Lecture 18

CIS 341: COMPILERS
Announcements

• Reminder: HW4: Compiling OAT v.1
• DUE: TONIGHT

• HW5: Full OAT: Objects & Typechecking
  – Implement (parts of) the typechecker and compiler for an OO-language
• Available soon
• DUE: April 6th
TYPECHECKING OAT
OAT Classes

- Class-based Object-Oriented Language
- Classes are named
- Explicit Constructor:
  - Marked with the ‘new’ keyword
  - Take a list of parameters
  - Explicitly invoke the superclass constructor
  - Fields with initializers declared in a ‘constructor block’
  - No other code allowed in the constructor

```java
class A {
    new (int x)() // constructor parameters & superclass call
    { int x = x; } // declare field ‘x’

    void print() { // method
        print_string(string_cat("A: x=", string_of_int(this.x)));
        return;
    }
}
```
OAT Inheritance

- Single, Declared Inheritance
  - Omitted superclass defaults to “Object”
  - Fields are all ‘public’ scope, children cannot override them
  - Methods are all ‘public’ scope, children can override them
  - No overloading
    - i.e. a method name appears at most once in each class

```java
class B <: A {
   // extend ‘A’
   new (int x, int y, int z)(x){  // invoke superclass with (x)
      int y = y;  // declare two new fields ‘y’ and ‘z’
      int z = z;
   }

   void print() {  // override the ‘print’ method
      print_string(string_cat("B: x=", string_of_int(this.x)));
      print_string(string_cat("B: y=", string_of_int(this.y)));
      print_string(string_cat("B: z=", string_of_int(this.z)));
   }
}
```
Typechecking

• Hierarchy is single-inheritance
  – Classes inherit only from classes previously declared
• Class scopes are mutually recursive
  – Class fields and methods may mention any class
• Strict distinction between “possibly null” reference types $R?$ and “definitely not null” types $R$
• Checked downcast:

```java
if? (typ x = exp) { ...} else { ...}
```

• Questions:
  – How to initialize fields of objects? (what data is in scope?)

```java
class A {
    new ()() {
        B b = new B();
    }
}
class B {
    new ()() {
        A? a = null;
    }
    void set(A a) {
        this.a = a;
    }
}
```
Subtyping in Oat: Base Types

- \( H \vdash \text{int} <: \text{int} \) \( \text{TYP}_\text{SUB}_\text{INT} \)
- \( H \vdash \text{bool} <: \text{bool} \) \( \text{TYP}_\text{SUB}_\text{BOOL} \)
- \( H \vdash \text{null} <: \text{null} \) \( \text{TYP}_\text{SUB}_\text{NULL} \)
Subtyping in Oat: Reference Types

\[
\begin{align*}
H \vdash \text{null} <: \text{ref} & \quad \text{TYP\_SUB\_NULLS} \\
H \vdash r \text{ ref}_1 <: \text{ref}_2 & \quad \text{TYP\_SUB\_NREF} \\
H \vdash \text{ref}_1 ?: <: \text{ref}_2 ? & \quad \text{TYP\_SUB\_NRREF} \\
H \vdash r \text{ ref}_1 <: \text{ref}_2 & \quad \text{TYP\_SUB\_REF} \\
H \vdash \text{ref}_1 <: \text{ref}_2 & \quad \text{TYP\_SUB\_NRREF} \\
\end{align*}
\]
Subtyping in Oat: Reference Types

\[
\begin{align*}
H \vdash_r \text{string} &: \text{string} & \text{TYP\_SUBRSTRING} \\
H \vdash_r t[\cdot] &: t[\cdot] & \text{TYP\_SUBARRAY} \\
H \vdash C_1 &: C_2 & \text{TYP\_SUBRCLASS} \\
C &: C_1 \{\ldots\} \in H \\
H \vdash C &: C_2 & \text{TYP\_EXTREFL} \\
C &: C_1 \{\ldots\} \in H & H \vdash C_1 &: C_2 & \text{TYP\_EXTTRANS}
\end{align*}
\]
Typechecking OAT: Classes & Constructors

