Case Study: Implementing a Java JIT Compiler in Haskell

Bernhard Urban
Johannes Kepler University, Austria
bernhard.urban@jku.at

Harald Steinlechner
Vienna University of Technology
haraldsteinlechner@gmail.com

Abstract

We present a JVM prototype implemented in the purely-functional language Haskell. It exploits several features of the language, such as strong static typing to implement an intermediate representation, and abstraction mechanism to express machine code generation in the manner of a domain specific language.

The compiler consists of (i) a pass to transform Java bytecode to a register-based intermediate representation, (ii) application of an existing data-flow analysis framework to our intermediate representation and (iii) machine code generation that targets the `x86` architecture. The implementation follows a compile-only approach. To implement certain Java features efficiently, code patching is used.

Various code samples demonstrate the elegance of our prototype. Results prove reasonable performance compared to real-world implementations.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers

General Terms Algorithms, Languages, Performance

Keywords Java, Haskell, virtual machine, intermediate representation, data-flow analysis, code generation

1. Introduction

The inherent complexity of compilers is increased in the context of dynamic compilation. The latter is of increasing relevance due to the rise of dynamic languages. For a high performance implementation of a language, compiler writers traditionally use low-level languages such as C and C++ which have mature and highly tuned tool chains and allow low level features such as raw memory access. More recent research work shows a trend towards adopting higher level implementation languages. For example, MaxineVM [8] is implemented in Java.

Functional languages provide other higher level design patterns. For example *monads* can be employed to model problems such as *monadic parsing* [3] more elegantly than imperative implementations. In our case study we use monads in different parts of the compiler, e.g. to express *imperative* algorithms in a functional setting or as abstraction mechanism for code generation.

Of course the high level of abstraction found in languages such as Haskell comes with a price, namely performance. Although Haskell has been shown to be faster than C in some areas [5], this is usually not the case, and it is usually considered to be some factors slower.

In this paper we present present an overview of our prototype, which consists of a Java bytecode compiler (main focus of this paper) and a run-time system, mostly implemented in Haskell with a small amount (about 100 lines) in C for signal handling.

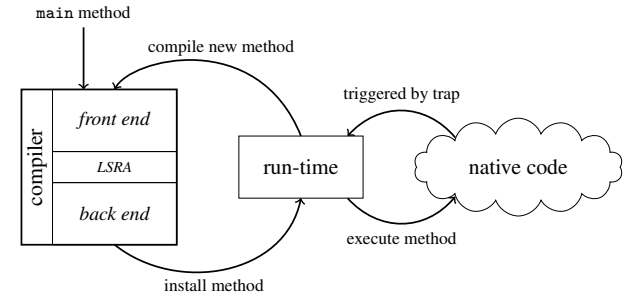

Figure 1. Execution flow in MateVM

2. Overview of MateVM

Our prototype is called *MateVM* and is available on GitHub[1] licensed under the GPL3.

A concise overview is depicted in Figure 1. The VM follows the compile-only approach. At startup, the VM finds the `main`-method and compiles it to machine code. The generated code is then registered and executed by the run-time system. Assume that the `main`-method calls another method `foo`. Instead of compiling `foo` at VM startup, the call to `foo` is compiled as a trap in `main`.

When this call is executed, the trap is triggered and handled by the run-time system. The run-time-system retrieves code for `foo`, compiling it first if necessary, and patches the trap with a call to the code. Execution is resumed at the patched call instruction.

The compiler is partitioned into a front end (Section 3), a register allocator (linear scan [6]) which constitutes a transition to the machine dependent part and a back end, for generating machine code (Section 4). The trap mechanism is presented in Section 4.1.

3. Front End

The `hs-java`[2] library is used to parse Java class files into a Haskell idiomatic data representation. It supports class path handling and is able to read class files from JAR files. The compiler is supplied with the instruction stream and a reference to the constant pool.

