Compiling object-oriented languages
Objects

An object consists of **Data** (attributes) and **Behaviour** (methods).

```java
class AstNode {
    location loc;
    public AstNode(location nodeloc)
    { loc = nodeloc; }
    public location getLocation()
    { return loc; }
}
abstract class Expr extends AstNode {
    public abstract int eval(Env);
    public Expr(location loc) { super(loc); }
}
public class AddExpr extends Expr {
    Expr left, right;
    public AddExpr(int loc, Expr x, Expr y)
    { super(loc); left = x; right = y; }
    public int eval(Env env)
    { return left.eval(env) + right.eval(env); }
}
```
Compiling objects

• Compiling OO languages with single inheritance:
  • Each class is associated with a dispatch vector (aka virtual table, vtable)
    • dispatch vector = record of function pointers – one for each method
  • Each object is associated with a record, with one field for the dispatch vector of its class, and
    one field for each attribute
Compiling methods

Each method is extended with an additional parameter for the current object

- Gives the method access to the attributes of the object
- Dispatch vector enables dynamic dispatch

```c
location AstNode_getLocation(self) {
    return self.loc;
}

int AddNode_eval(self, env) {
    return self.left.dispatch.eval(self.left, env)
        + self.right.dispatch.eval(self.right, env);
}

int IntNode_eval(self, env) {
    return self.value;
}
```
Subtyping

• Recall the *Liskov substitution principle*: if $s$ is a subtype of $t$, then terms of type $t$ can be replaced with terms of type $s$ without breaking type safety.
Subtyping

• Recall the Liskov substitution principle: if $s$ is a subtype of $t$, then terms of type $t$ can be replaced with terms of type $s$ without breaking type safety.

• If class $B$ extends class $A$, then $B$ is a subtype of $A$
Subtyping

• Recall the *Liskov substitution principle*: if $s$ is a subtype of $t$, then terms of type $t$ can be replaced with terms of type $s$ without breaking type safety.

• If class $B$ extends class $A$, then $B$ is a subtype of $A$

• This works for the same reason that record width subtyping works:
  • If $A$ has a method $foo$, it appears in the same position in $A$ and $B$'s dispatch vector
  • If $A$ has an attribute $x$, then $A$ objects and $B$ objects place $x$ in the same position in object records

\[
\text{RecordWidth} \\
\vdash \{lab_1 : s_1; ...; lab_m : s_m\} <: \{lab_1 : s_1; ...; lab_n : s_n\}^{n < m}
\]
Testing class membership

- Some OO languages support testing whether an object belongs to a given class, and performing (checked) downcasts.
Testing class membership

- Some OO languages support testing whether an object belongs to a given class, and performing (checked) downcasts
- To implement, we need a run-time representation class of the class hierarchy
Testing class membership

• Some OO languages support testing whether an object belongs to a given class, and performing (checked) downcasts
• To implement, we need a run-time representation class of the class hierarchy
• One solution:
  • The dispatch table serves as a type tag
    (i.e., $\text{typeOf}(o) = \text{AddExpr} \iff o\text{.dispatch} = \text{DispatchVector(AddExpr)})$
Testing class membership

- Some OO languages support testing whether an object belongs to a given class, and performing (checked) downcasts
- To implement, we need a run-time representation class of the class hierarchy
- One solution:
  - The dispatch table serves as a type tag
    (i.e., `typeof(o) == AddExpr ⇐⇒ o.dispatch = DispatchVector(AddExpr)`)  
  - The first member of each dispatch table is a pointer to parent type
    
    Checked downcasting: if `o instanceof c` then bitcast, otherwise throw run-time exception.
Testing class membership

- Some OO languages support testing whether an object belongs to a given class, and performing (checked) downcasts
- To implement, we need a run-time representation class of the class hierarchy
- One solution:
  - The dispatch table serves as a type tag
    (i.e., `typeof(o) == AddExpr ⇐⇒ o.dispatch = DispatchVector(AddExpr)`)
  - The first member of each dispatch table is a pointer to parent type
  - To check `o instanceof C`, walk up the class hierarchy
    - `o.dispatch = DispatchVector(C), or`
    - `o.dispatch != DispatchVector(Object) and o.dispatch.parent = DispatchVector(C), or`
    - `o.dispatch != DispatchVector(Object) and o.dispatch.parent != DispatchVector(Object) and o.dispatch.parent.parent = DispatchVector(C), or`
    - ...
  - Checked downcasting: if `o instanceof c` then bitcast, otherwise throw run-time exception.
Testing class membership

