COS320: Compiling Techniques

Zak Kincaid

April 18, 2022
Compiling object-oriented languages
Objects

An object consists of Data (attributes) and Behavior (methods).

```java
public 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);  }
}
```

```java
public class IntExpr extends Expr {
    int value;
    public IntExpr(int loc, int k)
    {  super(loc);  value = k;  }
    public int eval(int env)
    {  return value;  }
}
```
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

```java
class AstNode {
    public location getLocation() {
        return loc;
    }
}

public class AddExpr extends Expr {
    public int eval(Env env) {
        return left.eval(env) + right.eval(env);
    }
}

public class IntExpr extends Expr {
    public int eval(int env) {
        return value;
    }
}
```
Subtyping

- Recall the *Liskov substitution principle*: if $s$ is a subtype of $t$, then terms of type $s$ can be used as if they have type $t$ without breaking type safety.
  - If class $B$ extends class $A$, then $B$ is a subtype of $A$
• Recall the *Liskov substitution principle*: if $s$ is a subtype of $t$, then terms of type $s$ can be used as if they have type $t$ 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 $\text{foo}$, 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.

\[
\text{RECORDWIDTH}
\]

\[\vdash \{\text{lab}_1 : s_1 ; \ldots ; \text{lab}_m : s_m\} \prec \{\text{lab}_1 : s_1 ; \ldots ; \text{lab}_n : s_n\} \quad n < m\]
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., typeOf(o) == AddExpr ⇐⇒ o.dispatch = 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., type0f(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., \( \text{typeof}(o) \equiv \text{AddExpr} \iff o.\text{dispatch} = \text{DispatchVector}(\text{AddExpr}) \))
  • 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
    • \( o.\text{dispatch} \neq \text{DispatchVector}(\text{Object}) \) and \( o.\text{dispatch}.\text{parent} = \text{DispatchVector}(C) \), or
    • \( o.\text{dispatch} \neq \text{DispatchVector}(\text{Object}) \) and \( o.\text{dispatch}.\text{parent} \neq \text{DispatchVector}(\text{Object}) \) 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.
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.
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
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
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, whereas *dynamic analyses* collect information 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 or not it will 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.
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 or not it will 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 \( x \rightarrow f = y \)

\[
y \rightarrow \text{count } ++
\]
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 \( x \rightarrow f = y \)

```c
y->count ++
tmp = x->f
y->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 \( x \rightarrow f = y \)

\[
\begin{align*}
&y \rightarrow \text{count} \; \text{++} \\
&\text{tmp} = x \rightarrow f \\
&\text{tmp} \rightarrow \text{count} \; \text{--} \\
&\text{if} \; (\text{tmp} \rightarrow \text{count} \; \text{==} \; 0) \; \text{free} \; (\text{tmp});
\end{align*}
\]
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 \( x \rightarrow f = y \)

\[
\begin{align*}
y \rightarrow \text{count} & \ += 1 \\
tmp & = x \rightarrow f \\
tmp \rightarrow \text{count} & \ -= 1 \\
\text{if} \ (tmp \rightarrow \text{count} & \ == 0) \ \text{free}(tmp); \\
x \rightarrow f & \ = y
\end{align*}
\]
Problem: *cyclic* data structures never get collected
Problem: *cyclic* data structures never get collected

![Diagram](attachment:diagram.png)
Problem: *cyclic* data structures never get collected
Problem: *cyclic* data structures never get collected

```
ref: 2
forward
back
```

```
ref: 2
forward
back
```

```
ref: 1
forward
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
• **Boxing**: every value is a pointer to a block of memory that begins with metadata. In OCaml:
• **Boxing**: every value is a pointer to a block of memory that begins with metadata. In OCaml:
Memory layout

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

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

Diagram:

- 54 bits: size
- 8 bits: tag
- 2 bits for GC

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

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

- Boxing has high overhead

- Pointers are quadword aligned ⇒ last four (low-order) bits are 0

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

```plaintext
p
```

header
x
y

header
320

header
2022
Tagged pointers

- Boxing has high overhead

```
header
x
y
header 320
header 2022
```

- Pointers are quadword aligned ⇒ last four (low-order) bits are 0
- If 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 \( x \ll 1 | 1 \)
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

![Diagram of copying GC]

Old heap:
- `x`
- `w`
- `z`
- `y`

New heap:
- Blocks

Root node:
- Points to the old heap

Diagram showing the copying process from `old` to `new` heap.
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
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
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
- 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") $G_0, G_1, \ldots$
  • 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: inter-generational pointers (from older to newer generation) are new roots that must be managed
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") $G_0, G_1, \ldots$
  - 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: inter-generational 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
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
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