[1] `https://github.com/MateVM/MateVM`

[2] `http://hackage.haskell.org/package/hs-java`

```
data MateIR t e x where
 IRLabel :: Label -> HandlerMap -> MaybeHandler -> MateIR t C O

 -- dst <- src1 `op` src2
 IROp     :: OpType -> t -> t -> t ->              MateIR t O O
 IRStore  :: RTPool t -> t -> t ->                 MateIR t O O
 IRLoad   :: RTPool t -> t -> t ->                 MateIR t O O
 IRMisc1  :: Instruction -> t ->                   MateIR t O O
 IRMisc2  :: Instruction -> t -> t ->              MateIR t O O
 IRPrep   :: CallingConv -> [(t, VarType)] ->      MateIR t O O
 IRInvoke :: RTPool t -> Maybe t -> CallType ->    MateIR t O O
 IRPush   :: Word8 -> t ->                         MateIR t O O

 IRJump      :: Label ->                           MateIR t O C
 IRIfElse    :: CMP -> t -> t -> Label -> Label -> MateIR t O C
 IRExHandler :: [Label] ->                         MateIR t O C
 IRSwitch    :: t -> [(Maybe Int32, Label)] ->     MateIR t O C
 IRReturn    :: Maybe t ->                         MateIR t O C
```

Figure 2. `MateIR`, intermediate representation of the compiler

The first step transforms Java bytecode into the internal intermediate representation of the compiler, named *`MateIR`* and depicted in Figure 2.

For the definition of `MateIR`, the Haskell type extension *Generalized Algebraic Data Types* (GADTs) [4] is used. This kind of definition enables us to use Hoopl [7] for the intermediate representation, which is a library for data-flow analysis. It provides functions to build and process graphs, and expresses passes as either forward or backward by providing (i) a lattice, (ii) a transfer function and (iii) a rewrite function. Graphs built with Hoopl are typical control-flow graphs composed of basic blocks. A basic block is a sequence of nodes, that are represented by `MateIR` in our compiler. An example of Hoopl is presented in Section 3.2.

Consider the first line of the definition in Figure 2 which contains three type variables: `t`, `e` and `x`. The first type variable `t` is a normal type variable and represents a register type. That is, `MateIR` is *polymorphic* with respect to `t`. However `e` and `x` are interpreted differently. They are not mentioned anywhere in the constructor definitions. Instead, `O`'s and `C`'s are used and they are defined as follows:

```
data O = O -- open
data C = C -- closed
```

Each data type above has exactly one constructor and therefore one value. These so-called *phantom types* are used to express invariants using ordinary types. Those invariants are checked by the type checker at compile-time.

The names `e` and `x` are abbreviations for ***e**ntry* and ***ex**it*. Therefore, `e` and `x` denote if a single instruction is *open* or *closed* on entry and exit—this is the *shape* of an instruction[3].

Building on this, we derive the notion of basic blocks: An `IRLabel` is the *entry instruction* of a basic block. We say, `IRLabel` is *closed* on entry and *open* on exit. That is, the instruction can have several predecessors but has *exactly one* successor. Similar, `IROp` has exactly one predecessor and exactly one successor. `IRSwitch` in contrast, has exactly one predecessor and can have several successors—it denotes the end of a basic block.

Having such invariants modeled by the type system, allows us to write functions which expect arguments to be in a specific shape, where the compiler proves correct usage at compile-time. Consider the following signature of a function from the Hoopl-library:

```
(<*>) :: (GraphRep g, NonLocal n) => g n e O -> g n O x -> g n e x
```

where `g` is a type variable, restricted to be an instance of `GraphRep` (think of an interface for a graph data structure). In our example,

[3] the notion of shapes also applies to blocks and graphs.

```
class NonLocal ins where -- defined by Hoopl
  entryLabel :: ins C x -> Label
  successors :: ins e C -> [Label]

instance NonLocal (MateIR Var) where
  entryLabel (IRLabel l _ _) = l
  successors (IRJump l) = [l]
  successors (IRIfElse _ _ _ l1 l2) = [l1, l2]
  successors (IRExHandler t) = t
  successors (IRSwitch _ t) = map snd t
  successors (IRReturn _) = []
```

Figure 3. Definition of `NonLocal` for `MateIR`

`n` would be `MateIR`. See Figure 3 for an instance definition of `NonLocal`.

The function `(<*>)` is used to connect two graphs, with the following restriction: the first graph must be *open* on the exit and the second graph must be *open* on the entry. This allows a new graph to be *safely* constructed while preserving the shape of the original entry `e` of the first graph, and exit `x` of the second graph.

In order to make the graph iterable for Hoopl, it has to determine what the predecessor and successor of a `MateIR`-instruction are. This is solved by a type class called `NonLocal` (see Figure 3) which has to be implemented for `MateIR`. Note the elegance of GADTs here: The type system already knows that `IRLabel` is the only instruction which requires an implementation for `entryLabel`, as it is the only constructor which is *closed* on the entry.

Consider the first type variable `t` again: This is the place holder for a type describing an actual variable or register. First, `MateIR` is equipped with virtual registers. A virtual register is an integer annotated with its computational type or a constant of the corresponding type.

