Classless Closures for a Small Embedded VM

Oleg Pliss
Java ME Embedded
Internet of Things
Agenda

• Introduction to Monty JVM
• Closures in Java
• Classless implementation of closures
• Q & A
Introduction to Monty JVM
CLDC HI VM Overview

- Connected Limited Device Configuration JVM
  - Build-time choice of CLDC 1.0, 1.1, 1.1.1 and 8 profiles
  - First release: October, 2003
- HotSpot Implementation (optional)
  - Profiler-driven dynamic dynamic compilation
  - Optimistic speculative optimizations
  - Dynamic deoptimization (when necessary)
- Targeted to small mobile & embedded devices
  - Slow processor and memory
  - Constrained memory (16K-16M RAM)
  - May not have a fully capable OS
    - Single process
    - Single native thread
    - May or may not support page protection and memory-mapped files
    - May have no OS (bare metal)
Target processors

• ARM 9, 11 with optional coprocessors and instruction set extensions
  – Thumb, Thumb 2
  – JazelleDBX (HW bytecode interpreter)
  – ARM VFP (Vector Floating Point coprocessor)
• ARM Cortex A, M3, M4
• Intel x386+
  – For debugging and cross-compilation
• SuperHitachi SH3, SH4
• SPARC
  – For cross-compilation only
Evolution of CLDC profile

- **CLDC 1.0, 1.1, 1.1.1**
  - Subsets of J2SE (JDK 1.3)
  - No user-defined class loaders
  - No reflection (except for `Class.forName`)
  - No serialization
  - No JNI and native code in applications
  - No user-defined finalizers
  - Requires 32K RAM, 160K ROM for VM and class library

- **CLDC 8**
  - Subset of Java SE 8, released April 2014
  - Supports new language features (Generics, Annotations...)
  - Retains all limitations of the older CLDC profiles
  - No `invokedynamic`
  - No annotations with RUNTIME retention policy
  - Requires 128K RAM, 512K ROM for VM and class library
VM technologies under hood

- Manually optimized assembly interpreter
  - Most of the code is interpreted due to the lack of memory
  - Can use h/w acceleration
  - Execution stacks are elastic and allocated in the object heap
    - Grow and shrink when necessary
  - Can be easily extended with new internal bytecodes
    - But not many spare bytecodes left

- (Almost) Everything is a runtime object
  - But not necessarily Java object
    - Method, Compiled method, Execution stack...
    - Any runtime object could be made Java object
  - New kind of runtime objects can be easily defined
    - `object_size(obj)` – compute the object size
    - `oops_do(obj, func)` – apply a function to every reference field
    - `print_on(stream)` – pretty-printing for convenience of debugging (optional)
    - Statically register the kind and get the kind_id
VM technologies under hood (1)

- Single-pass dynamic adaptive compiler (optional)
  - Pauseless incremental schedulable compilation
  - Driven by a dynamic profiler
    - Combined sampling and instrumentation
  - Compiled code and temporary data allocated in a distinguished area of the heap
    - Relocatable and resizable
    - Execution of compiled code is profiled
    - “Cold” code is evicted, no GC is necessary
- No IR constructed: direct abstract interpretation of bytecodes
- Optimizations:
  - Constant folding
  - Type, constant and copy propagation
  - CSE (with a dictionary of bytecode strings)
  - Null check and checkcast elimination
  - Limited-depth inlining of method calls
  - Speculative devirtualization (unguarded)
  - Loop and branch optimizations
VM technologies under hood (2)

- System class pre-linking (*ROMization*)
  - System libraries and pre-installed applications loaded at build time
  - The classes are selectively initialized
  - Aggressively optimized for size and speed
    - Open- and closed-world models supported
    - Reduction of interface and virtual method calls
    - Elimination of unreachable methods, fields and classes
    - Selective AOT compilation
    - Symbolic information stripped
    - Constant pools are merged
    - Immutable data separated, stored in ROM, shared between isolates
  - The generated image is compiled & linked into VM executable
  - Reduces static and dynamic footprint
  - Greatly reduces VM start-up time
VM technologies under hood (3)

- Generational mark & compact garbage collector
  - Heap occupancy > 80%
  - Linear allocation, sliding window compaction
  - Preserves allocation order
    - Improves locality
    - Helps to eliminate cross-references in persistent groups of objects

