FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Implementation of a Java Just-In-Time Compiler in Haskell

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computer Architecture and Compiler Design

eingereicht von

Bernhard Urban
Matrikelnummer 0725771

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall

Wien, 31.1.2013

(Unterschrift Verfasserin) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Implementation of a Java Just-In-Time Compiler in Haskell

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Architecture and Compiler Design

by

Bernhard Urban
Registration Number 0725771

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall

Vienna, 31.1.2013

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Bernhard Urban
Mühlhofstraße 24/12, 3500 Krems

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasserin)

Acknowledgements

I first met the JVM specification in spring 2011 in the lecture “Abstract Machines”. In fall 2011, Josef Eisl suggested to build a Java virtual machine in Forth, as an exercise for the course “Stack-based languages” Although the idea was crazy, he insisted to do that and after some thoughts I was okay with that: JorthVM [ERU13] was born. During the implementation, the JVM specification and I went through a love–hate relationship. After the project, we were able to conclude that it is possible to build a JVM in Forth but it is hard[1], though we are glad we have done it.

Some weeks later (after the “Type Systems” exam) on some beers with Josef Eisl and Harald Steinlechner, the idea for MateVM was born. With the experience from JorthVM, the initial steps were a lot easier and many ideas were ported to MateVM, though being a just-in-time compiler.

Thus, many thanks to Josef Eisl and Harald Steinlechner for being motivating guys and always have an open ear to talk about VM stuff.

Thanks to Andreas Krall who enabled us to try new things and supported our project with his experience on virtual machines in general and CACAO in particular.

Also thanks to my family, friends and Iris for all the support and effort to make my study possible.

[1]of course, since Forth is not type safe! `;-)`

Abstract

Just-in-time compilation became popular lately, among others, due to the success of the Java programming language. In this thesis we successfully demonstrate a modern prototype of the Java virtual machine implemented as method-based just-in-time compiler in the purely functional programming language Haskell. Motivated by the powerful abstraction mechanism provided by the language, we were able to build a sophisticated compiler. Since a high-level language is used, we had to solve problems regarding the interface between Haskell and native code.

Existing libraries from the Haskell community are used. `hs-java` for processing Java class files. Hoopl to implement a sophisticated compiler, that supports data-flow analysis on an intermediate representation by making extensive use of the type system. Harpy is used to generate machine code for the target architecture `x86` in the manner of a domain specific language. We tried to keep as much as possible in pure code to gain the elegance of Haskell. The compilation pipeline, e.g. JVM stack elimination or linear scan register allocation, is presented in detail in this thesis.

Basic features of a run-time system in order to complete a JVM, such as method invocation, lazy class loading, dynamic type checking or exception handling, are presented.

Although implemented in Haskell and therefore at a high abstraction level, our prototype performs well. The performance is compared to mainstream JVMs.

Kurzfassung

Just-In-Time Übersetzung ist in der letzten Zeit populär geworden, unter anderem durch den Erfolg der Programmiersprache Java. In dieser Arbeit präsentieren wir einen modernen Prototypen einer Java virtuellen Maschine, implementiert als methodenbasierenden Just-In-Time Compiler in der rein funktionalen Programmiersprache Haskell. Motiviert durch die mächtigen Abstraktionsmöglichkeiten die Haskell bietet, waren wir in der Lage einen fortschrittlichen Compiler zu entwickeln. Probleme, die an der Schnittstelle zwischen Haskell und Maschinencode aufgetreten sind, galt es zu lösen.

Existierende Bibliotheken aus der Haskell Community werden verwendet.
`hs-java` wird zum Verarbeiten von Java Class Dateien genutzt. Der Compiler greift auf Hoopl zurück, das Datenflussanalyse auf einer Zwischendarstellung durch intensive Nutzung des Typsystems ermöglicht. Um Maschinencode für die Zielarchitektur `x86` zu erzeugen, wird Harpy eingesetzt. Diese Bibliothek ist als domänenspezifische Sprache implementiert. Die Compilierpipeline, zum Beispiel JVM Stack Eliminierung oder Linear Scan Register Allokation, werden detailiert in dieser Diplomarbeit vorgestellt.