- Some OO languages support testing whether an object belongs to a given class, and performing (checked) downcasts
- To implement, we need a run-time representation class of the class hierarchy
- One solution:
  - The dispatch table serves as a type tag
    (i.e., \( \text{typeof}(o) = \text{AddExpr} \iff o\text{.dispatch} = \text{DispatchVector(AddExpr)} \))
  - The first member of each dispatch table is a pointer to parent type
  - To check \( o \text{ instanceof } C \), walk up the class hierarchy
    - \( o\text{.dispatch} = \text{DispatchVector}(C) \), or
    - \( o\text{.dispatch} \neq \text{DispatchVector(} \text{Object}) \text{ and } o\text{.dispatch}\text{.parent} = \text{DispatchVector}(C) \), or
    - \( o\text{.dispatch} \neq \text{DispatchVector(} \text{Object}) \text{ and } o\text{.dispatch}\text{.parent} \neq \text{DispatchVector(} \text{Object}) \text{ and } o\text{.dispatch}\text{.parent}\text{.parent} = \text{DispatchVector}(C) \), or
    - ...
  - Checked downcasting: if \( o \text{ instanceof } C \) then bitcast, otherwise throw run-time exception.
Multiple inheritance

- Some languages (such as C++) support a class extending more than one base class.
Multiple inheritance

- Some languages (such as C++) support a class extending more than one base class.
- Previous strategy does not work: base classes have conflicting ideas about where methods are stored in vtable.
- Solution: Use hash tables instead of records.
- Cost can be reduced with optimizing compiler.
- Perform a conservative analysis to determine the class of (some) objects. If known statically, can replace dynamic dispatch with static dispatch.
- JIT compilation.
  - At compile time, we have more precise information about object classes.
  - Replace dynamic dispatch with static dispatch, optimize & compile the result.
Multiple inheritance

• Some languages (such as C++) support a class extending more than one base class
• Previous strategy does not work: base classes have conflicting ideas about where methods are stored in vtable
• Solution: Use hash tables instead of records
Multiple inheritance

- Some languages (such as C++) support a class extending more than one base class
- Previous strategy does not work: base classes have conflicting ideas about where methods are stored in vtable
- Solution: Use hash tables instead of records
- Cost can be reduced with optimizing compiler
  - Perform a conservative analysis to determine the class of (some) objects. If known statically, can replace dynamic dispatch with static dispatch
  - JIT compilation
    - At compile time, we have more precise information about object classes
    - Replace dynamic dispatch with static dispatch, optimize & compile the result.
Garbage Collection
Garbage collection

- Many modern languages feature *garbage collectors*, which automatically reclaim memory that was allocated by a program but no longer used.
Garbage collection

- Many modern languages feature *garbage collectors*, which automatically reclaim memory that was allocated by a program but no longer used.
- A memory location is *garbage* if it will not be used in the remainder of the program.

*Static analyses* collect information about a program without running it, while *dynamic analyses* collect information about a program while running it.
Garbage collection

• Many modern languages feature garbage collectors, which automatically reclaim memory that was allocated by a program but no longer used

• A memory location is garbage if it will not be used in the remainder of the program

• Determining whether it will not be used is undecidable
  • But, we are happy with a conservative approximation: free memory if it cannot possibly be used in the remainder of the program

• Usually not a static analysis, but rather a dynamic analysis
  • Static analyses collect information about a program without running it
  • Dynamic analyses collect information about a program while running it
Many modern languages feature garbage collectors, which automatically reclaim memory that was allocated by a program but no longer used.

A memory location is garbage if it will not be used in the remainder of the program.

Determining whether it will not be used is undecidable.

But, we are happy with a conservative approximation: free memory if it cannot possibly be used in the remainder of the program.

