_TYPE

New SELF_TYPE Rules

- There are also two other new rules specifically for SELF_TYPE:

New SELF_TYPE Rules

- There are also two other new rules specifically for SELF_TYPE:

$$\frac{}{\Gamma, M, C \vdash \textbf{self} : \text{SELF\_TYPE}_C} \textbf{[Self]}$$

New SELF_TYPE Rules

- There are also two other new rules specifically for SELF_TYPE:

$$\frac{}{\Gamma, M, C \vdash \mathbf{self} : \mathbf{SELF\_TYPE}_C} \text{ [Self]}$$

$$\frac{}{\Gamma, M, C \vdash \mathbf{new\ SELF\_TYPE} : \mathbf{SELF\_TYPE}_C} \text{ [New-Self]}$$

New SELF_TYPE Rules

- There are also two other new rules specifically for SELF_TYPE:

$$\frac{}{\Gamma, \mathrm{M}, \mathrm{C} \vdash \mathbf{self} : \mathbf{SELF\_TYPE_C}} \quad \textbf{[Self]}$$

$$\frac{}{\Gamma, \mathrm{M}, \mathrm{C} \vdash \mathbf{new\ SELF\_TYPE} : \mathbf{SELF\_TYPE_C}} \quad \textbf{[New-Self]}$$

- There are a number of other places in the rules where SELF_TYPE appears - read the CRM carefully

Where is SELF_TYPE illegal in Cool?

Where is SELF_TYPE illegal in Cool?

- `m(x : T) : T' { … }`
    - only `T'` (not `T`) can be SELF_TYPE!

Where is SELF_TYPE illegal in Cool?

- `m(x : T) : T' { … }`
  - only `T'` (not `T`) can be SELF_TYPE!
  - What would go wrong if `T` were SELF_TYPE?

Where is SELF_TYPE illegal in Cool?