Grundlegende Funktionen einer Laufzeitumgebung um eine JVM zu vervollständigen, wie zum Beispiel Methodenaufrufe, träges Klassenladen, dynamische Typüberprüfung oder Exceptionhandling, werden vorgestellt.

Trotz des hohen Abstraktionsniveau durch Haskell, ist die Performance des Prototyps gut. Das Ergebnis wird mit bekannten JVMs verglichen.

Contents

CHAPTER 1

Introduction

1.1 The Java virtual machine

Instead of emitting native code directly, the Java compiler (`javac`) generates a so-called *class file*. Thus, the application is *portable* without recompilation for different target architectures.

The class file contains a code section describing methods with a compact representation, called Java bytecode. The format of the class file and the behaviour of each bytecode instruction is specified by the JVM specification [LYBB13]. A *virtual machine* is used to execute such class files. The idea is, to have a separate virtual machine for each target architecture, that implements the same semantics for Java bytecode across different machines.

The JVM specification does not specify which implementation technique should be used to execute bytecode, and therefore offers plenty room for improvements by implementers regarding optimizations.

For the sake of simplicity, the first JVMs were interpreters. One technique used to optimize execution of Java applications is *just-in-time* compilation. That is, a given bytecode sequence is compiled at *run-time* to native code. Although initial effort is higher than pure interpretation, the execution of the method itself is a lot faster.

Interpretation and just-in-time compilation can be mixed too, as done for example in the HotSpot$^{\text{TM}}$ client compiler [KWM$^+$08].

1.2 Tasks of a Java virtual machine

The entry point of the JVM is the `main`-method of a given class file. It has to execute each bytecode instruction, including different kinds of method invocation. The JVM is a *stack machine*, that is a stack is used for computations. For example, the `IADD` instruction expects the summands on the stack and pushes its result on the stack.

Primitive types such as integer or float are supported. There is a special type called `reference` which is used to manipulate heap objects—objects are not allocated on the mentioned stack but on a heap. Although the JVM bytecode is mostly typed, in some cases the type has to be determined at run-time, e.g. on `invokevirtual` the receiver method has to be obtained depending on the object type at run-time, since the type of the object type is not known at compile-time in general—this is also called *dynamic dispatch*.

Certain bytecode instructions can raise an *exception*. The JVM has to find the correct handler for the exception, which depends on the type of the exception and the installed handlers.

Apart from that, the JVM has to look up classes at run-time if required. Also, not directly included in a JVM, a base library is necessary for executing primitive programs.

1.3 Motivation

Nowadays virtual machines for the Java language are complex software applications [AFG+11, KWM+08]. Prominent JVMs are implemented in C or C++, some are even implemented with Java itself using tricky bootstrap mechanism. C is arguably a rather old language and does not provide much features useful for abstraction. C++ on the other hand is powerful, but complex itself and sometimes obscure. Haskell, being a high-level language, provides mechanism helpful for abstraction. It is indeed often used for compiler tasks, such as parsing, in order to demonstrate the beauty and power of the language.

At first glance, low-level tasks do not fit in the picture of a high-level language: They need access to raw hardware, such as controlling every bit. But the appearance is deceptive: Haskell, in the environment of the Glasgow Haskell Compiler (GHC), provides useful features for such low-level tasks. As Haskell is not widespread, some of those areas are uncovered. The challenge is to elaborate those caveats when implementing a just-in-time compiler, that indeed requires low-level interaction.

Compilers are complex and expensive software, which tend to be hard to test and maintain. Equipped with Haskell, its powerful abstraction mechanism and tools we expect significant improvements regarding maintainability, testability and code reusage. Also, with respect to code quality and safety we expect improvements due to the advanced type system provided by Haskell.