As a result of register allocation, virtual registers are replaced by hardware registers of the target CPU architecture. The graph representation is preserved by this transformation. That is, we want a function that has the type signature:

```
registerAllocation :: Graph (MateIR VReg) e x
                   -> Graph (MateIR HardwareReg) e x
```

However, it is not straight forward to apply a linear scan register allocation algorithm to a graph. As such, the graph is flattened to a linear representation prior to register allocation.

3.1 Java bytecode to `MateIR`

In order to build a list of `MateIR`-instructions for a Java method, the bytecode for the method is read from a class file. To identify basic blocks, two passes are needed over the Java bytecode stream:

1. find all jump targets in the code. This "preparation" pass is necessary to resolve backward references, e.g. loops. Also, exception handlers and try blocks are marked as block boundaries in this pass.
2. building basic blocks by translating Java bytecode instructions to `MateIR` instructions. Since block boundaries are produced by the first pass, basic blocks can be easily determined.

`MateIR` is a register based representation, therefore stack temporaries have to be mapped to virtual registers. Furthermore, some Java bytecode instructions are not typed. Both problems are tackled in the second pass: Types are determined by abstract interpretation of the Java stack and new virtual registers are created during this process.

3.2 Implementing Liveness Analysis with Hoopl

After generating a graph from Java bytecode, we apply data-flow passes with the help of Hoopl. The example given in Figure 4 implements a transfer function for a backward pass that computes

```
type LiveSet = Set VirtualReg

factLabel :: FactBase LiveSet -> Label -> LiveSet
factLabel f l = fromMaybe Set.empty (lookupFact l f)

varsIR' :: forall e x. MateIR Var e x -> ([Var], [Var])
varsIR' ins = (defIR ins, useIR ins)

livenessTransfer :: BwdTransfer (MateIR Var) LiveSet
livenessTransfer = mkBTransfer3 liveCO liveOO liveOC
  where
    liveCO ins f = factMerge f (varsIR' ins)
    liveOO ins f = factMerge f (varsIR' ins)
    liveOC ins f = factMerge facts (varsIR' ins)
      where
        facts = foldl Set.union Set.empty
                (map (factLabel f) (successors ins))

    addVar :: Var -> LiveSet -> LiveSet
    addVar (VReg v) f = Set.insert v f
    addVar _ f = f

    removeVar :: Var -> LiveSet -> LiveSet
    removeVar (VReg v) f = Set.delete v f
    removeVar _ f = f

    factMerge :: LiveSet -> ([Var], [Var]) -> LiveSet
    factMerge f (defs, uses) =
                foldr removeVar (foldr addVar f uses) defs
```

Figure 4. Transfer function for the liveness pass

```
emitOO :: MateIR HVarX86 O O -> CodeGen e CompileState ()
emitOO (IRLoad (RTPool cpidx) src dst) = do
  cls <- classfile <$> getState
  case constPool cls M.! cpidx of
    (CField desc) -> do -- GETFIELD
      offset <- liftIO (getFieldOffset cls desc)
      mov dst (Disp32 offset, src)
    -- handle other cases

getFieldOffset :: Class Direct -> FieldDescription -> IO Word32
```

Figure 5. Example of the DSL provided by Harpy

liveness information of variables. This liveness information is used as a basis for (i) computing live ranges during register allocation, (ii) dead-code elimination and (iii) safepoints for garbage collection.

4. Back End

For code generation, the Haskell library *Harpy* is used [2]. Harpy exposes the so-called `CodeGen`-monad, which enables us to express `x86` instructions in a style of a domain specific language similar to the Intel syntax. The example in Figure 5 generates machine code for the instruction `IRLoad`, originally created from a `GETFIELD` bytecode instruction, that has the object reference in the register `src` and stores the value of the read in `dst`.

Consider the type signature in the first line: The function requires a `MateIR` instruction instrumented with a hardware register. The return type of the function is the result of a sequence of actions, encapsulated in a *monad*. Harpy provides a specialized monad for code generation, `CodeGen`. In this example it is parameterized with an environment (here the type variable `e`), a state (`CompileState`) and a return value `()`, which has a similar meaning as `void` in Java.

In the second line, *pattern-matching* is used to make a case distinction and to unpack the class index for the corresponding field.

