Expressing high level optimizations within LLVM

Artur Pilipenko
artur.pilipenko@azul.com

AZUL SYSTEMS®

This presentation describes advanced development work at Azul Systems and is for informational purposes only. Any information presented here does not represent a commitment by Azul Systems to deliver any such material, code, or functionality in current or future Azul products.

AZUL SYSTEMS®

Azul Systems

- We make Java virtual machines
- Known for scalable, low latency JVM implementation
- We use LLVM to build high performance, production quality JIT compiler for our Java VM

AZUL SYSTEMS®

LLVM for a JIT for Java

- There are certain challenges
  - GC interaction [1]
  - Interaction with the runtime [2]
  - Expressing high-level optimizations

[1] http://llvm.org/devmtg/2014-10/Slides/Reames-GarbageCollection.pdf
[2] http://llvm.org/devmtg/2015-10/slides/DasReames-LLVMForAManagedLanguage.pdf

AZUL SYSTEMS®

Why is it important?

- High tier JIT
  - Main goal is peak performance
  - Compile time is less important
- To achieve good performance we need to make use of high-level semantics of the language

AZUL SYSTEMS®

Motivational Example

- Methods are virtual by default in Java
  - We need to devirtualize in order to inline
  - To devirtualize we need to know possible types of the receiver objects
- It's not only about devirtualization
  - Type check optimizations, aliasing, etc.

AZUL SYSTEMS®

High-level Optimizations and LLVM

- Originally targeted to C/C++
  - Rather low-level IR
  - Some bits of high-level information can be provided using attributes/metadata, like TBAA
- Has been used for other languages recently
  - Swift, Webkit, HHVM, Microsoft LLILC, …

AZUL SYSTEMS®

Split Optimizer

- High-level IR to perform high-level optimization
- Lower it to LLVM IR for mid-level optimizations and code generation

Swift, HHVM, Webkit, …

AZUL SYSTEMS®

Split Optimizer

- High-level optimizer
  - IR, analyses, transformations, infrastructure
- Didn’t really want to write everything from scratch
- This infrastructure already exists in LLVM

AZUL SYSTEMS®

Embedded High-Level IR

- Express the required information in LLVM IR
- Introduce missing optimizations
- Teach existing parts of the optimizer to make use of new information

AZUL SYSTEMS®

Agenda

- Abstractions
- Java Type Framework
- Exploiting Java Types
  - Java Specific Optimizations
  - Existing Optimizations

AZUL SYSTEMS®

Agenda

- Abstractions
- Java Type Framework
- Exploiting Java Types
  - Java Specific Optimizations
  - Existing Optimizations

AZUL SYSTEMS®

Abstractions

- Functions with the semantics known by the optimizer
  - Similar to intrinsics, but have an IR implementation
- Late inlining mechanism
  - Abstractions are inlined at specific points of the pipeline
  - Optimization phases separation
  - Gradual lowering

AZUL SYSTEMS®

Abstractions Example

```
define i32 @get_class_id(i8 addrspace(1)* %object)
    "late-inline"="2" {
...
}

define i1 @is_subtype_of(i32 %parent_id, i32 %child_id)
    "late-inline"="1" {
...
}
```


AZUL SYSTEMS®

Abstractions Example

```
define i32 @get_class_id(i8 addrspace(1)* %object)
    "late-inline"="2" {
...
}

define i1 @is_subtype_of(i32 %parent_id, i32 %child_id)
    "late-inline"="1" {
...
}
```

Inlined after phase 2 and 1 accordingly

AZUL SYSTEMS®

Abstractions Example

```
// Java code
static boolean isString(Object obj) {
    return obj instanceof String;
}
```

```
; LLVM IR
define i1 @isString(i8 addrspace(1)* %obj) {
  ...
  %class_id = call i32 @get_class_id(i8 addrspace(1)* %obj)
  %result = call i1 @is_subtype_of(i32 <StringID>, i32 %class_id)
  ret i1 %result
}
```


AZUL SYSTEMS®

Optimization Over Abstractions

- For example:
  - Lock coarsening/elision
  - Redundant GC barrier/polls elimination
  - Allocation and initialization sinking

AZUL SYSTEMS®

Upstream Late Inlining

- Does this mechanism makes sense upstream?
- Function attribute to specify the inlining phase
  - “late-inlining”= “phase”
- EnableLateInlining pass to do the inlining
  - PM.add(createEnableLateInliningPass(“phase”))