1.4 Project context and activities

A prototype called MateVM was initiated with Harald Steinlechner. The origin of the name was inspired by the caffeine-contained drink called *Club-Mate*, to stay in tradition with other JVM implementations such as Kaffe or CACAO.

The first commit was 15th March 2012. We evaluated, if it is possible to implement and apply just-in-time compilation techniques in Haskell. On our first steps we heavily relied on the *Foreign Function Interface* provided by GHC and implemented some tricky bits in C. But over time we were able to move more things into Haskell, resulting in having less than 100 lines of code in C now.

At the beginning, we decided to choose a simpler scheme for the back end, namely emit native code directly from Java bytecode, i.e. mapping the Java stack to the `x86` stack. After we had certain features implemented, we came up with the current compilation pipeline. For more details, see Chapter 2.

MateVM is licensed under the GPL3 and is available on GitHub `https://github.com/MateVM`. Build instructions are included. Note that the only supported architecture is `x86` at the moment. Harald Steinlechner contributed BöhmGC bindings for Haskell and an exact garbage collector implemented in Haskell [Ste13]. Apart from that, he contributed many useful thoughts, reviews and code to the implementation of MateVM.

Although we try not to reference to the actual source code directly, we refer to commit id `c1a5c49ec` of the `git`-repository.

CHAPTER 2

Overview of MateVM

MateVM is a just-in-time compiler for Java bytecode. In the current state not all parts of the JVM specification [LYBB13] are covered, but we are trying to be as accurate as possible when implementing features.

2.1 The virtual machine

MateVM consists of a compiler and a run-time environment. The execution flow is depicted in Figure 2.1. The entry point of the virtual machine is the `main` method of the given program. After compiling the `main` method, the generated code is installed by the run-time system and triggers the execution of that code immediately. On certain points, the native code transfers the control to the run-time system, which then can take further actions, such as compile a new method, code patching, allocating memory and so forth. The parts are explained in detail in the next chapters.

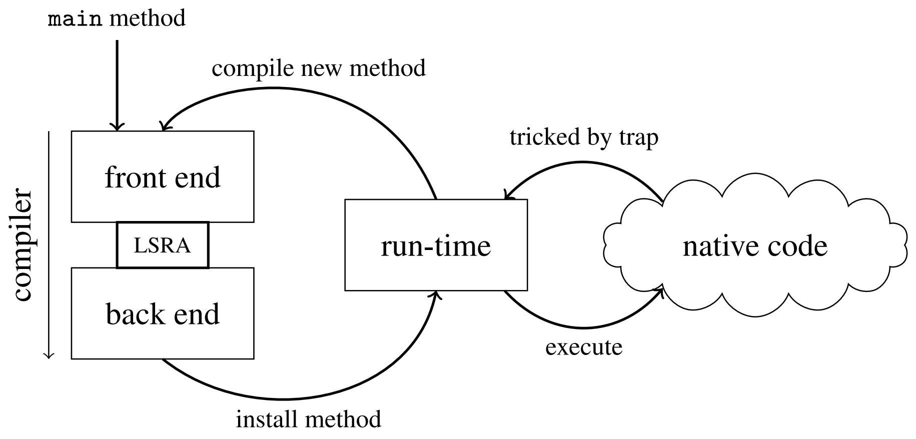


Figure 2.1: Execution flow in MateVM

Current features of MateVM are:

- different kinds of method invocation (Section 4.2.4)
- notion of lazy class loading (Section 5.1)
- array- and object-creation (Section 4.2.3)
- dynamic type checking (Section 5.3)
- exception handling (Section 4.4)
- linear scan register allocation (Section 3.6)
- efficient code generation for `x86` (Section 4.2)
- garbage collection (Section 5.5)

Worth noting, features missing (cf. Chapter 9) to be a real-world Java virtual machine:

- floating point support. However, some preparation are already there and Harpy is able to emit SSE instructions.
- `long` primitive type. Again, preparations are there, but this is a "won't do"-point unless we can generate code for `x86_64`.
- the concept of class loaders
- GNU Classpath support
- implementation of the *Java Native Interface* [Mic03]
- threads and concurrency
- verification of Java bytecode

2.2 Libraries useful for this project

In the Haskell community people contribute applications and libraries to a common place, called Hackage [Com12]. Fortunately, there are already some useful libraries for compiler engineering, in particular for this task.