In the third line the local state is consulted, which contains a reference to the class file. With that, the entry of the constant pool can be accessed. The `CField` value contains a description of the field. `getFieldOffset` asks the run-time system for the field's

```
girStatic :: Word16 -> Maybe HVarX86
          -> CallType -> PreGCPoint HVarX86
          -> CodeGen e CompileState ()
girStatic cpidx haveReturn ct mapping = do
  cls <- classf <$> getState
  let l = buildMethodID cls cpidx
  newNamedLabel (show l) >>= defineLabel
  -- emits the sequence 0xff 0xff 0x90 0x90 0x90
  calladdr <- emitSigIllTrap 5
  let patcher :: WriteBackRegs
              -> CodeGen () () WriteBackRegs
      patcher wbr = do
        entryAddr <- liftIO $ lookupMethodEntry l
        let relative = entryAddr - ((wbr M.! eip) + 5)
        call $ fromIntegral relative
        return wbr
  setGCPoint mapping
  let mname  = methodNameTypeByIdx cls cpidx
  let argcnt = methodGetArgsCount mname * ptrSize
  when (argcnt > 0) (add esp argcnt)

  case haveReturn of
    Just (HIReg dst) -> mov dst eax
    Nothing -> return ()
  let patchEntry = StaticMethod patcher
  modifyState (\s -> s
     {traps = M.insert calladdr patchEntry (traps s)})
```

Figure 6. Code patching via `CodeGen`-monad

offset—`liftIO` encapsulates an expression with side-effects (e.g. class loading). Expressions with side-effects have to be *explicitly* annotated, otherwise the Haskell compiler will not accept it as valid program since the types do not match.

Finally, the actual operation can be emitted: Storing the result of a memory access into a register. Note, that the `mov`-instruction will be written into memory as-is. For example, assuming `src = eax`, `dst = ebx` and `offset = 0x20`, this would be:

```
mov eax, [ebx+0x20]
```

The resulting representation after register allocation is a linearized graph of `MateIR` instructions, already in terms of hardware registers. At that point, native code is generated for each `MateIR` instruction in a manner similar to the above example. We use other type system extensions, such as `RankNTypes` in order to group instructions for the `x86` architecture with certain properties together to avoid code duplication.

4.1 Trapping Code and Code Patching

The code generator emits traps at points where run-time system assistance is required. Traps are registered with the back end in terms of the program counters where they can occur. The two kinds of traps are (i) one-time traps such as for unresolved calls and (ii) persistent traps such as `ATHROW` which always traps into the run-time system to allocate the exception object and execute the exception handling logic.

During patching we do not want to deal with architecture specific details. Instead, the back end should provide details about how to patch code.

In the current implementation, the `CodeGen`-monad is used a second time, namely at patch-time. Consider the example in Figure 6: `girStatic` emits code for a static call. In order to get the entry address of the callee, the run-time system must be queried. However, a query could have side-effects such as compilation of the callee. Therefore, in line 9 a sequence is generated that eventually triggers `SIGILL`. In line 10 a *closure*[4] named `patcher` is defined, that is also a `CodeGen`-monad.

`patcher` describes the logic needed to resolve the call-site, which is to (i) request the entry point of the callee (annotated

[4] i.e. a function that can access variables of its lexically enclosing scope.

with `liftIO` to explicitly express that the operation can have side-effects) and (ii) replace the trap with the actual `call` instruction. `patcher` uses local information such as `cls`, the class reference, for the latter.

The unevaluated closure is stored in the trap map (line 27). When a trap occurs at execution time, the trap handler uses the current instruction pointer to get the responsible closure for handling the trap. It calls the closure with relevant context information, such as the current values in the hardware registers (`WriteBackRegs`).

Catching of traps is implemented with a small amount of C code, as the GHC runtime system is able to catch signals but does not provide context information such as the register state of the machine.

4.2 Bridging the gap to bare-metal hardware

Except for signal handling, the presented system is implemented in pure Haskell. The Glasgow Haskell Compiler (GHC) compiles a Haskell application to a standard binary. Furthermore, GHC has a well-defined *foreign function interface* (FFI), enabling interoperability with non-Haskell code e.g. C.

Usually, FFI is used to link functions at compile-time. However, this obviously is not possible for a JIT compiler. Thus, we use the *dynamic import* statement provided by the GHC FFI API [1]. This mechanism is similar to function pointers in C and can be used to call to arbitrary memory blobs containing machine code.

We use `cdecl` as our calling convention, which obviates the need for stubs to marshal arguments and return values across native calls. An additional advantage is that the stack of the Haskell runtime can be used without any modification.

5. Results

Due to the lack of feature-completeness (e.g. floating point or multi threading) we cannot run sophisticated benchmarks such as SpecJVM or DaCapo. Instead, we created some (micro)benchmarks, which stress implemented features of MateVM. The source code for these benchmarks is included in our repository.