- `m(x : T) : T' { … }`
  - only `T'` (not `T`) can be SELF_TYPE!
  - What would 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); ...
...
```

Summary of SELF_TYPE

Summary of SELF_TYPE

- The extended ≤ and ***lub*** operations can do a lot of the work.
    - Implement them to handle SELF_TYPE

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.***

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

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 typechecking of **dispatch**.

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 typechecking of **dispatch**.
  - SELF_TYPE as the return type in an invoked method might have ***nothing to do*** with the current class

Why Do We Cover SELF_TYPE?

Why Do We Cover SELF_TYPE?

- SELF_TYPE is an example of a **research idea**

Why Do We Cover SELF_TYPE?

- SELF_TYPE is an example of a **research idea**
    - it adds **expressiveness** to the type system without allowing any "bad" programs

Why Do We Cover SELF_TYPE?

- SELF_TYPE is an example of a **research idea**
  - it adds **expressiveness** to the type system without allowing any "bad" programs
  - but at the cost of **additional complexity**

Why Do We Cover SELF_TYPE?

- SELF_TYPE is an example of a **research idea**
  - it adds **expressiveness** to the type system without allowing any "bad" programs
  - but at the cost of **additional complexity**
- SELF_TYPE itself isn't that important
  - although you have to get it right for PA2…

Why Do We Cover SELF_TYPE?

- SELF_TYPE is an example of a **research idea**
  - it adds **expressiveness** to the type system without allowing any "bad" programs
  - but at the cost of **additional complexity**
- SELF_TYPE itself isn't that important
  - although you have to get it right for PA2…
- But it is **illustrative** of a class of ideas that trade-off expressiveness for complexity
  - and gives you a taste of how this works in practice!

Type Systems

- The rules in these lectures were **Cool-specific**

Type Systems

- The rules in these lectures were **Cool-specific**
  - Other languages have (very!) different rules
  - We'll survey some other type systems later in the course

Type Systems

- The rules in these lectures were **Cool-specific**
  - Other languages have (very!) different rules
  - We'll survey some other type systems later in the course
- **General themes** of type systems (that aren't Cool-specific):

Type Systems

- The rules in these lectures were **Cool-specific**
  - Other languages have (very!) different rules
  - We'll survey some other type systems later in the course
- **General themes** of type systems (that aren't Cool-specific):
  - Type rules are defined on the **structure** of expressions

Type Systems

- The rules in these lectures were **Cool-specific**
  - Other languages have (very!) different rules
  - We'll survey some other type systems later in the course
- **General themes** of type systems (that aren't Cool-specific):
  - Type rules are defined on the **structure** of expressions
  - Types of variables are modeled by a **type environment**

Type Systems

- The rules in these lectures were **Cool-specific**
  - Other languages have (very!) different rules
  - We'll survey some other type systems later in the course
- **General themes** of type systems (that aren't Cool-specific):
  - Type rules are defined on the **structure** of expressions
  - Types of variables are modeled by a **type environment**
  - There is a tradeoff between **safety** and **flexibility**

Type Systems

- The rules in these lectures were **Cool-specific**
  - Other languages have (very!) different rules
  - We'll survey some other type systems later in the course
- **General themes** of type systems (that aren't Cool-specific):
  - Type rules are defined on the **structure** of expressions
  - Types of variables are modeled by a **type environment**
  - There is a tradeoff between **safety** and **flexibility**
  - There is another tradeoff between **expressiveness** and **complexity**

In-class Activity

In-class Activity

- Get into groups of three.
- These typing judgments have one or more flaws. For each judgment, list the flaws and explain how they affect the judgment.

In-class Activity

- Get into groups of three.
- These typing judgments have one or more flaws. For each judgment, list the flaws and explain how they affect the judgment.

$$\frac{\begin{array}{c} O \vdash e_0 : T \\ O \vdash T \leq T_0 \\ O \vdash e_1 : T_1 \end{array}}{O[x/T_0] \vdash \textbf{let } x : T_0 \leftarrow e_0 \textbf{ in } e_1 : T_1} \; (let - init)$$

$$\frac{\begin{array}{c} O(\text{id}) = T_0 \\ O \vdash e_1 : T_1 \\ T_0 \leq T_1 \end{array}}{O \vdash \text{id} \leftarrow e_1 : T_1} \; (assign)$$

Trivia Break: Computer Science

This prolific Hungarian-American was a professor at Princeton, and lived in New Jersey from 1933 until his death. He made major contributions to multiple fields, including mathematics, physics, economics, and computer science. While he is the inventor of the merge sort algorithm, he is best known in computing for the architecture named after him (despite the fact that he did not directly invent it - J. Presper Eckert and John Mauchly did, while working on the ENIAC), which is the basis for the architecture of most modern digital computers.

Trivia Break: Holidays

This holiday, typically occurring sometime in February or March, marks the final day of new year celebrations in a widely-used lunar calendar. It is always celebrated during the full moon. As early as two millennia ago, it had become a festival of great significance. The day is traditionally marked by the consumption of tangyuan, a traditional dessert made of glutinous rice shaped into balls; and by the releasing of paper lanterns, which are typically red to symbolize good luck.

Course Status

- We have finished all the material that you *need* for PA2

Course Status

- We have finished all the material that you *need* for PA2
    - though next week we'll do a more in-depth discussion of other kinds of static analysis

Course Status

- We have finished all the material that you *need* for PA2
    - though next week we'll do a more in-depth discussion of other kinds of static analysis
- For the rest of this week, we'll focus on ***how*** code actually gets executed

Course Status

- We have finished all the material that you *need* for PA2
  - though next week we'll do a more in-depth discussion of other kinds of static analysis
- For the rest of this week, we'll focus on ***how*** code actually gets executed
  - today: basics of run-time organization

Course Status

- We have finished all the material that you *need* for PA2