- Multiple virtual threads over single native thread

- Multitasking within single native process (Isolates)
  - With task priorities, resource quotas and shared libraries
  - Synchronous native finalization on isolate termination

- Lightweight native interface
  - Direct access to Java objects via the generated C++ structures
  - KNI
Bytecode Quickening

• Interpreter rewrites some bytecodes during execution
  – When necessary, a method can be quickened by request

• Resolve symbolic references

• Validate the semantics

• If successful, patch the bytecode with a quicker version
  – To avoid repeated quickening of the same bytecode
  – Bytecode size has to remain the same
    • Can be padded with *nop's*

\[
\text{quick\_getstatic, quick\_<T>getfield, quick\_invokevirtual, quick\_invokeinvirtual\_final, quick\_invokespecial, quick\_invokeinterface, quick\_instanceof, quick\_checkcast}
\]

• Frequently used sequences of bytecodes can be replaced with faster *super-instructions*
  – i.e. *aload\_0\_fast\_agetfield\_1*
Closures in Java
Closures

- **Closure** is a function (or reference to a function) together with the environment referenced by the function
  - Introduced in *Scheme* programming language (1975-80)
- In stateful programming language a function can modify its environment
  - Block in *Smalltalk* and *Self*
  - Activation record in *Beta*
  - Locals of outer scopes can be modified
    ```
    |count incrementCount|
    count := 0.
    incrementCount := [count := count + 1].
    1 to: 10 do: [:i | i even ifTrue: incrementCount].
    ^count
    ```
- Closures in Java can be modeled with inner classes and lambda expressions (in Java 8+)
  - Non-local variables are captured and cannot be modified
Lambda expressions in Java 8+

• Lambda expression produces an instance of functional interface
  – Essentially an interface with a single abstract method
  – Notional interface can be induced by an intersection type
    FunctionalInterface & MarkerInterface(s)
  – The JLS 8 carefully avoids unnecessary restrictions on the implementation of the interface
  – Usually local class implementing the functional interface is created

• Lambda expression is compiled to:
  – the method representing the lambda body
  – invokedynamic for a method of java.lang.invoke.LambdaMetafactory

• Can we use lambda expressions in CLDC?
  – They are convenient and expressive
  – But there is no invokedynamic and java.lang.invoke package
  – Invokedynamic for LambdaMetafactory can be treated as an idiom
Example: Inner class implementing a functional interface and capturing one value

```java
interface MyFunction {
    int my_function();
}

class OuterClass {
    int x;

    class MyClass implements MyFunction {
        int my_function() {
            return x;
        }
    }
}
```
Internals of MyClass implementing MyFunction and capturing one value

Class MyClass

- Java mirror
- header
  - size
  - field_map
  - class_info
  - superclass
  - array_class
  - prototypical_proxy
  - supertype_cache_0
  - supertype_cache_1
  - hash_next
  - instance_field_map

No static fields

Overhead: at least 50 words (200 bytes)
Classless Implementation of Closures
Classless closure for MyFunction

<table>
<thead>
<tr>
<th>header</th>
<th>com.sun.cldchi.jvm.SimpleClosure</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>my_method</td>
</tr>
<tr>
<td>MyFunction_id</td>
<td>args_oop_map</td>
</tr>
<tr>
<td>captured values</td>
<td></td>
</tr>
</tbody>
</table>

- **SimpleClosure**
  - Hidden abstract instance class, extends Object
  - May (but not required to) override inherited methods (equals, hashCode, toString)
- **Method**
  - A reference to a method of any class, must have compatible type
- **MyFunction_id**
  - Every class and interface has unique id
  - Max 16K classes per Isolate: fits in 14 bits
- **args_oop_map**
  - Number of captured words and a bitmap of pointers among them
    - 4 bits for size + 14 for bitmap, or 17 for bitmap + 1 for the terminator bit
Relaxed type compatibility for fully quickened methods

• Quick bytecodes are fully resolved
  – Symbolic references are replaced by addresses, offsets and indices
  – Access rights are already validated during the quickening

• Method is fully quickened if:
  – Contains only quick versions of bytecodes
  – Is invoked only by quick bytecodes

• Type compatibility of static and virtual methods
  – SomeClass.static_method(SomeClass receiver, args) and
    SomeClass.virtual_method(args) are type-compatible
  – Only total number of arguments, their order and types are important