2.2.1 `hs-java`

`hs-java` is designed to build and disassemble Java class files, while MateVM only use the latter functionality. It implements processing of class files according to the JVM specification in a Haskell idiomatic way. The interface enables us to avoid messing around with details of the class file format, which can be a time consuming task. For example, `hs-java` resolves references into the *constant pool*.

2.2.2 Hoopl

Hoopl [RDPJ10] is a library for data-flow analysis on a basic block basis. Although originally designed for GHC [oG12] (a static compiler), it turned out to be suitable for MateVM too, i.e. for applying analysis or transformations on the intermediate representation. Luckily, the compile-time overhead seems to be low too, therefore matching the requirements of a just-in-time compiler. Hoopl uses the Chamber-Lerner-Grove algorithm [LGC02] to solve its task. A consequence is, that the used intermediate language does not require to be in static single assignment form.

There are two types of passes which can be implemented with Hoopl: forward- and backward passes. Both consists of three components:

Lattice describes the data structure used by the data-flow analysis and the merge function used on join points in the control flow graph.

Transfer function collects information or the so-called *data-flow facts*. One can specify for every intermediate instruction how Hoopl should obtain facts from it. In particular instructions closed on the exit (cf. Section 3.1) are interesting: The defined join function for the lattice has to guarantee that it is possible to reach a fix point with it.

Rewrite function can be used to *rewrite* the Hoopl-graph supported by the data-flow facts. The result type is realized with the `Maybe`[1] data type: You can leave the instruction as-is (`Nothing`) or replace it with a new graph (denoted by `Just ...`).

Hoopl itself is responsible for computing a fix point, therefore its actual implementation is not relevant for the client. In order to use Hoopl as a library, one has to apply some design decisions to its intermediate language, for example the notion of *shapes*, see Section 3.1.

2.2.3 Harpy

Harpy enables run-time code generation in Haskell [GK07]. The library exposes the so-called `CodeGen`-monad, which enables us to write actual `x86` instructions in a style of a domain specific language[2], cf. Listing 2.1. This example generates native code for the Java bytecode instruction `GETSTATIC`, that pushes the value of a static field on the stack.

Consider the type signature in the first line: The function requires a Java instruction, as defined by `hs-java`. The “result” of the function is a bit weird to understand for non-Haskell user: It is a so-called *monad*, which encapsulate a sequence of actions. In this example, the *CodeGen*-monad provided by the Harpy-library is used. The `CodeGen`-monad consists of an environment (here the type variable `e`), a state (`CompileState`) and a return value (here `()` [3]). In the second line, *pattern-matching* is used in order to make a case distinction and to unpack the class index for the corresponding field.

Then, the local state is consulted, which contains a reference to the class file containing the compiling method. With that, the address can be requested via `getStaticFieldAddr`. But

[1]definition: `data Maybe a = Just a | Nothing`

[2]similar to the Intel syntax

[3] `()` has a similar meaning in Haskell as `void` in C. The type is called `()` and its only constructor is `()` too.

```
1 emit :: J.Instruction -> CodeGen e CompileState ()
2 emit (GETSTATIC cpidx) = do
3   cls <- classfile <$> getState
4   addr <- liftIO (getStaticFieldAddr cls cpidx)
5   mov eax (Addr addr)
6   push eax
7
8 getStaticFieldAddr :: Class Direct -> Word16 -> IO Word32
```

Listing 2.1: Example of the DSL provided by Harpy

what means `liftIO` here? That the expression in parentheses has side-effects. Those side-effects could be class loading, which in turn allocates memory for static fields and requires to execute a static initializer of a class[4]. That is, such an expression has to be *explicitly* annotated, otherwise the Haskell compiler does not accept it as valid program as the types do not match.