  - though next week we'll do a more in-depth discussion of other kinds of static analysis
- For the rest of this week, we'll focus on ***how*** code actually gets executed
  - today: basics of run-time organization
  - Wednesday: formal description of how a program actually runs (***operational semantics***)

Course Status

- We have finished all the material that you *need* for PA2
  - though next week we'll do a more in-depth discussion of other kinds of static analysis
- For the rest of this week, we'll focus on ***how*** code actually gets executed
  - today: basics of run-time organization
  - Wednesday: formal description of how a program actually runs (***operational semantics***)
- Goal of all of this: make sure you have the **foundation** for PA3
  - (also, operational semantics + type rules are closely related)

Run-time Environments

- Before discussing code execution, we need to understand **what we are trying to execute**

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

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

Run-time Organization Outline

- Management of run-time resources

Run-time Organization Outline

- Management of run-time resources
- Correspondence between **static** and **dynamic** structures
  - remind me: what do “static” and “dynamic” mean?

Run-time Organization Outline

- Management of run-time resources
- Correspondence between **static** and **dynamic** structures
  - remind me: what do “static” and “dynamic” mean?
- Storage organization

Run-time Organization Outline

- **Management of run-time resources**
- Correspondence between **static** and **dynamic** structures
  - remind me: what do “static” and “dynamic” mean?
- Storage organization

Run-time Resources

- Execution of a program is initially under the control of the operating system

Run-time Resources

- Execution of a program is initially under the control of the operating system
- When a program is invoked:

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

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 of the program is loaded into some part of that space

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 of the program is loaded into some part of that space
  - The OS jumps to the entrypoint (i.e., “main”)

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 of the program is loaded into some part of that space
  - The OS jumps to the entrypoint (i.e., “main”)
- How does “space” work?

Space: Virtual Memory (OS/Arch review?)

- An ***address space*** is a partial mapping from addresses to values. Like a big array: the value at memory address 0x12340000 might be 87. *Partial* means some addresses may be invalid.

Space: Virtual Memory (OS/Arch review?)

- An ***address space*** is a partial mapping from addresses to values. Like a big array: the value at memory address 0x12340000 might be 87. *Partial* means some addresses may be invalid.
- There is an address space associated with the ***physical memory*** in your computer. If you have 1GB of RAM, addresses 0 to 0x40000000 are valid.

Space: Virtual Memory (OS/Arch review?)

- An ***address space*** is a partial mapping from addresses to values. Like a big array: the value at memory address 0x12340000 might be 87. *Partial* means some addresses may be invalid.
- There is an address space associated with the ***physical memory*** in your computer. If you have 1GB of RAM, addresses 0 to 0x40000000 are valid.
- If I want to store some information on MachineX and you want to store other information on MachineX, we would have to collude to use different physical addresses (= different parts of the address space).

Space: Virtual Memory 2 (OS/Arch review?)

- ***Virtual memory*** is an abstraction in which each process gets its own ***virtual address space***. The OS and hardware work together to provide this abstraction. All modern general computers use it.

Space: Virtual Memory 2 (OS/Arch review?)

- ***Virtual memory*** is an abstraction in which each process gets its own ***virtual address space***. The OS and hardware work together to provide this abstraction. All modern general computers use it.
- Each virtual address space is then mapped separately into a different part of physical memory. (simplification)

Space: Virtual Memory 2 (OS/Arch review?)

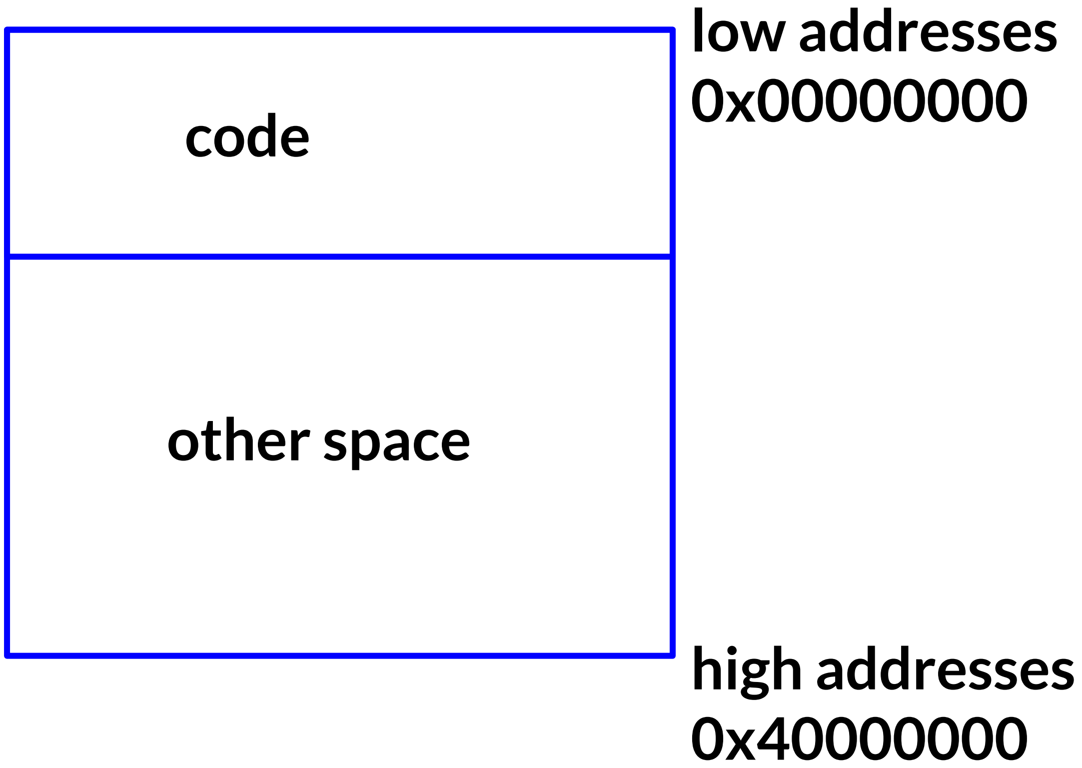
- ***Virtual memory*** is an abstraction in which each process gets its own ***virtual address space***. The OS and hardware work together to provide this abstraction. All modern general computers use it.
- Each virtual address space is then mapped separately into a different part of physical memory. (simplification)
- So **Process1** can store information at its virtual address **0x4444** and **Process2** can also store information at its virtual address **0x4444** and there will be ***no overlap*** in physical memory.

Space: Virtual Memory 2 (OS/Arch review?)

- ***Virtual memory*** is an abstraction in which each process gets its own ***virtual address space***. The OS and hardware work together to provide this abstraction. All modern general computers use it.
- Each virtual address space is then mapped separately into a different part of physical memory. (simplification)
- So **Process1** can store information at its virtual address **0x4444** and **Process2** can also store information at its virtual address **0x4444** and there will be ***no overlap*** in physical memory.
  - e.g., **P1 0x4444** virtual -> 0x1000 physical
  - and **P2 0x4444** virtual -> 0x8000 physical

Program Memory Layout

a program's virtual memory:



Notes on How I’ve Presented This

- Our pictures of machine organization have:
  - Low address at the top
  - High address at the bottom
  - Lines delimiting areas for different kinds of data

Notes on How I've Presented This

- 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

Notes on How I’ve Presented This

- 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 (doesn't matter)

“Other Space”

- “Other Space” in the picture holds all of the **data** for the program
    - i.e., “Other Space” = “Data Space”

“Other Space”

- “Other Space” in the picture holds all of the **data** for the program
  - i.e., “Other Space” = “Data Space”
- A **compiler** is responsible for:
  - generating code (that will be run later)
  - orchestrating use of this data space

“Other Space”

- “Other Space” in the picture holds all of the **data** for the program
  - i.e., “Other Space” = “Data Space”
- A **compiler** is responsible for:
  - generating code (that will be run later)
  - orchestrating use of this data space
- An **interpreter** only has to:
  - directly execute the code
  - manage the program’s run-time data itself

“Other Space”