• Mobility of static methods
  – Fully quickened static method of one class can be moved to any
    other class while all references to the method preserved
Invocation of Simple Closures

• Polymorphism of functional interfaces
  – Java type system cannot distinguish regular Java class and SimpleClosure implementations of the same interface
    • Type system of dynamic/AOT compiler can be richer – it can be able to make it for some call site at compile time
  – The same code must work with both representations
  – Bytecodes may need to handle the difference in run time

• Four bytecodes require modification
  – quick_invokespecial
  – quick_invokeinterface
  – quick_isinstanceof
  – quick_checkcast

• No need to modify quick_invokevirtual
  – For final classes invokevirtual is always quickened to quick_invokevirtual_final
  – Calls a resolved reference to the method in the constant pool
Modification of quick_invokespecial

- `quick_invokespecial <method_index> (receiver ...)`

  ```java
  Method method;
  if (receiver.class == SimpleClosure) {
      method = ((SimpleClosure)receiver).method;
  } else {
      // Regular Java class
      method = receiver.classinfo.get_virtual_method(method_index);
  }
  invoke(method);
  ```
Modification of quick_invokeinterface

- quick_invokeinterface <interface_id, method_index, n_args> (receiver ...)

    Method method;
    if (receiver.class == SimpleClosure) {
        method = ((SimpleClosure)receiver).method;
    } else {
        // Regular Java class
        const ClassInfo classinfo = receiver.classinfo;
        method = classinfo.lookup_interface_table(interface_id)
            [method_index];
    }
    invoke(method);
Modification of quick_isinstanceof

- quick_isinstanceof <class_id> (obj)
  
  ```java
  Class klass = obj.class;
  if (klass == SimpleClosure) {
      // Lookup the superclasses of SimpleClosure
      // Object is the only accessible superclass of SimpleClosure
      if (class_id == Object_id) {
          return true;
      }
      klass = get_class_by_id(((SimpleClosure)obj).interface_id);
  }
  return klass.is_subtype_of(class_id);
  ```

- quick_checkcast <class_id> (obj)
  - Is similar but throws an exception instead of returning a boolean
Creation of SimpleClosures

• New internal bytecode
  new_simple_closure<interface_id, method, args_map>
  – Args_map combines n_args and oop_map in a short value
  – Creates new SimpleClosure for n_args of captured values
  – Initializes interface_id and method fields
  – Pops a block of n_args words from stack to the captured fields
    • Plain data copy (could use memcpy) followed by the adjustment of SP
    • No need in write barrier – it is an initialization of a young object
    • Type correctness must be guaranteed by bytecode construction

• Do we have enough space at the capture site to generate this bytecode without the method expansion?
  – For the old Java the answer is positive
  – For the new Java it is negative
  • Have to move interface_id from the bytecode to the cpool or the method:
    new_simple_closure<method_cp_index, args_map>
Creating a Simple Closure in Old Java

```java
new <MyClass> ; 3 bytes
dup ; 1 byte
aload_0 ; N captured values (N >= 0)

... invokespecial <MyClass.<init>> ; 3 bytes

aload_0 ; N captured values

... new_simple_closure <MyFunction_id, method, N> ; 7 bytes:
  ; interface_id - immediate 2 bytes
  ; method - constant pool index to resolved method, 2 bytes
  ; args_map - immediate 2 bytes
```

- Exception table may need to be updated
  - Bytecode indices have changed
Creating a Simple Closure in New Java

```java
aload_0                             ; N captured values
...
invokedynamic <call site specifier> ; 5 bytes

aload_0                             ; N captured values
...
new_simple_closure <method, N> ; 5 bytes:
; method – constant pool index to the pair of indices
; (resolved_static_method, class_id)
; n_args_and_oopmap – immediate 2 bytes

• Constant pool entries can be shared between the bytecodes
  – It may be easier to store interface_id somewhere in the method
    and to share just a ResolvedStaticMethod entry between equivalent call sites
```
Simple Closure conversion for old Java

• Class implementing the interface is created statically
  – It contains a virtual method implementing the interface method

• Check if the class can be converted to Simple Closure

• Adjust the reads of captured values in the method
  – Offset of captured fields is different in Simple Closure (3 words) and the original Java class (1 word)
  – Java stack may grow downwards
    • It is easier to reorder fields once than to modify code generators for all supported ISAs
    – Adjust field offset in the instance field descriptors before quickening the method

