cs153: Compiling Object Oriented Languages

Gregory Malecha

Harvard SEAS

October 16, 2009

Roadmap of Language Features

- So far we've compiled the following:
  - Expressions (+, -, ==, ...)
  - Statements (`if`, `for`, `while`,...)
  - Procedures
  - Structures/records (`struct`)
- These are most of the features of C.
- What about object-oriented languages?
  - Classes
  - Interfaces
- Let's look at some examples.

Outline

1. Simple Classes
2. Inheritance
3. Dynamic Dispatch
4. Language Mechanisms & Performance Considerations

Simple Classes – Chess Pieces

```
class Piece {
  var row, col;
  var color;

  Piece(color, row, col) {
    this.row = row; this.col = col; this.color = color;
  }

  func move(row,col) {
    this.row = row; this.col = col;
  }

  func getRow() { return this.row; }
  func getCol() { return this.col; }
}
```

- First thing we should ask ourselves:
  - **Does this look like something that we've already done?**
- Convert the `class` to a `struct`.
  - Wrap the fields in a `struct`/`record`.
  - Lift the methods to the top level.
  - Add *this* parameter to each method.
- So let's convert the `Piece` class.

Naïve Class Compilation (Data & Constructor)

```
struct Piece {
  var row, col, color;
}

func make_Piece(color, row, col) {
  let res = malloc(sizeof(Piece)) in {
    init_Piece(res, color, row, col);
    return res;
  }
}

func init_Piece(this, color, row, col) {
  this.row = row; this.col = col;
  this.color = color;
}
```

Naïve Class Compilation (Data & Constructor)

```
func Piece_move(this, row, col) {
  this.row = row;
  this.col = col;
}

func Piece_getRow(this) {
  return this.row;
}

func Piece_getCol(this) {
  return this.col;
}
```

More Advanced Features

- Our approach works great for very simple classes.
- But how does it work with other features:

  Inheritance
  : Inherit the fields and methods of a class to build a new class.

  Dynamic dispatch
  : Resolve the method to call based on the type of the object at runtime.

Outline

1 Simple Classes

2 Inheritance

3 Dynamic Dispatch

4 Language Mechanisms & Performance Considerations

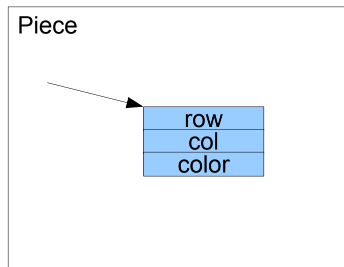
Simple Example – Pawns

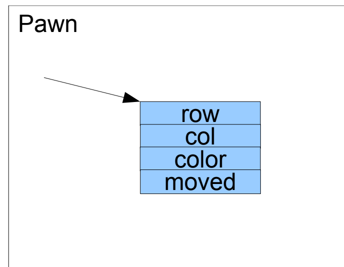
```
class Piece {
   var row, col, color;
   Piece(color, row, col) {
     this.row = row; this.col = col;
     this.color = color;
   }
}
class Pawn extends Piece {
  var moved;
  Pawn(color, row, col) {
    Piece::Piece(color, row, col);
    moved = 0;
  }
  func move(row,col) {
    this.row = row; this.col = col;
    moved = 1;
  }
}
```

Compiling the Pawn

- Since `Pawn` extends `Piece`, we inherit.
  - The fields.
  - The methods.
  - The ability to call methods on the parent class, e.g. `Piece::Piece`
- We can augment the record with the new fields.
  - In order to be able to call the parent we need to have the same memory layout.

Memory Layout of Subclasses





- Access `structs` using the offset from the base pointer.
  - Code to access `p.row` is the same if `p` is a `Piece` or a `Pawn`.
  - Same for `col` and `color`.
- Therefore code written to work with `Piece` will do the same thing for `Pawn`.

Compiling Pawns

```
struct Pawn {
  var row, col, color;
  var moved;
}

func make_Pawn(color, row, col) {
  let res = malloc(sizeof(Pawn)) in {
    init_Pawn(res, color, row, col);
    return res;
  }
}

func init_Pawn(this, color, row, col) {
  /* Piece::Piece(color, row, col) */
  init_Piece(this, color, row, col);
  this.moved = 0;
}
```

Compiling Pawns

- The prefix of `Pawn` looks the same as the prefix of `Piece`, we can treat `Pawns` like we treat `Pieces`.

```
Piece p = new Pawn(WHITE, 0, 1);
p.move(0, 3);
```

  - The above function would call `Piece_move`.
  - We want to call `Pawn_move`.
- Currently we determine the function to call by the type.
  - This style of object-oriented compilation requires types.
  - This is naïve classes *a. la.* C++ without `virtual`.
  - More traditional object-oriented languages use dynamic dispatch.