\[ G; H; C_2 \vdash ctr \Rightarrow L \quad G; H; L; C_1; C_2 \vdash fdecl_1 \quad \ldots \quad G; H; L; C_1; C_2 \vdash fdecl_i \]
\[ G; H \vdash \text{class } C_1 <: C_2 \{ ctr \ fdecl_1 \ldots fdecl_i \} \quad \text{ok} \]

\[ C <: C' \{ (t'_1, \ldots, t'_j) \rightarrow C \text{ fieldtyps methodtyps} \} \in H \]
\[ G; H; x_1 : t_1, \ldots, x_i : t_i \vdash \text{exp}_1 : t''_1 \quad \ldots \quad G; H; x_1 : t_1, \ldots, x_i : t_i \vdash \text{exp}_j : t''_j \]
\[ H \vdash t''_1 <: t'_1 \quad \ldots \quad H \vdash t''_j <: t'_j \]
\[ G; H; x_1 : t_1, \ldots, x_i : t_i \vdash f \ \text{decls} \Rightarrow L \]
\[ G; H; C \vdash \text{new} (t_1 x_1, \ldots, t_i x_i)(\text{exp}_1, \ldots, \text{exp}_j)\{\text{decls}\} \Rightarrow L \]
Typechecking OAT: Downcasts

\[
G;H;L \vdash \text{exp} : t' \quad H \vdash t <: t' \quad t \neq t' \quad x \notin L
\]
\[
G;H;L,x:t;rt \vdash \text{block}_1 \quad G;H;L;rt \vdash \text{block}_2
\]
\[
G;H;L;rt \vdash \text{if?}(t \ x = \text{exp}) \ \text{block}_1 \ \text{else} \ \text{block}_2 \Rightarrow L
\]

- The checks \( t <: t' \) and \( t \neq t' \) ensure that this is not a “silly” downcast:
  - If \( t = t' \) then no cast is needed and the ‘true’ branch will always be taken
  - If \( t \) is not a subtype of \( t' \) then the false branch will always be taken
COMPILING CLASSES AND OBJECTS
Code Generation for Objects

• Classes:
  – Generate data structure types
    • For objects that are instances of the class and for the class tables
  – Generate the class tables for dynamic dispatch

• Methods:
  – Method body code is similar to functions/closures
  – Method calls require dispatch

• Fields:
  – Issues are the same as for records
  – Generating access code

• Constructors:
  – Object initialization

• Dynamic Types:
  – Checked downcasts
  – “instanceof” and similar type dispatch
Multiple Implementations

• The same interface can be implemented by multiple classes:

```java
interface IntSet {
    public IntSet insert(int i);
    public boolean has(int i);
    public int size();
}
```

class IntSet1 implements IntSet {
    private List<Integer> rep;
    public IntSet1() {
        rep = new LinkedList<Integer>();
    }

    public IntSet1 insert(int i) {
        rep.add(new Integer(i));
        return this;
    }

    public boolean has(int i) {
        return rep.contains(new Integer(i));
    }

    public int size() {return rep.size();}
}

class IntSet2 implements IntSet {
    private Tree rep;
    private int size;
    public IntSet2() {
        rep = new Leaf(); size = 0;
    }

    public IntSet2 insert(int i) {
        Tree nrep = rep.insert(i);
        if (nrep != rep) {
            rep = nrep; size += 1;
        }
        return this;
    }

    public boolean has(int i) {
        return rep.find(i);
    }

    public int size() {return size;}
}
Consider a client program that uses the IntSet interface:

```java
IntSet set = ...;
int x = set.size();
```

Which code to call?
- `IntSet1.size`?
- `IntSet2.size`?

Client code doesn’t know the answer.
- So objects must “know” which code to call.
- Invocation of a method must indirect through the object.
Compiling Objects

- Objects contain a pointer to a **dispatch vector** (also called a **virtual table** or **vtable**) with pointers to method code.