• Convert the signature to the static method

• Move the method to the closest outer Java class

• Dispose the implementing class
When an instance class can be converted to Simple Closure

- Final, extends Object, implements single functional interface
- Contains no fields or methods except for the implemented functional method and the constructor
- The constructor initializes every field by the respective argument
- Contains only resolvable symbolic references
  - And so can be fully quickened
- Referenced only at capture site to create a closure
  - Anonymous class is just a class with mangled name
  - The enclosing scope of inner class definition is lost during compilation to the class file
    - Closed syntactic scope within a method or a class is expanded to the package
    - A loaded later class can refer to the any other class in its package
  - We can guess but cannot really prove the class is properly used
Simple Closure conversion for new Java

• Lambda body is represented by a method of the enclosing class
  – All captured values are passed as arguments
• Can the interpreter push a block of previously captured values and call this method?
  
  closure lambda_args --> closure lambda_args captured_values
  – Unfortunately, no: the number and the order of arguments differ
  – … and their types can be different too
• Adapter method has to be generated
  – Let's make it a static method in the same class as the lambda body
  – LambdaMetafactory has to be partially re-implemented in the runtime
  – The generation of the adapter may require boxing/unboxing and widening conversions of the arguments and the result
Transparency of Simple Closures

• Is there an observable difference between Simple Closure and anonymous internal Java class implementing the same interface?
  – closure.getClass() returns a different class
    • It is the same for all simple closures, and this is observable
  – SimpleClosure cannot have the same properties as the respective internal class
    • closure.getClass().newInstance() never creates a closure, throws an exception
    • MyFunction.isAssignableFrom(SimpleClosure) returns false regardless of the interface implemented by its instances
      – Not a part of older CLDC profiles
      – Specified in CLDC 8 but never used anywhere in the libraries
  – If required, closure.getClass() could create a fake Java class lazily
    • Must be shared between all Simple Closures created by the same capture site
      – class_id has to be allocated eagerly
    • Must be in correct relation with its instances and the implemented interface
    • Disposable when not referenced
Which Java is better for Simple Closures

- Ideally, there should be one statically generated method
  - Captured values can be represented as fields or the tail arguments
  - The head arguments must be compatible with the interface

- Both the old and the new Java deviate from the ideal

- The old Java:
  - Generates a method with the matching arguments
  - … but it is located in a wrong package-private class
  - All references to the class have to be analyzed
    - The analysis would be easier if the class could be local within a method
  - Bytecodes of the capture site must be analyzed and rewritten

- The new Java:
  - Generates a method in the proper class but with wrong arguments
  - An additional adapter method must be generated by VM
    - Memory and performance overhead
    - Bytecode generator does not naturally belong to this VM
Bridge methods and marker interfaces

• Bridge methods are artifacts of generics in Java type system
  – Do not naturally belong to closures or lambda expressions

• Marker interfaces is a value add-on for lambdas
  – Do not naturally belong to closures or lambda expressions
  – Memory overhead
  – Performance overhead in current implementation of `invokeinterface`, `instanceof` and `checkcast` bytecodes
    • The interface table lookup is linear on the number of implemented interfaces regardless of the number of the methods

• Hopefully, they can be omitted in CLDC subset
• … But what if we had to implement them anyway?
Simple Closures with bridge methods

- Fields `interface_id` and `method` moved from Simple Closure to Static Call Site
- Field `size/oop_map` is the same Simple Closures created by one Static Calls Site. But it cannot be moved to Static Call Site: it defines object size and so must be accessible via no more than one hop from object header.
- A bit more complicated implementation of bytecodes `quick_invokeinterface`, `quick_invokespecial`, `quick_instanceof` and `quick_checkcast`
Simple Closures with bridge methods and marker interfaces

- A bit more complicated implementation of bytecodes `quick_instanceof` and `quick_checkcast`
  - Extra 1-2 words for supertype cache could improve the performance
- Getting closer to regular classes... Are they really so terrible?
Alternative Approach

- Adapters can be generated by the runtime-specific external convertor
  - Standard class file format can be used
  - The change can be encoded by a different method of LambdaMetafactory in the call site descriptor
  - The method may not exist – it is just an idiom for the runtime
Q&A