AZUL SYSTEMS®

Agenda

- Abstractions
- Java Type Framework
- Exploiting Java Types
  - Java Specific Optimizations
  - Existing Optimizations

AZUL SYSTEMS®

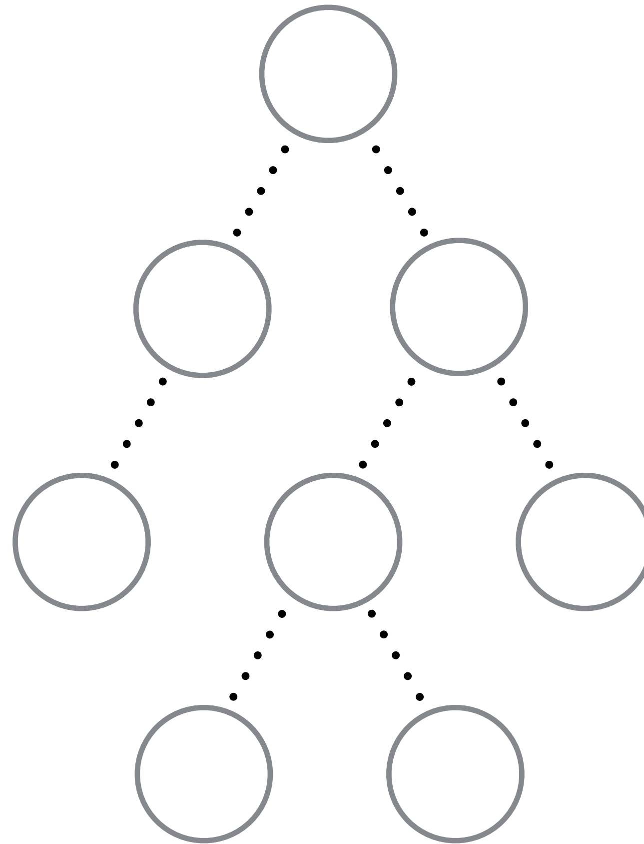
Java Type Framework

- A mechanism to reason about properties of the objects pointed to by reference values
    - Specifically, Java classes of the objects