- Code receiving `set:IntSet` only knows that `set` has an initial dispatch vector pointer and the layout of that vector.

```
IntSet1
  rep:List
  IntSet1.insert
  IntSet1.has
  IntSet1.size

IntSet2
  rep:Tree
    size:int
  IntSet2.insert
  IntSet2.has
  IntSet2.size

IntSet
  set
  ?.insert
  ?.has
  ?.size
```
Method Dispatch (Single Inheritance)

- Idea: every method has its own small integer index.
- Index is used to look up the method in the dispatch vector.

```java
interface A {
    void foo();
}

interface B extends A {
    void bar(int x);
    void baz();
}

class C implements B {
    void foo() {...}
    void bar(int x) {...}
    void baz() {...}
    void quux() {...}
}
```

<table>
<thead>
<tr>
<th>Index</th>
<th>Inheritance / Subtyping:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>C &lt;: B &lt;: A</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
• Each interface and class gives rise to a dispatch vector layout.
• Note that inherited methods have identical dispatch indices in the subclass. (Width subtyping)
Representing Classes in the LLVM

• During typechecking, create a *class hierarchy*
  – Maps each class to its interface:
    • Superclass
    • Constructor type
    • Fields
    • Method types (plus whether they inherit & which class they inherit from)

• Compile the class hierarchy to produce:
  – An LLVM IR struct type for each object instance
  – An LLVM IR struct type for each vtable (a.k.a. class table)
  – Global definitions that implement the class tables
class A {
    new (int x)()
    { int x = x; }

    void print() { return; }
    int blah(A a) { return 0; }
}

class B <: A {
    new (int x, int y, int z)(x){
        int y = y;
        int z = z;
    }

    void print() { return; } // overrides A
}

class C <: B {
    new (int x, int y, int z, int w)(x,y,z){
        int w = w;
    }

    void foo(int a, int b) {return;}
    void print() {return;} // overrides B
}
%Object = type { %_class_Object* }  
%_class_Object = type { }  

%A = type { %_class_A*, i64 }  
%_class_A = type { %_class_Object*, void (%A*)*, i64 (%A*, %A*)* }  

%B = type { %_class_B*, i64, i64, i64 }  
%_class_B = type { %_class_A*, void (%B*)*, i64 (%A*, %A*)* }  

%C = type { %_class_C*, i64, i64, i64, i64 }  
%_class_C = type { %_class_B*, void (%C*)*, i64 (%A*, %A*)*, void (%C*, i64, i64)* }  

@_vtbl_Object = global %_class_Object { }  

@_vtbl_A = global %_class_A { %_class_Object* @_vtbl_Object,  
  void (%A*)* @print_A,  
  i64 (%A*, %A*)* @blah_A }  

@_vtbl_B = global %_class_B { %_class_A* @_vtbl_A,  
  void (%B*)* @print_B,  
  i64 (%A*, %A*)* @blah_A }  

@_vtbl_C = global %_class_C { %_class_B* @_vtbl_B,  
  void (%C*)* @print_C,  
  i64 (%A*, %A*)* @blah_A,  
  void (%C*, i64, i64)* @foo_C }
Method Arguments

• Methods bodies are compiled just like top-level procedures…
• … except that they have an implicit extra argument: 
  this or self
  – Historically (Smalltalk), these were called the “receiver object”
  – Method calls were thought of as sending “messages” to “receivers”

A method in a class...

```java
class IntSet1 implements IntSet {
  ...
  IntSet1 insert(int i) { <body> }
}
```

… is compiled like this (top-level) procedure:

```java
IntSet1 insert(IntSet1 this, int i) { <body> }
```

• Note 1: the type of “this” is the class containing the method.
• Note 2: references to fields inside <body> are compiled like this.field
LLVM Method Invocation Compilation

• Consider method invocation:
  \[
  \begin{align*}
  [G;H;L \vdash e.m(e_1,\ldots,e_n) : t]
  \end{align*}
  \]

• First, compile \([G;H;L \vdash e : C]\)
  to get a (pointer to) an object value of class type \(C\)
  – Call this value \(obj\_ptr\)

• Use \texttt{Getelementptr} to extract the vtable pointer from \(obj\_ptr\)

• Load the vtable pointer

• Use \texttt{Getelementptr} to extract the function pointer from the vtable
  – using the information about \(C\) in \(H\)

• Load the function pointer

• Call through the function pointer, passing ‘\(obj\_ptr\)’ for this:
  \[
  \text{call (cmp_typ t) m(obj\_ptr, \llbracket e_1 \rrbracket, \ldots, \llbracket e_n \rrbracket)}
  \]

• In general, function calls may require \texttt{bitcast} to account for subtyping: arguments may be a subtype of the expected “formal” type
Suppose \( b : B \)