Finally, the actual operation can be done: a memory access and pushing the result of that read operation. Note, that the `mov`- and `push`-instruction will be written into memory as-is, i.e. assuming `addr = 0xcafebabe` this would be:

```
mov 0xcafebabe eax # AT&T syntax. Note the missing $
push %eax
```

While emitting instructions within the `CodeGen`-monad, side-effects are produced to the local state, e.g. the program counter is incremented from line 5 to line 6.

Note that this example would be a wrong implementation of the lazy class loading requirement. In MateVM we address this issue via hardware traps and code patching, see Section 4.3.

[4]The `cls` reference only contains the description to the class that has the information about the actual field offset

CHAPTER 3

Front End

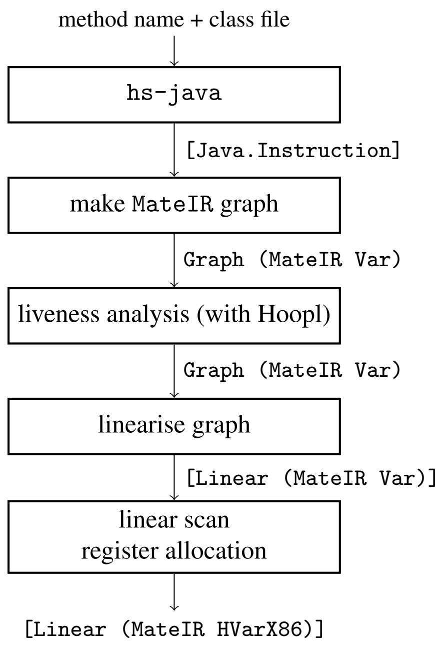


Figure 3.1: Stages in the front end pipeline

As already mentioned in Section 1.4, we moved from a simpler scheme to a more sophisticated one.

In Figure 3.1 the current state of the front end is depicted. After each stage the resulting type is denoted. In this chapter we describe the intermediate language first and then present each part of the front end in detail. The main contributions of the intermediate language are

- a more suitable representation than Java bytecode in terms of manipulation
- enable usage of Hoopl, which in turn enables the oppurtunity to develop further optimizations easily
- having a register-based representation, which is more suitable for optimizations, register allocation and efficient code generation

3.1 Intermediate Language: `MateIR`

In Listing 3.1 the definition of the `MateIR` data type is shown. *Generalized algebraic data types* (GADTs) [JWW04] is used, a type extension useful for defining `MateIR`. Actually, this is a requirement by Hoopl: Consider the first line, there are three type variables: `t`, `e` and `x`. `t` is used as one would assume when thinking of a type variable, namely as placeholder for other types. That is, `MateIR` is *polymorphic* regarding `t`.

But, this is not exactly true for `e` and `x`: They are not mentioned anywhere in the definition of the constructors. Instead, `O`'s and `C`'s are everywhere—they are defined like this:

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

So they have one constructor each, with the same name as the type. They are also called *phantom types*. Instead having a value associated, they are used to express invariants that are proved by the type checker. They naming of `e` and `x` is not accidentally: `e` should be an abbreviation for ***e**ntry* and `x` for *e**x**it*. Therefore, `e` and `x` denotes if a single instruction is open or closed on the entry as well for the exit; this is called the shape of an instruction[1].

```
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 {-ref-} -> t {-src-} -> MateIR t O O
 IRLoad    :: RTPool t -> t {-ref-} -> t {-dst-} -> MateIR t O O
 IRMisc1   :: Instruction -> t {-src-} -> MateIR t O O
 IRMisc2   :: Instruction -> t {-dst-} -> t {-src-} -> 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 {-src-} -> [(Maybe Int32, Label)] -> MateIR t O C
 IRReturn     :: Maybe t -> MateIR t O C
```

Listing 3.1: Intermediate representation of MateVM

Taking this further, one will discover the notion of basic blocks: An `IRLabel` can be the "entry instruction" of a basic block. 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.

