Implementing OOPLs

Tobias Wrigstad
tobias.wrigstad@it.uu.se

Features in OOPLs

• Polymorphism and subtyping

• Dynamic binding—virtual calls need run-time support

• Run-time type testing

• Inheritance

Only consider class-based OOPLs. Prototype-based roughly the same.

Want: fast access to parts of “unknown” structures

Want: minimal on-line search time for maximal flexibility

Want: fast access to parts of "unknown" structures

Want: minimal on-line search time for maximal flexibility

Want: minimal on-line search time for maximal flexibility

Want: minimal on-line search time for maximal flexibility

Want: minimal on-line search time for maximal flexibility
Features in OOPLs

- Polymorphism and subtyping
  
  *Want:* fast access to parts of "unknown" structures

- Dynamic binding—virtual calls need run-time support
  
  *Want:* minimal on-line search time for maximal flexibility

- Run-time type testing
  
  *Want:* fast tests and cheap metadata

- Inheritance

Only consider class-based OOPLs. Prototype-based roughly the same.

Outline

- Field access
- Method calls
- Calls through interface types
- Options for untyped languages
- Call-site optimisation techniques

Simplified World View

- Accessing C struct
  
  `x.f` # address of x + compile-time calculated offset of f in x's type

- Accessing instance variable
  
  `x.f` # location (existence) of f depends on run-time type of x's value

```c
struct Point {
    int x;
    int y;
    Colour* colour;
};
```
Unified access is desirable

- Simple—can use same method of access everywhere
- Not as simple as records
  - Inheritance
  - Multiple inheritance
  - Separate compilation
  - Dynamic class loading

Prefixing for unified access

```java
class Point2D {
    int x;
    int y;
}

class Point3D < Point 2D {
    int z;
}

Point2D p = Point3D();
p.y # can be translated to – *(p+1)
```

Prefixing with multiple inheritance

```java
class Point2D {
    int x;
    int y;
}

class Coloured {
    colour c;
}

class ColouredPoint < Point2D, Coloured {
}
```

Prefixing with multiple inheritance (cont’d)

```java
Coloured p = ColouredPoint();
p.c # works fine, *(p+0) still denotes a colour c

Point2D p = ColouredPoint();
p.x # breaks! *(p+0) is a colour, not an integer

Point2D p = ColouredPoint();
p.c # breaks! *(p+0) is an integer, not a colour
```
C++: pointer shifting

- Modify pointer address whenever type of variable changes
  - Pick any of the possible embeddings
- Keeps field access a constant-time operation (good)
- Implicit casts gets a run-time cost (bad)
- Tricky if type information gets lost (e.g., void* pointers) (bad)

```cpp
ColouredPoint cp = ColouredPoint() // cp
Point2D p = cp // p
Coloured c = cp // c
```

Limitations

- Does not work in an untyped setting
- Subtyping the only way to extend a class layout
  - Does not work with "open classes"
  - Does not work if class layout can be modified dynamically

- For untyped/dynamic/… languages
  - Store fields in a hash table, loads and stores are hash table accesses
    (talk more about this later)

Not the Whole Story
Not the Whole Story

• Check access control at run-time (e.g. due to separate compilation)
  Store a table of flags for variables in a class
  Perform expensive access control check

• When does offset calculation happen?
  Load time—might trigger propagating inclusion
  Run-time—too expensive?