Usually not a static analysis, but rather a dynamic analysis:

- static analyses collect information about a program without running it.
- dynamic analyses collect information about a program while running it.
Reference counting

• Each memory location gets an extra int field to hold the number of active references to that memory
• Collect when count is zero
• Example: compiling a store $x\rightarrow f = y$

```
x->f = y
y->count ++
tmp = x->f
tmp->count --
if (tmp->count == 0) free(tmp);
x->f = y
```
Reference counting

- Each memory location gets an extra int field to hold the number of active references to that memory
- Collect when count is zero
- Example: compiling a store $x->f = y$

$$y->\text{count }++$$
Reference counting

- Each memory location gets an extra int field to hold the number of active references to that memory
- Collect when count is zero
- Example: compiling a store $x->f = y$

$y->count ++$
$\text{tmp} = x->f$

```
y->count ++
```
Reference counting

- Each memory location gets an extra int field to hold the number of active references to that memory
- Collect when count is zero
- Example: compiling a store $x\rightarrow f = y$
  
  ```
  y->count ++
  tmp = x->f
  tmp->count --
  if (tmp->count == 0) free(tmp);
  ```
Reference counting

• Each memory location gets an extra int field to hold the number of active references to that memory
• Collect when count is zero
• Example: compiling a store \(x \rightarrow f = y\)

\[
\begin{align*}
y &\rightarrow \text{count ++} \\
tmp &\rightarrow x \rightarrow f \\
tmp &\rightarrow \text{count --} \\
\text{if (tmp->count == 0) free(tmp);} \\
x &\rightarrow f = y
\end{align*}
\]
Problem: cyclic data structures never get collected
Problem: *cyclic* data structures never get collected
Problem: cyclic data structures never get collected
Problem: *cyclic* data structures never get collected

```
dll
  \[\text{ref: 2}\]
  \[\text{forward}\]
  \[\text{back}\]
```

```
\[\text{ref: 2}\]
\[\text{forward}\]
\[\text{back}\]
```

```
\[\text{ref: 1}\]
\[\text{forward}\]
\[\text{back}\]
```
Problem: *cyclic* data structures never get collected

```
ref: 1
forward
back
```
```
ref: 2
forward
back
```
```
ref: 1
forward
back
```
Tracing-based GC

- **Tracing garbage collection**: a memory location is garbage if it is unreachable from the program's *roots*
  - *roots* = registers, stack, global static data
Tracing-based GC

• **Tracing garbage collection**: a memory location is garbage if it is unreachable from the program's *roots*
  - *roots* = registers, stack, global static data

• **Mark-and-sweep**
  - Each memory location gets an extra bit to hold a “mark”
  - *Mark*: When there is no remaining free memory, run a DFS search from the roots, marking all memory locations
  - *Sweep*: Traverse the entire heap; unmarked nodes are collected; marked nodes are unmarked
Memory layout

- **Boxing**: every value is a pointer to a block of memory that begins with metadata. In OCaml:

```
value[0]...value[n]
```

![Diagram](image.png)

- Variants for algebraic datatypes
  - Mark block as reachable:
    - `value[0]...value[n]` scanned by GC
  - Mark block as no scan:
    - `value[0]...value[n]` not scanned by GC
Memory layout

- **Boxing**: every value is a pointer to a block of memory that begins with metadata. In OCaml:

```
header
value[0]
value[1]
value[2]
...
value[n]
```

- **Variants for algebraic datatypes**
  - Mark block as reachable
  - Mark block as no scan

```
value[0]...value[n]
```

- Not scanned by GC

- 2 bits for GC
- 8 bits
- 54 bits
- 54 bits

Caution: This is a manual transcription and may not be 100% accurate.
Memory layout

- **Boxing**: every value is a pointer to a block of memory that begins with metadata. In OCaml:

```
| header | value[0] | value[1] | value[2] | ... | value[n] |
```

Size: 54 bits
Tag: 8 bits
2 bits for GC

Mark block as reachable

Mark block as no scan: value[0]...value[n] not scanned by GC
Memory layout

- **Boxing**: every value is a pointer to a block of memory that begins with metadata. In OCaml:

  ```
  header
  value[0]
  value[1]
  value[2]
  ...  
  value[n]
  ```

  - 54 bits
  - 8 bits
  - 2 bits for GC

  - Variants for algebraic datatypes
  - Mark block as *no scan*: value[0]...value[n] not scanned by GC

  # values in this block
  Mark block as reachable
Finding roots

Stack is a sequence of 64-bit values

- Values (pointers in the heap); i.e., roots
- Saved frame pointers (pointers in the stack)
- Saved return addresses (pointers in code)
Tagged pointers