AZUL SYSTEMS®

Java Class Hierarchy

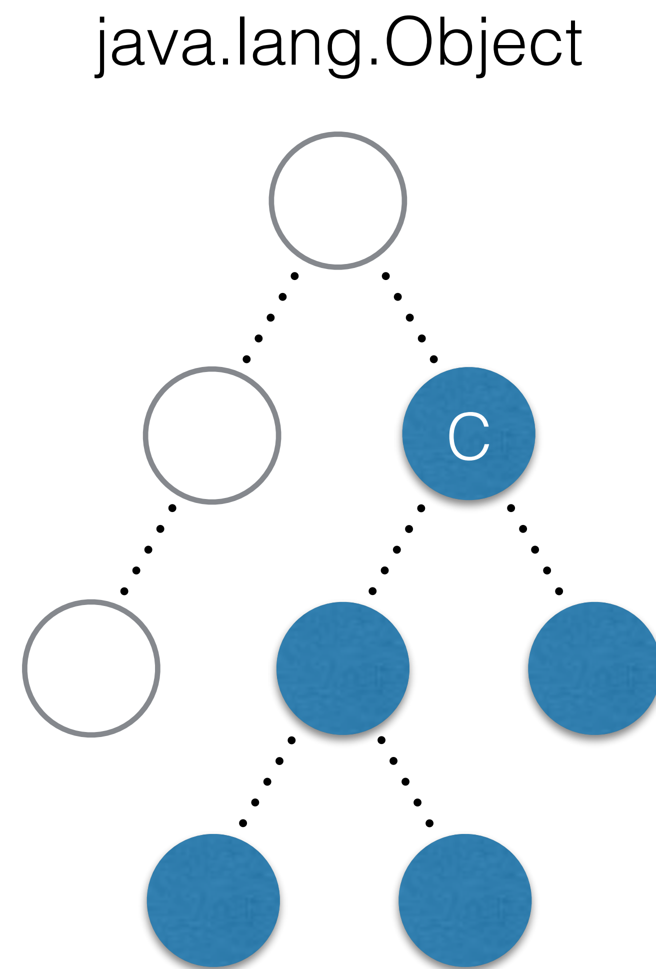
java.lang.Object



AZUL SYSTEMS®

Java Class Hierarchy

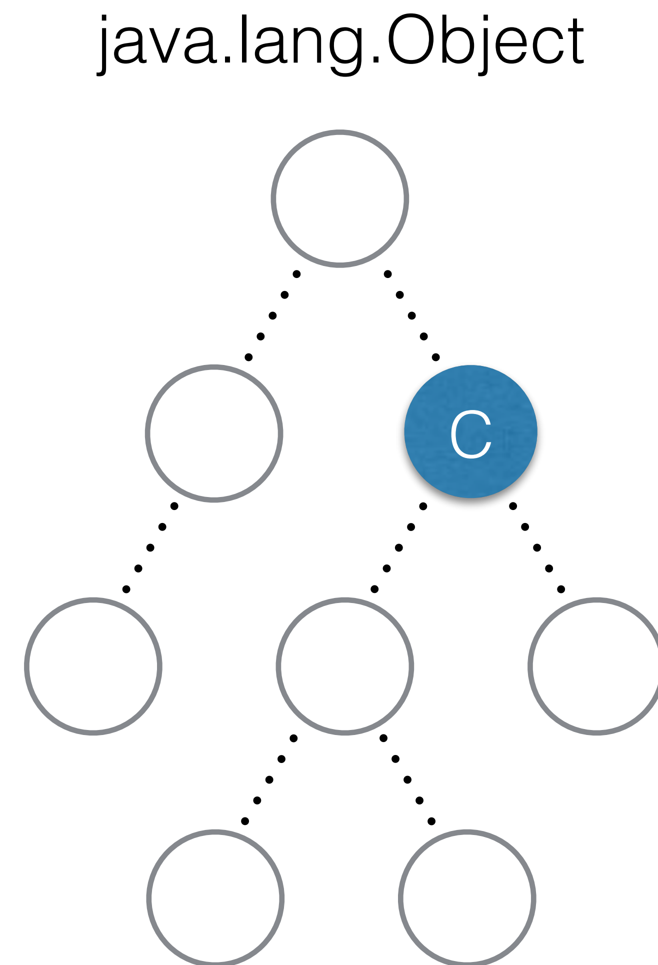
```
C c = ...
```



AZUL SYSTEMS®

Java Class Hierarchy

```
C c = new C();
```



AZUL SYSTEMS®

JavaType

```
// Defines a set of Java classes
struct JavaType {
  // An ID of a Java class or interface
  uint64_t ClassID;
  // If set the class defined by ClassID is the only class
  // in the set.
  // Otherwise the set includes all the subclasses of that
  // class.
  bool IsExact;
};
```


AZUL SYSTEMS®

JavaType Operations

- JavaType union(JavaType, …)
- Optional<JavaType> intersect(JavaType, …)
- bool isSubtypeOf(JavaType, JavaType)
- bool canTypesIntersect(JavaType, JavaType)

AZUL SYSTEMS®

JavaType Limitations

- Might not be the most precise type system for Java code, but something we got away with thus far
  - Can’t represent unions of types precisely
  - Can’t represent multiple inheritance of interface types

AZUL SYSTEMS®

Java Type Analysis

```
Optional<JavaType> getJavaType(Value *V,
                               Instruction *CtxI = nullptr,
                               DominatorTree *DT = nullptr)
```

- Currently a value-tracking style analysis
- Relatively expensive — it would be good to cache the results
- Context sensitive/insensitive queries
- Can conservatively return None

AZUL SYSTEMS®

Java Type Base Facts

- Attached to IR in the form of attributes/metadata
- Emitted by the front-end based on types in Java bytecode
- Inferred by the optimizer

AZUL SYSTEMS®

Base Facts Examples

```
; Attributes on arguments and return values
define "java-type-class-id"="234" i8 addrspace(1)* @foo(
                i8 addrspace(1)*
                 "java-type-exact"
                 “java-type-class-id"="193" %arg) {
  ; Metadata on loads
  load i8 addrspace(1)*, i8 addrspace(1)* addrspace(1)* %p,
    !java-type-class-id !{i32 234}

  ; Attributes on call site return values
  call "java-type-class-id"="234" "java-type-exact"
    i8 addrspace(1)* @new.instance(i32 234)

  ; Attributes on call site arguments
  call void @foo(i8 addrspace(1)*
                 "java-type-class-id"="234" %arg)
}
```