• Opportunity for optimisation, if classes are invariant
  Direct second access after slow first-time check
  JIT:ed code can omit checks and use calculated offsets

Object Layout

• Languages with GC, RTTI, etc. will use additional overhead per object, e.g.,
  Forward pointer space for copying GC, mark bits, etc.
  Pointer to object's class
  Sometimes, object can be broken up in slices (e.g., for fragmentation-sensitive applications)
  Monitor for storing a lock

• Push as much shared information into the class
• Java and C++ are about equally efficient wrt. object layout (except for POD)
• Dynamic languages generally more space demanding
Example: JRuby

- JRuby is considered an efficient implementation of Ruby on the JVM
- Ruby is a dynamic language, fields are ultimately stored in Java hash maps
- Empty JRuby object uses ~72 bytes
  - plus 40 bytes per variable for a 32 bit VM
  - plus 64 bytes per variable for a 64 bit VM
- Compare with an empty Java object that should use <12 bytes

Outline

- Field access
- Method calls
- Calls through interface types
- Options for untyped languages
- Call-site optimisation techniques

Simplified World View (cont’d)

- Calling functions and procedures:
  - `foo(y)` # location of foo can be determined at compile-time (link-time)
  - Allows inlining to reduce call-time overhead, etc.
Simplified World View (cont’d)

- Calling functions and procedures:
  foo(y) # location of foo can be determined at compile-time (link-time)
  Allows inlining to reduce call-time overhead, etc.
- Calling closures:
  foo(y) # push y onto stack, jump to address of foo (-ish)

Method call in untyped OOPLs

- Search inheritance hierarchy from x’s class for foo/1
  Very slow lookup time! (function of #classes and #methods)
- Use a hash table in each object x.foo(y) -> push x,y + jmp x.get(foo/1)
  Still much slower than a procedure call!
- Make an entry for each method in the object just like a field
  Fast, constant-time dispatch (load + jump)
  Very large objects
- Optimisation: share method entries for objects of same class in vtables
  Much smaller object for the cost of one extra indirection

Vtables for efficient dispatch

- Virtual tables
- Complication due to multiple inheritance

```c
void* header = p-1;
void* vtable = *header;
int (*getY)() = vtable+1;
int temp = getY(p); // this
```

...code...
Vtable prefixing with single inheritance

```cpp
class Point2D {
    int x, y;
    int getX() ...
    int getY() ...
}
```

class Point3D < Point2D {
    int z;
    int getZ() ...
}

Vtable prefixing with multiple inheritance

```cpp
class Point2D {
    int x, y;
    int getX() ...
    int getY() ...
}
```

class Coloured {
    colour c;
    colour getC() ...
}

class ColouredPoint < Point2D, Coloured { }

```cpp
class Coloured {
    colour c;
    colour getC() ...
}
```

class ColouredPoint < Point2D, Coloured { }
Vtable prefixing with multiple inheritance

```java
class Point2D {
    int x, y;
    int getX() ...
    int getY() ...
}

class Coloured {
    colour c;
    colour getC() ...
}

class ColouredPoint < Point2D, Coloured {
    ...
}
```

Notably, the header for ColouredPoint and Point2D can be merged since there are no members in the class.

Outline

- Field access
- Method calls
- Calls through interface types
- Options for untyped languages
- Call-site optimisation techniques

Invocation through interface types

- Observation: Impossible to achieve uniform access through interface types
- **Technique 1**: translate interface offsets to implementing class offsets
  
  Each class has a dispatch "itable" for each interface it implements
  
  On call: search for correct itable by some interface id and use it for dispatch

```
Header
f1 f2
vtable

0 1 2
Interface IDs

0 1 2 3 4
itable

...code...
```

Optimisation: give each interface a global id, give each class an interface array with pointers to itables for interfaces it implements, an null pointers otherwise (Can truncate size after last implemented interface)
Invocation through interface types (cont’d)

**Technique 2:** "Selector-indexed tables"

- Give each interface method a numerical id (e.g., at load-time)
- Give each class an itable for all methods in all interfaces
- Dispatch becomes additional indirection—lookup in the selector index table
- Fast but very costly wrt. space

![Diagram of selector-indexed tables]

**Benefit:** constant-time to find correct itable

Optimisation: Use graph colouring to re-use same numerical index for methods that may never be called on the same classes.

Can greatly reduce size of all itables.