- “Other Space” in the picture holds all of the **data** for the program
  - i.e., “Other Space” = “Data Space”
- A **compiler** is responsible for:
  - generating code (that will be run later)
  - orchestrating use of this data space
- An **interpreter** only has to:
  - directly execute the code
  - manage the program’s run-time data itself
- Of these two, the compiler’s task is **much harder**: the compiler must **predict** the program’s behavior to do it right!

Code Execution Goals

- We have two goals when generating code to execute:

Code Execution Goals

- We have two goals when generating code to execute:
  - **Correctness**
  - **Speed**

Code Execution Goals

- We have two goals when generating code to execute:
  - **Correctness**
  - **Speed**
- Which of these **matters more**?

Code Execution Goals

- We have two goals when generating code to execute:
  - Correctness
  - Speed
- Which of these **matters more**?
  - Correctness! First rule of compilers…

Code Execution Goals

- We have two goals when generating code to execute:
  - **Correctness**
  - **Speed**
- Which of these **matters more**?
  - **Correctness**! First rule of compilers…
- Most complications in run-time organization, though, come from trying to be both fast *and* correct

Assumptions About Execution

Assumptions About Execution

- **Assumption (1):** Execution is **sequential**; control moves from one point in a program to another in a well-defined order

Assumptions About Execution

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

Assumptions About Execution

- **Assumption (1):** Execution is **sequential**; control moves from one point in a program to another in a well-defined order
- **Assumption (2):** When a procedure is called, control eventually returns to the point **immediately after the call**
- Do these assumptions always hold?

Assumptions About Execution

- **Assumption (1):** Execution is **sequential**; control moves from one point in a program to another in a well-defined order
- **Assumption (2):** When a procedure is called, control eventually returns to the point **immediately after the call**
- Do these assumptions always hold?
    - **Of course not**! But, they're useful simplifications and hold enough of the time that we can use them.

Assumptions About Execution

- **Assumption (1):** Execution is **sequential**; control moves from one point in a program to another in a well-defined order
- **Assumption (2):** When a procedure is called, control eventually returns to the point **immediately after the call**
- Do these assumptions always hold?
  - **Of course not**! But, they're useful simplifications and hold enough of the time that we can use them.
  - Examples violating (1):

Assumptions About Execution

- **Assumption (1):** Execution is **sequential**; control moves from one point in a program to another in a well-defined order
- **Assumption (2):** When a procedure is called, control eventually returns to the point **immediately after the call**
- Do these assumptions always hold?
  - **Of course not**! But, they're useful simplifications and hold enough of the time that we can use them.
  - Examples violating (1): scheduler, having more than one CPU

Assumptions About Execution

- **Assumption (1):** Execution is **sequential**; control moves from one point in a program to another in a well-defined order
- **Assumption (2):** When a procedure is called, control eventually returns to the point **immediately after the call**
- Do these assumptions always hold?
  - **Of course not**! But, they're useful simplifications and hold enough of the time that we can use them.
  - Examples violating (1): scheduler, having more than one CPU
  - Examples violation (2):

Assumptions About Execution

- **Assumption (1):** Execution is **sequential**; control moves from one point in a program to another in a well-defined order
- **Assumption (2):** When a procedure is called, control eventually returns to the point **immediately after the call**
- Do these assumptions always hold?
  - **Of course not**! But, they're useful simplifications and hold enough of the time that we can use them.
  - Examples violating (1): scheduler, having more than one CPU
  - Examples violation (2): exceptions, kill signals

Activations

Definition: Each invocation of some procedure P is an ***activation*** of P

Activations

Definition: Each invocation of some procedure P is an ***activation*** of P

- The ***lifetime*** of an activation of P is all the steps to activate P
  - including all the steps of procedures that P calls, and that those procedures call, etc.

Activations

Definition: Each invocation of some procedure P is an ***activation*** of P

- The ***lifetime*** of an activation of P is all the steps to activate P
    - including all the steps of procedures that P calls, and that those procedures call, etc.
- We also will discuss **lifetimes of variables**.
    - The lifetime of a variable x is the portion of execution during which x is defined.

Activations

Definition: Each invocation of some procedure P is an ***activation*** of P

- The ***lifetime*** of an activation of P is all the steps to activate P
  - including all the steps of procedures that P calls, and that those procedures call, etc.
- We also will discuss **lifetimes of variables**.
  - The lifetime of a variable x is the portion of execution during which x is defined.
- Note the relation with **scope**: scope is static, lifetimes are dynamic

Activation Trees

- Assumption (2) requires that when P calls Q, then Q returns before P does

Activation Trees

- Assumption (2) requires that when P calls Q, then Q returns before P does
  - That is, that the lifetimes of procedure activations are **properly nested**

Activation Trees

- Assumption (2) requires that when P calls Q, then Q returns before P does
    - That is, that the lifetimes of procedure activations are **properly nested**
- As a result, we can depict activation lifetimes as a **tree**

Activation Trees

- Assumption (2) requires that when P calls Q, then Q returns before P does
  - That is, that the lifetimes of procedure activations are **properly nested**
- As a result, we can depict activation lifetimes as a **tree**
- Example ->

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

Activation Trees

- Assumption (2) requires that when P calls Q, then Q returns before P does
  - That is, that the lifetimes of procedure activations are **properly nested**
- As a result, we can depict activation lifetimes as a **tree**
- Example ->

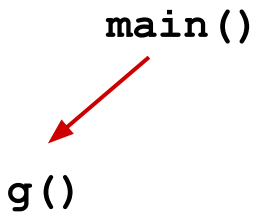
```
class Main {
  g() : Int { 1 };
  f() : Int { g() };
  main() : Int {{ g(); f(); }};
};
```