- Boxing has high overhead

- Pointers are quadword aligned ⇒ last four (low-order) bits are 0
- If a values for a type fit into 63 bits, can use unboxed value, marked with a last (low-order) bit so GC does not scan
- Integers are 63 bit: \( x \) is represented as \( x \ll 1 \mid 1 \)

```
type point = { x : int; y : int }
```
Tagged pointers

- Boxing has high overhead

\[
\text{type point} = \{ \ x : \text{int}; \ y : \text{int} \ \}
\]

- Pointers are *quadword aligned* \(\Rightarrow\) last four (low-order) bits are 0
Tagged pointers

- Boxing has high overhead

```plaintext
type point = { x : int; y : int }
```

- Pointers are *quadword aligned* ⇒ last four (low-order) bits are 0
- If a values for a type fit into 63 bits, can used *unboxed* value, marked with a last (low-order) bit so GC does not scan
  - Integers are 63 bit: \( x \) is represented as \( x « 1 \mid 1 \)
Copying GC

- Mark-and-sweep can lead to memory fragmentation
Copying GC

- Mark-and-sweep can lead to memory fragmentation
- Since GC traverses the heap anyway, might as well compact as it goes
Copying GC

- Mark-and-sweep can lead to memory fragmentation
- Since GC traverses the heap anyway, might as well compact as it goes
- Copying (or Moving) GC
  - Maintain two heaps (roughly equal size), *old* and *new*
  - GC sequentially copies reachable blocks from old heap to new heap
Generational GC

- Generational hypothesis:
  - Most memory becomes garbage quickly after allocation
  - Memory that does not quickly become garbage is likely to not be garbage for a very long time
Generational GC

• Generational hypothesis:
  • Most memory becomes garbage quickly after allocation
  • Memory that does not quickly become garbage is likely to not be garbage for a very long time

• Generational GC: maintain several heaps (“generations”) $G_0$, $G_1$, ...
  • Allocate in $G_0$, and scan frequently
  • Scan $G_1$ less frequently, $G_2$ less frequently than that, ...
  • After collecting garbage in $G_i$, non-garbage is promoted to $G_{i+1}$
Generational GC

- Generational hypothesis:
  - Most memory becomes garbage quickly after allocation
  - Memory that does not quickly become garbage is likely to not be garbage for a very long time

- Generational GC: maintain several heaps ("generations") $G_0, G_1, ...$
  - Allocate in $G_0$, and scan frequently
  - Scan $G_1$ less frequently, $G_2$ less frequently than that, ...
  - After collecting garbage in $G_i$, non-garbage is promoted to $G_{i+1}$

- Complication: intergenerational pointers (from older to newer generation) are new roots that must be managed
Summary

- Reference counting
  - No long pauses (as for tracing GC)
  - Performance penalty for maintaining refcounts, cycles cause leaks

- Mark-and-sweep GC
  - Low memory requirements
  - Memory fragmentation, long pauses

- Copying GC
  - Simple (no free list), less memory fragmentation
  - Cuts available memory in half, long pauses

- Generational GC
  - Shortens average GC pauses; can combine mark-and-sweep & copying GC
  - Relatively complicated, performance penalty for managing intergenerational pointers
Summary

- **Reference counting**
  - No long pauses (as for tracing GC)
  - Performance penalty for maintaining refcounts, cycles cause leaks

- **Mark-and-sweep GC**
  - Low memory requirements
  - Memory fragmentation, long pauses
Summary

- **Reference counting**
  - No long pauses (as for tracing GC)
  - Performance penalty for maintaining refcounts, cycles cause leaks

- **Mark-and-sweep GC**
  - Low memory requirements
  - Memory fragmentation, long pauses

- **Copying GC**
  - Simple (no free list), Less memory fragmentation
  - Cuts available memory in half, long pauses

- **Generational GC**
  - Shortens average GC pauses; can combine mark-and-sweep & copying GC
  - Relatively complicated, performance penalty for managing intergenerational pointers
Summary

- Reference counting
  - No long pauses (as for tracing GC)
  - Performance penalty for maintaining refcounts, cycles cause leaks
- Mark-and-sweep GC
  - Low memory requirements
  - Memory fragmentation, long pauses
- Copying GC
  - Simple (no free list), Less memory fragmentation
  - Cuts available memory in half, long pauses
- Generational GC
  - Shortens average GC pauses; can combine mark-and-sweep & copying GC
  - Relatively complicated, performance penalty for managing intergenerational pointers