AZUL SYSTEMS®

Context-Insensitive Analysis

- Look at the value to get context-insensitive result
  - Base facts from metadata and attributes
  - If the value is a PHI node recursively queries the types of the incoming values and calculates the union of the incoming types
  - Some of the Java methods with known semantics (Object.clone)

AZUL SYSTEMS®

Context-Sensitive Sharpening

- If context is provided perform context-sensitive sharpening from dominating conditions
- Walk the dominators tree and look for type checks on the object in question

AZUL SYSTEMS®

Exact Type Checks

```
%cid = call i32 @get_class_id(i8 addrspace(1)* %object)
%cond = icmp eq i32 %cid, <SomeClassID>
br i1 %cond, label %true, label %false

true:
; JavaType {<SomeClassID>, exact} is implied
```


AZUL SYSTEMS®

Non-Exact Type Checks

```
%cid = call i32 @get_class_id(i8 addrspace(1)* %object)
%cond = call i32 @is_subtype_of(i32 <SomeClassID>,
                                i32 %cid)
br i1 %cond, label %true, label %false

true:
; JavaType {<SomeClassID>, non-exact} is implied
```


AZUL SYSTEMS®

Java Type Analysis Result

- The final result is an intersection of
  - Context-insensitive result
  - Types from all the dominating type checks

AZUL SYSTEMS®

Metadata Healing

- We use metadata to represent type information
- Metadata can be dropped
- Sometimes it inhibits optimizations
- We can heal metadata on loads using JavaTypes of the accessed objects

AZUL SYSTEMS®

Metadata Healing

- InstCombine rule for loads and stores
  - Get the underlying object for the memory access
  - Get JavaType of the underlying object value
  - Ask the VM about the layout of the object
  - Update the metadata accordingly

AZUL SYSTEMS®

Agenda

- Abstractions
- Java Type Framework
- Exploiting Java Types
  - Java Specific Optimizations
  - Existing Optimizations

AZUL SYSTEMS®

Devirtualization

- Indirect call sites come in different shapes and forms
- Depending on the profile information the front-end may generate
  - Explicit lookup
  - Profile guided call sites: guarded direct calls for predicted targets

AZUL SYSTEMS®

Explicit Lookup

- Explicit lookup

```
%target = call i8* @resolve_virtual(i8 addrspace(1)* %object,
                                    i32 <vtable_index>)
%target.casted = bitcast i8* %target to void (i8 addrspace(1)*)*
call void %target.casted(i8 addrspace(1)* %object)
```


AZUL SYSTEMS®

Explicit Lookup

- Explicit lookup

```
%target = call i8* @resolve_virtual(i8 addrspace(1)* %object,
                                    i32 <vtable_index>)
%target.casted = bitcast i8* %target to void (i8 addrspace(1)*)*
call void %target.casted(i8 addrspace(1)* %object)
```

Devirtualization via constant folding of resolve_virtual abstractions for known receiver JavaTypes

AZUL SYSTEMS®

Profile Guided Call Sites

- Monomorphic call site with deoptimize fallback

```
if (get_class_id(%receiver) == expected_class_id)
  call expected_target
else
  call deoptimize
```


AZUL SYSTEMS®

Profile Guided Call Sites

- Monomorphic call site with deoptimize fallback

```
if (get_class_id(%receiver) == expected_class_id)
  call expected_target
else
  call deoptimize
```

To “devirtualize” the call site we need to fold the comparison away

AZUL SYSTEMS®

Profile Guided Call Sites

- Trimorphic call site with explicit lookup fallback

```
switch (get_class_id(%receiver)) {
case expected_receiver_1: call expected_target_1; break;
case expected_receiver_2: call expected_target_2; break;
case expected_receiver_3: call expected_target_3; break;
default:
  %target = resolve_virtual(%receiver, i32 <vtable_index>)
  call %target
}
```


AZUL SYSTEMS®

Profile Guided Call Sites

- Trimorphic call site with explicit lookup fallback

```
switch (get_class_id(%receiver)) {
case expected_receiver_1: call expected_target_1; break;
case expected_receiver_2: call expected_target_2; break;
case expected_receiver_3: call expected_target_3; break;
default:
  %target = resolve_virtual(%receiver, i32 <vtable_index>)
  call %target
}
```