```
main()
```

Activation Trees

- Assumption (2) requires that when P calls Q, then Q returns before P does
  - That is, that the lifetimes of procedure activations are **properly nested**
- As a result, we can depict activation lifetimes as a **tree**
- Example ->

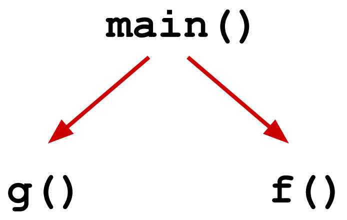
```
class Main {
  g() : Int { 1 };
  f() : Int { g() };
  main() : Int {{ g(); f(); }};
};
```



Activation Trees

- Assumption (2) requires that when P calls Q, then Q returns before P does
  - That is, that the lifetimes of procedure activations are **properly nested**
- As a result, we can depict activation lifetimes as a **tree**
- Example ->

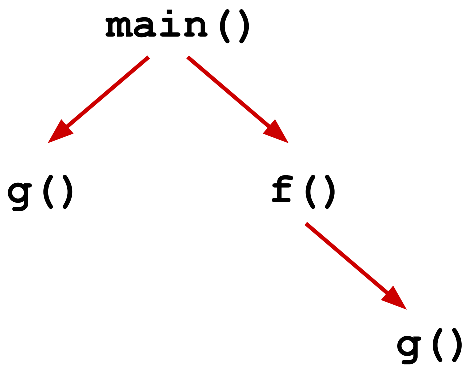
```
class Main {
  g() : Int { 1 };
  f() : Int { g() };
  main() : Int {{ g(); f(); }};
};
```



Activation Trees

- Assumption (2) requires that when P calls Q, then Q returns before P does
  - That is, that the lifetimes of procedure activations are **properly nested**
- As a result, we can depict activation lifetimes as a **tree**
- Example ->

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



Activation Trees: Another Example

- What's the activation tree for this example?

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

(on the whiteboard)

Activation Tree Notes

Activation Tree Notes

- The activation tree **depends on run-time behavior**

Activation Tree Notes

- The activation tree **depends on run-time behavior**
  - The activation tree may be different for every program input

Activation Tree 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

Activation Tree 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***

Activation Tree Example Revisited

- Let’s track activations with a stack on the example from before:

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

Stack

Activation Tree Example Revisited

- Let’s track activations with a stack on the example from before:

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

Stack

main()

Activation Tree Example Revisited

- Let’s track activations with a stack on the example from before:

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

Stack

`main()`

`g()`

Activation Tree Example Revisited

- Let’s track activations with a stack on the example from before:

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

Stack

`main()`

`f()`

Activation Tree Example Revisited

- Let's track activations with a stack on the example from before:

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

<u>Stack</u>

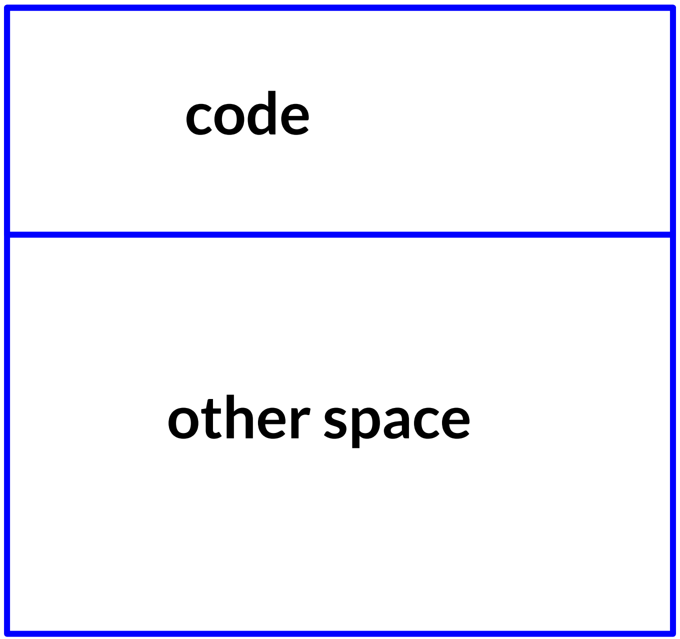
`main()`

`f()`

`g()`

Revised Memory Layout

a program's virtual memory:



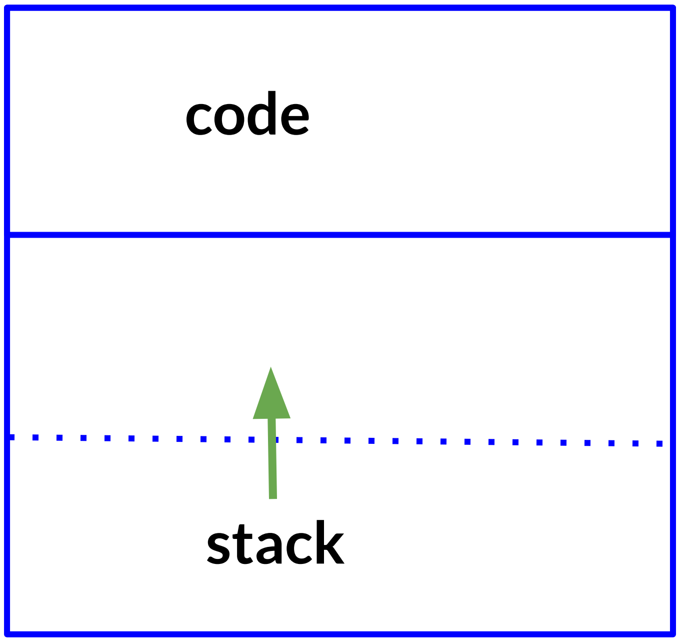
low addresses
0x00000000

high addresses
0x40000000

Revised Memory Layout

a program's
virtual
memory:

low addresses
0x00000000



high addresses
0x40000000

Activation Records

- On many machines the stack starts at higher addresses and grows towards lower addresses

Activation Records

- On many machines the stack starts at higher addresses and grows towards lower addresses
- The information needed to manage one procedure activation is called an ***activation record*** (***AR***) or ***frame***

Activation Records

- On many machines the stack starts at higher 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**.

Activation Records

- On many machines the stack starts at higher 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**.