The wall time is shown in Table 1, where `HelloWorld` is used as a baseline; the time needed for `HelloWorld` is subtracted from each result in order to have better comparison of the actual time spent, ignoring startup-time needed for the base library.

Unsurprisingly, MateVM does not (yet!) outperform other just-in-time compilers. The `Virtual` and `Interface` benchmarks suffer from a suboptimal layout for tables used for dispatching `virtual` and `interface` calls. The simple layout currently used requires more dereferencing during virtual dispatch. The `InstanceOf` benchmark shows the penalty we pay for making a run-time call via the trapping mechanism for every type check and the slow implementation of the type check itself.

The `Compiletime` benchmark feeds the compiler with a method that contains about 23000 Java bytecode instructions. MateVM spends about 80% of the time in the compiler. For comparison, CACAO spends about 50ms for compilation.

Considering the effort expended in implementing our system (about six man-months), we are encouraged to see that the generated code is only 2x–3x slower than that of CACAO.

6. Future Work

The compile-only approach with a single compiler makes aggressive optimizations challenging. Modern compilers use an interpreter or a baseline compiler as fallback if an optimistic assumption is invalidated. Given the relative simplicity of an interpreter, we consider two options for a mixed mode configuration: (i) interpret the graph-based `MateIR` or, (ii) use an existing interpreter such as JamVM[5]. The latter solution would require creating an interface between the interpreter (written in C) and our compiler. It would allow us to collect profile information during interpretation for use in the compilation process.

benchmark	`server`	`client`	`cacao`	`mate`	`jamvm`
HelloWorld	0.06	0.03	0.12	0.00	0.03
Fib	**0.15**	0.16	0.38	0.46	3.35
Objectfield	**0.02**	0.39	0.52	0.88	4.52
Staticfield	**0.02**	0.39	0.40	0.83	5.68
Virtual	**0.55**	0.65	2.02	4.97	25.33
Interface	**0.02**	0.12	0.24	0.65	3.37
InstanceOf	**0.00**	**0.00**	0.01	1.72	0.01
Array	0.85	**0.83**	0.89	1.59	5.70
Exception	0.24	**0.10**	0.19	0.43	0.45
Compiletime	0.14	0.14	0.20	0.94	**0.04**

Table 1. Measurements of execution time in seconds

7. Conclusion

We presented aspects of a JVM prototype implemented in Haskell. The compiler uses an intermediate representation that leverages features provided by the language (e.g. GADTs) in order to define the notion of basic blocks on a type-level. Hoopl enables adding passes to the compiler in the traditional form of a data-flow problem as shown by an implementation of liveness analysis.

The `monad`-type class proved to be useful for several tasks, e.g. to express an assembly-like domain specific language. The latter feels like writing assembler code with regular Haskell code in between.

Acknowledgments

We thank Andreas Krall for advising this project as part of some lecture exercises and master thesis. Thanks to Gilles Duboscq, Josef Eisl and Doug Simon for their valuable feedback on this paper.

References

[1] S. Finne, D. Leijen, E. Meijer, and S. Peyton Jones. Calling Hell from Heaven and Heaven from Hell. In *Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, ICFP '99, pages 114–125. ACM, 1999.

[2] M. Grabmüller and D. Kleeblatt. Harpy: Run-Time Code Generation in Haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, Haskell '07, pages 94–94. ACM, 2007.

[3] G. Hutton and E. Meijer. Monadic parsing in Haskell. *Journal of functional programming*, 8(04):437–444, 1998.

[4] S. P. Jones, G. Washburn, and S. Weirich. Wobbly Types: Type Inference for Generalised Algebraic Data Types. Technical report, 2004.

[5] G. Mainland, R. Leshchinskiy, S. Peyton Jones, and S. Marlow. Haskell beats C using generalized stream fusion. In submission, 2013.

[6] M. Poletto and V. Sarkar. Linear Scan Register Allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, 1999.

[7] N. Ramsey, J. a. Dias, and S. Peyton Jones. Hoopl: A Modular, Reusable Library for Dataflow Analysis and Transformation. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 121–134, New York, NY, USA, 2010. ACM.

[8] C. Wimmer, M. Haupt, M. L. Van De Vanter, M. Jordan, L. Daynès, and D. Simon. Maxine: An approachable Virtual Machine for, and in, Java. *ACM Transactions on Architecture and Code Optimization*, 9(4):30, 2013.

[5] `http://jamvm.sf.net/`