To “devirtualize” the call site we need to prune switch cases

AZUL SYSTEMS®

Control Flow Simplification

```
%cid = call i32 @get_class_id(i8 addrspace(1)* %object)
%cond = icmp eq i32 %cid, <SomeClassID>
```

- Let T be JavaType of %object, P be JavaType {SomeClassID, exact}
  - If T.IsExact => fold get_class_id to a constant
  - If !canTypesIntersect(T, P) => fold the comparison to false
- Use the same idea to prune switch cases

AZUL SYSTEMS®

Type Check Optimizations

```
%cid = call i32 @get_class_id(i8 addrspace(1)* %object)
%cond = call i32 @is_subtype_of(i32 <parent>, i32 %cid)
```

- Let T be JavaType of %object, P be JavaType {parent, non-exact}
  - isSubtypeOf(P, T) => true
  - !canTypesIntersect(P, T) => false
- If <parent> doesn't have subclasses => replace with an exact class ID check

AZUL SYSTEMS®

Agenda

- Abstractions
- Java Type Framework
- Exploiting Java Types
  - Java Specific Optimizations
  - Existing Optimizations

AZUL SYSTEMS®

Alias Analysis

- LLVM's Type Based Alias Analysis
  - Optimizations like inlining, CFG simplification don't make TBAA more accurate
  - Dropped like any other metadata
- JavaType framework has the same information
  - Benefits from more sophisticated analysis and healing
  - JavaTypes are refined during optimizations

AZUL SYSTEMS®

JavaType Based AA

Pointers don’t alias if base object types can’t intersect

AZUL SYSTEMS®

Context-Sensitive AA

- Want to make use of context-sensitive type sharpening
- Can't make context sensitive queries in AA
  - Introduced a new metadata - base_object_java_type
  - Updated by InstCombine

AZUL SYSTEMS®

Base Object Java Type Example

```
  %cid = call i32 @get_klass_id(i8 addrspace(1)* %object)
  %cond = icmp eq i32 %cid, 42
  br i1 %cond, label %match, label %mismatch

match:
  %addr = getelementptr i8, i8 addrspace(1)* %object, i64 20
  %addr.typed = bitcast i8 addrspace(1)* %addr to i64 addrspace(1)*
  %field = load i64, i64 addrspace(1)* %addr.typed
```

AZUL SYSTEMS®

Base Object Java Type Example

```
  %cid = call i32 @get_klass_id(i8 addrspace(1)* %object)
  %cond = icmp eq i32 %cid, 42
  br i1 %cond, label %match, label %mismatch

match:
  %addr = getelementptr i8, i8 addrspace(1)* %object, i64 20
  %addr.typed = bitcast i8 addrspace(1)* %addr to i64 addrspace(1)*
  %field = load i64, i64 addrspace(1)* %addr.typed,
                  !base-object-java-type !{i32 42, i1 true}
```

Attached by InstCombine

AZUL SYSTEMS®

Dereferenceability

- Expressed using dereferenceable/ derferenceable_or_null attributes and metadata
- JavaTypes is a more accurate way to derive dereferenceability information
- It benefits from more sophisticated analysis and metadata healing
  - Handles control flow merges
  - Type sharpening

AZUL SYSTEMS®

Inline Cost

- InlineCost is taught about JavaType based optimizations
  - InstCombine maintains JavaTypes for the arguments on call sites
  - InlineCost uses argument types to estimate the effect of potential optimizations

AZUL SYSTEMS®

Inline Cost

- Constant folding of get_class_id for known argument types

```
if (get_class_id(%arg) == expected_class_id)
  inlined_target_1
else if (get_class_id(%arg) == expected_class_id_2)
  inlined_target_2
```

- Bonus for call sites devirtualizable after inlining

```
%target = resolve_virtual(%arg, i32 <vtable_index>)
call %target
```


AZUL SYSTEMS®

Future Work

- JavaType analysis pass
    - Caching the results
- Improve the type system
    - Multiple inheritance of interfaces, array types
- Upstream generalised type framework?
    - Do you need a similar functionality for your language?

AZUL SYSTEMS®

Conclusion

- Express Java-specific semantics using high-level embedded IR
  - Very flexible and low-cost representation
- Introduced few Java specific optimizations
- Heavily rely on the existing LLVM optimizations
  - Made existing optimizations benefit from new information

AZUL SYSTEMS®

Questions?