    - **F** is "***suspended***" until **G** completes, at which point **F** resumes.

Activation Records

- On many machines the stack
  addresses and grows toward
- The information needed to m
  procedure activation is calle
  record (*AR*) or *frame*
- If procedure **F** calls **G**, then G
  contains a mix of info about
  - **F** is "*suspended*" until **G** c
    point **F** resumes.

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

- 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

Activation Records

- On many machines the stack
addresses and grows toward
- The information needed to r
procedure activation is calle
record (***AR***) or ***frame***
- If procedure **F** calls **G**, then **G**
contains a mix of info about
    - **F** is "***suspended***" until **G** c
point **F** resumes.

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

- 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

Activation Records

- On many machines the stack
addresses and grows toward
- The information needed to r
procedure activation is calle
record (***AR***) or ***frame***
- If procedure **F** calls **G**, then **G**
contains a mix of info about
  - **F** is "***suspended***" until **G** c
point **F** resumes.

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

- G's AR contains information needed to resume execution of F.
- G's AR may also contain:
  - Actual parameters to G (supplied by F)

Activation Records

- On many machines the stack
  addresses and grows toward
- The information needed to r
  procedure activation is calle
  record (*AR*) or *frame*
- If procedure **F** calls **G**, then
  contains a mix of info about
  - **F** is "*suspended*" until **G** c
    point **F** resumes.

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

- 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

Activation Records

- On many machines the stack
  addresses and grows toward
- The information needed to r
  procedure activation is calle
  record (*AR*) or *frame*
- If procedure **F** calls **G**, then G
  contains a mix of info about
  - **F** is "*suspended*" until **G** c
    point **F** resumes.

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

- 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

Contents of a Typical AR (for some procedure **G**)

Contents of a Typical AR (for some procedure **G**)

- Space for **G**'s return value

Contents of a Typical AR (for some procedure **G**)

- Space for **G**’s return value
- Actual parameters

Contents of a Typical AR (for some procedure **G**)

- Space for **G**'s return value
- Actual parameters
- Pointer to the previous activation record
    - This ***control link*** points back to the AR of **F** (caller of **G**)

Contents of a Typical AR (for some procedure **G**)

- Space for **G**’s return value
- Actual parameters
- Pointer to the previous activation record
    - This ***control link*** points back to the AR of **F** (caller of **G**)
        - sometimes also called the ***frame pointer***

Contents of a Typical AR (for some procedure **G**)

- Space for **G**'s return value
- Actual parameters
- Pointer to the previous activation record
  - This ***control link*** points back to the AR of **F** (caller of **G**)
    - sometimes also called the ***frame pointer***
- Machine status prior to calling **G**
  - Local variables
  - Register and program counter contents

Contents of a Typical AR (for some procedure **G**)

- Space for **G**'s return value