Outline

1. Simple Classes
2. Inheritance
3. Dynamic Dispatch
4. Language Mechanisms & Performance Considerations

Compiling for Dynamic Dispatch

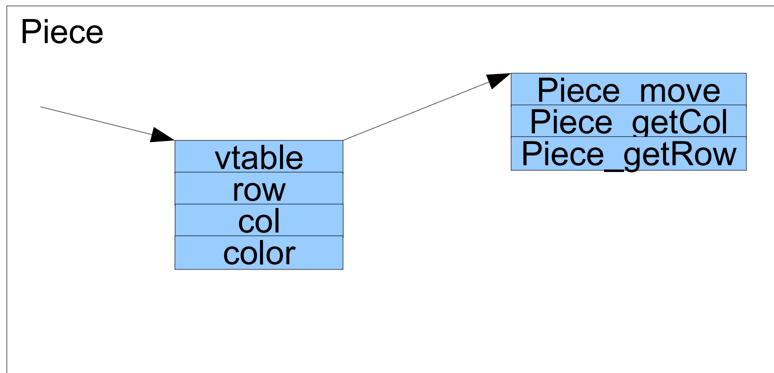
```
Piece p = new Pawn(WHITE, 0, 1);
p.move(0, 3);
```

- At compile time, we only know that `p` is a `Piece`.
  - Based on the *compile-time type* of the variable `p` (its declaration).
  - This is conservative, but necessary for consistency.
    - What if we got `p` through a function parameter.
    - What if we got `p` from a function call.
- We need some way to lookup what function to call at run-time.
  - "All problems in computer science are solved by an extra level of indirection."
  - Build a table of function pointers and store it in the structure.
  - Called the *virtual function table* or *vtable*.
  - Calls to methods indirect through the jump table.

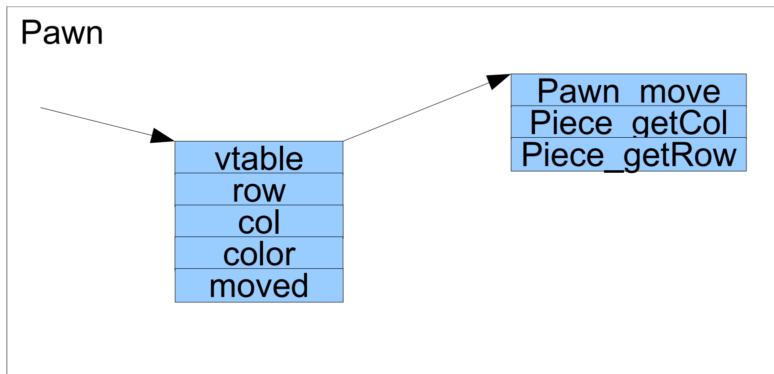
Compiling with a vtable

```
struct Piece_iface { /** The type of vtables **/
  var move;
  var getRow;
  var getCol;
}

/** Prefix object structs with a vtable pointer **/
struct Piece {
  var vtable;
  var row, col, color;
}

struct Pawn {
  var vtable;
  var row, col, color;
  var moved;
}
```

Memory Layout



Memory Layout



Initializing the vtable

```
Piece_vtable = { Piece_move, Piece_getRow, Piece_getCol };
func make_Piece(color, row, col) {
  let this = malloc(sizeof(Piece)) {
    this.vtable = Piece_vtable;
    init_Piece(this, color, row, col);
    return this;
  }
}

Pawn_vtable = { Pawn_move, Piece_getRow, Piece_getCol };
func make_Pawn(color, row, col) {
  let this = malloc(sizeof(Pawn)) {
    this.vtable = Pawn_vtable;
    init_Pawn(this, color, row, col);
    return this;
  }
}
```

Dynamic Dispatch & Function Calls

- Indirect function calls are a little bit more complex.
  1. Save caller-save registers.
  2. Update stack pointer.
  3. Look up the virtual function table.
  4. Index into table based on the function name.
  5. First parameter is `this`.
  6. `jal` to the function address.
- There's a lot of overhead in this.
  - At least two memory references.
  - Extra space requirements to store vtables (per class).
  - Extra space on every object (per object).

Outline

1 Simple Classes

2 Inheritance

3 Dynamic Dispatch

4 Language Mechanisms & Performance Considerations

Things to Think About — Performance & Optimization

- Multiple inheritance and interfaces.
    - Every interface has it's own representation.
    - Need to wrap objects to include the correct vtable pointers.
    - Casts are not no-ops.
        - They require allocation!
    - Some optimization techniques using negative structure indicies.
- Some ways to mitigate the performance penalty.
    - Static analysis to determine run-time types.
        - Call method on newly constructed object.
        - Call method inside `final` constructor.
        - Calling `final` methods.
    - We can always optimize the fast path.

Things to Think About — Language Features

- Other abstractions to address the same issues.
    - "A Comparative Study of Language Support for Generic Programming". (linked on website)
    - **Haskell type classes** pass the vtable around.
        - Avoids the extra space for each object.
    - **ML modules** avoid this by specialization.
        - No dynamic dispatch means very little overhead.
        - But increase code size. (Maybe bad cache performance?)
- Do we really need to declare all the interfaces?
    - Scripting languages often implement objects as finite maps.
- *Multi-parameter dispatch*
    - Dispatch on the type of the arguments as well.
    - Very sparse dispatch matrices. Ways to compress them?

Recap

- **Don't reinvent the wheel** Reduce new concepts to old ones.
  - Classes, inheritance → structs.
  - Dynamic dispatch → indirection, jump tables.
- You've implemented pretty much all of this in your compiler already.
- The only thing remaining is types.
  - Necessary for this type of compilation.
  - There are alternatives (e.g. hash tables).