Outline

- Field access
- Method calls
- Calls through interface types
- Options for untyped languages
- Call-site optimisation techniques
Alternatives for untyped code

- No type information to base field, vtable (or itable) offsets from
- Storing closures in hash tables is not space efficient for methods (but for fields)
- Naïve implementation:
  - Search receiver's class' hierarchy for method signature, and call
    Slow, terrible worst-case times
- Possible to use static table internally, esp. with JIT compilation
  For example, each class trivially knows it super class statically

Global dynamic table

- Global cache in the form of a hash table indexed by class + signature
  Translate invocations into lookups in hash table
  Cache miss: perform expensive search through class hierarchy, then update cache
- Flush (part of) the cache as a result of reflective operations that change classes
- Space costs are reasonable
- Overhead is reasonable and table is constructed incrementally

Is it effective?

OK average call time, bad worst-case call time

One dispatch table per message names

- Create a separate table per unique signature mapping classes to methods
- Each call site can statically know what table to consult
- Performance is better than the single global table
  Especially if methods names are relatively unique
  (Smalltalk names fare quite well here)
- Per-signature dispatch tables can be constructed incrementally
Outline

• Field access
• Method calls
• Calls through interface types
• Options for untyped languages
• Call-site optimisation techniques

Inlining

• Difficult to do in flexible programs
  Analysis of e.g., possible run-time binding is limited by dynamic loading

• Java
  Can possibly inline final and static methods
  JITing allows more aggressive inlining

  class A {
    int foo(int x, int y) { return x+y; } // can be inlined by HotSpot if ÆB <: A
  }

• Dynamic class loading requires remembering JITed methods and storing their
  prerequisites (e.g. ÆB <: A above)
  Check prerequisites and possibly "retire" (unoptimise) compiled code on
  class loading

Call-site optimising through caching

• Useful esp. for untyped code and interface calls
• Techniques addressed here:
  Inline Caching [Deutsch and Shiffman, 1984]
  Polymorphic Inline Caching [Hölzle et al., 1991]

Inline Caching [Deutsch and Shiffman 1984]

• Each call site has a single-element lookup cache
  Remember what actual method was called for class of last receiver
  Next call, if same receiver we can get method immediately from cache
  Cache miss: slow-path through lookup, update caches

• Efficient implementation through self-modifying code
  x.m(…)
  next call

  c = x.class
  am = c.search(m/1)
  jump am
  switch (x.class) {
  case c: jump am; break
default:
  c = x.class
  am = c.search(m/1)
  jump am}
Inline Caching [Deutsch and Shiffman 1984]

- Each call site has a single-element lookup cache
- Remember what actual method was called for class of last receiver
- Next call, if same receiver we can get method immediately from cache
- Cache miss: slow-path through lookup, update caches

- Efficient implementation through self-modifying code

```
x.m(…)
c = x.class
am = c.search(m/1)
jump am
```

Is it effective?
Smalltalk: 90-95% cache hit frequency and = 4 instructions for fast path.
Slow path real slow though.

Polymorphic inline caching [Hölzle and Ungar, 1991]

- Handles polymorphic and megamorphic call sites
- Extension is simple: use a multi-element cache
- Allows relatively fast dispatch for polymorphic call sites
- If several classes are equally common, performance degrades
- Can get large space overhead (esp. for megamorphic call sites)

```
switch (x.class) {
case Foo: jump m1 break;
case Bar: jump m2 break;
default: ... # lookup + install
}
```

call when x is a Bar

Extensions: change case ordering based on hit frequency.
But will it earn back the incurred run-time overhead?

Improvement: use binary search instead of linear switch.
Requires global knowledge (to map classes to integer ids)
Little additional overhead
public IRubyObject call(IRubyObject caller, IRubyObject self, IRubyObject arg1) {
    RubyClass selfType = pollAndGetClass(self);
    if (CacheEntry.typeOk(localCache, selfType)) {
        return localCache.method.call(self, selfType, arg1);
    }
    return cacheAndCall(caller, selfType, self, arg1);
}

(Notably not polymorphic)