- Actual parameters
- Pointer to the previous activation record
  - This ***control link*** points back to the AR of **F** (caller of **G**)
    - sometimes also called the ***frame pointer***
- Machine status prior to calling **G**
  - Local variables
  - Register and program counter contents
- Other temporary values

Revisiting An Example

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

Revisiting An Example

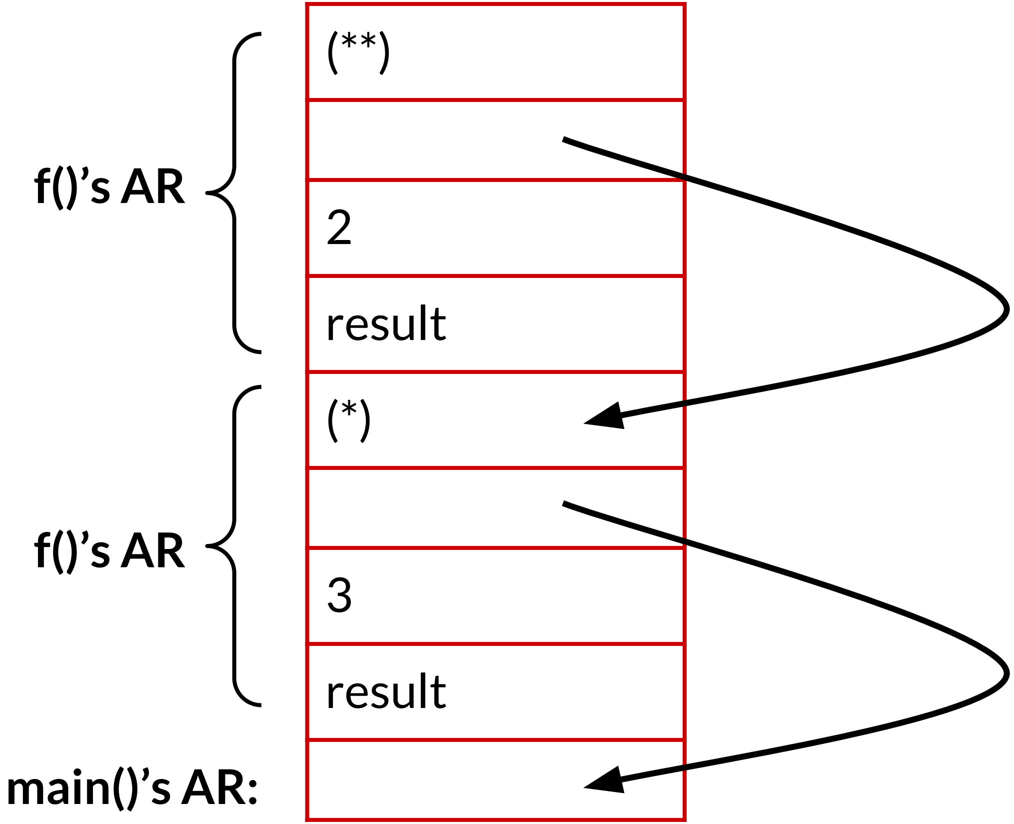
```
class Main {
   g() : Int { 1 };
   f(x : Int) : Int {
     if x = 0
       then g()
       else f(x - 1) (**)
     fi
   };
   main() : Int {{
     f(3); (*) }};
};
```

AR for **f**:

return address
control link
argument
space for result

Revisiting An Example: Stack after 2 Calls to f()

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



Notes on The Example

- **main()** has no argument or local variables and its result is “never” used; its AR is uninteresting

Notes on The Example

- **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

Notes on The Example

- **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.

The Main Point

The Main Point

The compiler must determine, at compile-time, the layout of activation records and generate code that, when executed at run-time, correctly accesses locations in those activation records.

The Main Point

The compiler must determine, at compile-time, the layout of activation records and generate code that, when executed at run-time, correctly accesses locations in those activation records.

*Thus, the AR layout and the compiler must be **designed together**!*

Discussion

Discussion

- The advantage of placing the return value first 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

Discussion

- The advantage of placing the return value first 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!

Discussion

- The advantage of placing the return value first 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

Discussion

- The advantage of placing the return value first 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

Discussion

- The advantage of placing the return value first 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**

Discussion

- The advantage of placing the return value first 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**
        - This is an important tradeoff! On an embedded device with fixed software, you might make different choices!

Discussion

- The advantage of placing t
the caller can find it at a fi
  - The caller must write t
- There is **nothing magic** ab
  - 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**
    - This is an important tradeoff! On an embedded device with fixed software, you might make different choices!

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

Globals

Globals

- All references to a global variable must point to the same object
  - Can’t really store a global in an activation record

Globals

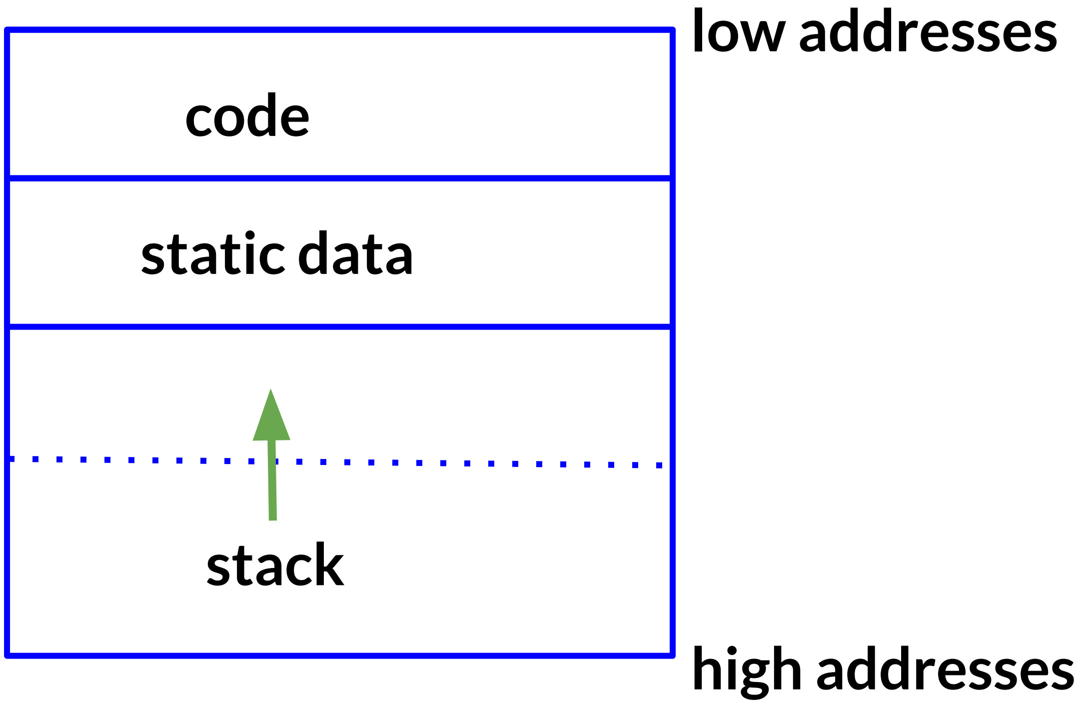
- All references to a global variable must point to the same object
  - Can't really store a global in an activation record
- Globals are assigned a fixed address once
  - Variables with fixed address are "***statically allocated***"

Globals

- All references to a global variable must point to the same object
  - Can't really store a global in an activation record
- 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

Memory Layout with Static Data

a program's
virtual