What code for \( b.bar(3) \)?
- \( bar \) has index 1
- Offset = \( 8 \times 1 \)

\[
\begin{align*}
\text{movq} & \ [b], \ %rax \\
\text{movq} & \ [%rax], \ %rbx \\
\text{movq} & \ [%rbx+8], \ %rcx \quad \text{// D.V. + offset} \\
\text{movq} & \ %rax, \ %rdi \quad \text{// “this” pointer} \\
\text{movq} & \ 3, \ %rsi \quad \text{// Method argument} \\
\text{call} & \ %ecx \quad \text{// Indirect call}
\end{align*}
\]
Sharing Dispatch Vectors

- All instances of a class may share the same dispatch vector.
  - Assuming that methods are immutable.
- Code pointers stored in the dispatch vector are available at link time – dispatch vectors can be built once at link time.

- One job of the object constructor is to fill in the object’s pointer to the appropriate dispatch vector.
- Note: The address of the D.V. *is* the run-time representation of the object’s type.
Inheritance: Sharing Code

- Inheritance: Method code “copied down” from the superclass
  - If not overridden in the subclass
- Works with separate compilation – superclass code not needed.
Compiling Static Methods

• Java supports static methods
  – Methods that belong to a class, not the instances of the class.
  – They have no “this” parameter (no receiver object)

• Compiled exactly like normal top-level procedures
  – No slots needed in the dispatch vectors
  – No implicit “this” parameter

• They’re not really methods
  – They can only access static fields of the class
• OAT, Java, C++ classes can declare constructors that create new objects.
  – Initialization code may have parameters supplied to the constructor
  – e.g. new Color(r,g,b);

• Modula-3: object constructors take no parameters
  – e.g. new Color;
  – Initialization would typically be done in a separate method.

• Constructors are compiled just like static methods, except:
  – The “this” variable is initialized to a newly allocated block of memory big enough to hold D.V. pointer + fields according to object layout
  – Constructor code initializes the fields
    • What methods (if any) are allowed?
  – The D.V. pointer is initialized
    • When? Before/After running the initialization code?
Compiling Checked Casts

• How do we compile downcast:

```java
if? (t x = exp) { ... } else { ... }
```

• Reason by cases:
  – t must be either null, ref or ref? (can’t be just int or bool)
• If t is null:
  – The static type of exp must be ref? for some ref.
  – If exp == null then take the true branch, otherwise take the false branch
• If t is string or t[]:
  – The static type of exp must be the corresponding string? Or t[]?
  – If exp == null take the false branch, otherwise take the true branch
• If t is C:
  – The static type of exp must be D or D? (where C <: D)
  – If exp == null take the false branch, otherwise:
    – Emit code to walk up the class hierarchy starting at D, looking for C
    – If found, then take true branch else take false branch
• If t is C?:
  – The static type of exp must be D? (where C <: D)
  – If exp == null take the true branch, otherwise:
    – Emit code to walk up the class hierarchy starting at D, looking for C
    – If found, then take true branch else take false branch
“Walking up the Class Hierarchy”

• A non-null object pointer refers to an LLVM struct with a type like:

\[
\%B = \text{type} \{ \_\text{class}_B*, i64, i64, i64 \}
\]

• The first entry of the struct is a pointer to the vtable for Class B
  – This pointer is the dynamic type of the object.
  – It will have the value \_vtbl\_B

• The first entry of the class table for B is a pointer to its superclass:

\[
\_\_\text{vtbl}_B = \text{global} \_\_\text{class}_B \{ \_\_\text{class}_A* \_\_\text{vtbl}_A, \\
  \text{void} (\%B*)* \_\text{print}_B, \\
  i64 (\%A*, \%A*)* \_\text{blah}_A \}
\]

• Therefore, to find out whether an unknown type X is a subtype of C:
  – Assume C is not Object (ruled out by “silliness” checks for downcast)
  LOOP:
  – If X == \_\_\text{vtbl}_\_Object then NO, X is not a subtype of C
  – If X == \_\_\text{vtbl}_C then YES, X is a subtype of C
  – If X = \_\_\text{vtbl}_D, so set X to \_\_\text{vtbl}_E where E is D’s parent and goto LOOP