memory:



Heap Storage

- A value that outlives the procedure that creates it cannot be kept in the AR, even if it's not a global

Heap Storage

- A value that outlives the procedure that creates it cannot be kept in the AR, even if it's not a global
  - e.g., `foo : Bar () { new Bar };`

Heap Storage

- A value that outlives the procedure that creates it cannot be kept in the AR, even if it's not a global
  - e.g., `foo : Bar () { new Bar };`
    - this `Bar` value must survive deallocation of `foo`'s AR

Heap Storage

- A value that outlives the procedure that creates it cannot be kept in the AR, even if it's not a global
  - e.g., `foo : Bar () { new Bar };`
    - this `Bar` value must survive deallocation of `foo`'s AR
- Languages with dynamically-allocated data (such as Cool!) use a ***heap*** to store such dynamic data

Summary

Summary

- The **code area** contains object code
    - For most languages, fixed size and read only

Summary

- 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

Summary

- 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

Summary

- 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
- The **heap** contains all other data
  - In C, heap is managed by *malloc* and *free*

Summary

- The **code area** contains ob
    - For most languages, fi
- The **static area** contains d
  (e.g., global data)
    - Fixed size, may be read
- The **stack** contains an AR for each currently active procedure
    - Each AR usually fixed size, contains locals
- The **heap** contains all other data
    - In C, heap is managed by *malloc* and *free*

- Both the stack and the heap grow

Summary

- The code area contains ob
  - For most languages, fi
- The static area contains d (e.g., global data)
  - Fixed size, may be read
- The stack contains an AR for each currently active procedure
  - Each AR usually fixed size, contains locals
- The heap contains all other data
  - In C, heap is managed by *malloc* and *free*

- Both the stack and the heap grow
- Compilers must take care that they don't grow into each other!

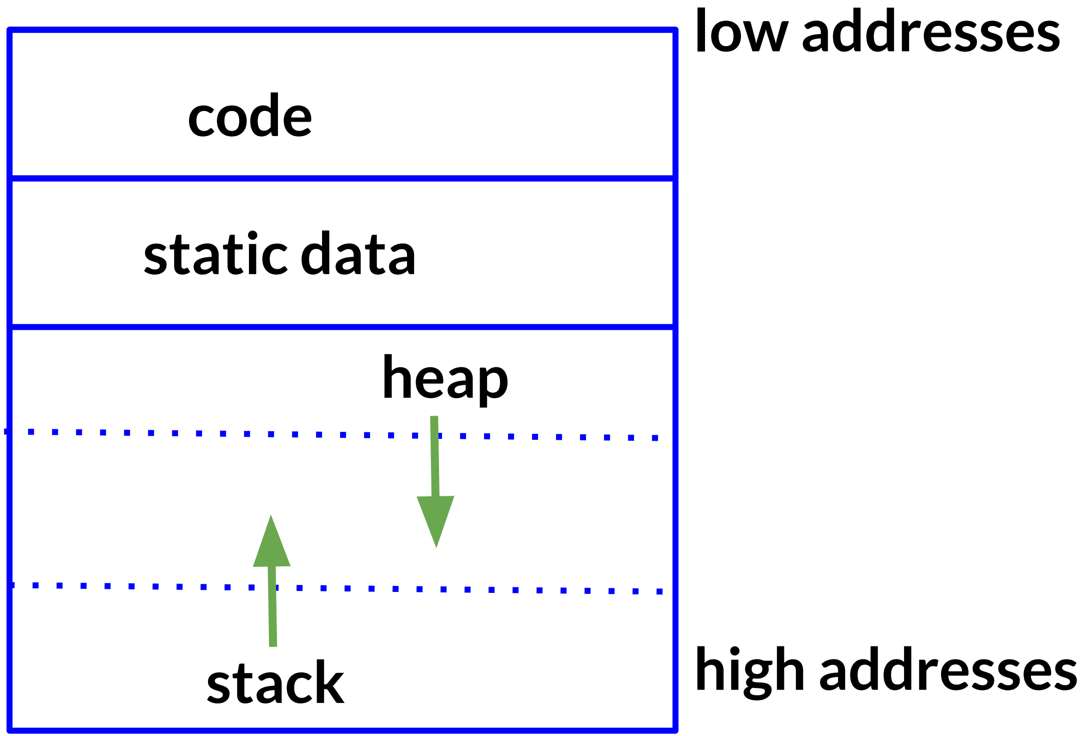
Summary

- The code area contains ob
  - For most languages, fi
- The static area contains d
  (e.g., global data)
  - Fixed size, may be reac
- The stack contains an AR for each currently active procedure
  - Each AR usually fixed size, contains locals
- The heap contains all other data
  - In C, heap is managed by *malloc* and *free*

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

Memory Layout with Heap

a program's
virtual
memory:



Your Own Heap

- In PA3, you'll need to emit assembly code for things like:

```
let x = new Counter(5) in
let y = x in {
  x.increment(1);
  out_int( y.getCount() ); // what does this print?
}
```

Your Own Heap

- In PA3, you'll need to emit assembly code for things like:

```
let x = new Counter(5) in
let y = x in {
  x.increment(1);
  out_int( y.getCount() ); // what does this print?
}
```

- You'll need to use and manage an **explicit heap** (introduced today and covered in more detail in later lectures). A heap maps addresses (i.e., integers) to values.

Course Announcements

- **PA2c2** due next Monday
  - requires typechecking + semantic analysis of everything but expressions
  - if you haven't started yet, I'm worried for you
  - don't forget that you can work in pairs!
    - I strongly recommend this option
    - it's not too late to pair up, even if both of you have started independently