This probably seems unnecessary complicated on the first sight, but consider building a graph for `MateIR`: Blocks can be built up of a certain shape. The type system is told what shape is expected when building such a block. Then, the type system *guarantees* that things are

[1]the notion of shapes also applies for blocks and graphs

correct at compile-time. When things are done wrong (e.g., try to pass a graph of the wrong shape), the type system of Haskell will refuse compilation with a type error.

Consider this type 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 for `GraphRep`—think of an interface for a graph data structure. `n` would be in our case `MateIR`, see below for an instance of `NonLocal`. The function `(<*>)` is used to connect two graphs, with a certain restriction: The first being open on the exit and the second open on the entry. Due to this property, a new graph can be *safely* connected preserving the shape of the original entry of the first graph, and exit of the second graph. The resulting and actual shape does not concern the function `(<*>)`.

In order to gain support for Hoopl, it has to determine what the predecessor or successor (depending on the shape) of a `MateIR`-instruction are. This is solved by a type class called `NonLocal` which has to be implemented for `MateIR`, cf. Listing 3.2. Note the elegance of

```haskell
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 _) = []
```

Listing 3.2: Definition of `NonLocal` for `MateIR`

GADTs here: The type system already knows, that `IRLabel` is the only instruction where an implementation is required for `entryLabel`, as it is the only constructor which is closed on the entry. For a more detailed explanation look up the Hoopl-paper [RDPJ10].

Let's consider the first type variable `t` again: This is the place holder for a type describing an actual variable or register. First, as shown in Section 3.2, `MateIR` is equipped with virtual registers as hardware registers are not relevant yet. A virtual register is an integer annotated with its computational type[2] or a constant of the corresponding type.

However after register allocation, virtual registers should be replaced by hardware registers of the target CPU architecture, but preserving the graph representation, or, described differently, a function that has the type signature:

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

While this is basically the idea, it is not easily possible to apply the register allocation algorithm

[2] `int`, `long`, `float`, `double` or `reference` as defined in JVM specification [LYBB13, § 2.11.1.]

to a graph. Therefore the graph is flattened to a linear representation before doing register allocation, as described in Section 3.4.

3.1.1 Design of the intermediate language

The pseudo instruction `IRLabel` defines the beginning of a basic block. It contains information about relevant exception handlers for this block and if this block is a handler itself. This information is valid for the whole block, as it is ensured that basic blocks are split on exception region boundaries.

Fall through instructions (shape O O)

`IROp` is used to represent a binary operation, e.g. addition, which has two registers as input ("source") and one register as output ("destination"). The representation is therefore a register machine. In order to achieve this, the computation stack of the Java bytecode has to be eliminated (Section 3.2).

For `IRStore` and `IRLoad` respectively, a further element with the type `RTPool t` is introduced, which is a reference to the *run-time constant pool* of the class of the method being compiled: The definition in Listing 3.3 reveals, that some constructors need a register (denoted

```haskell
data RTPool t
    = RTPool Word16 -- e.g. field of objects
    | RTPoolCall Word16 GCPoint -- method invoc. or 'new'
    | RTArrayNew Word8 MateObjType GCPoint t
    | RTArrayIndex t VarType
    | RTArrayLength
```

Listing 3.3: Definition of `RTPool`, a reference to the run-time constant pool

by the type variable `t`). For instance, `RTArrayNew` is used for allocating a new array. The size could be determined by a constant or by a register (whether the size is known at compile-time of the Java program or calculated at run-time). `IRStore` expects a reference to a run-time object and a value, which should be stored to it. Depending on the run-time object, the destination is computed differently. For example, assume a store-operation to a field, then the corresponding offset has to be obtained from the run-time system (see Section 5.1). `IRLoad` is similar, but loads a value from an object into a register.

`IRMisc1` and `IRMisc2` forwards an `Instruction` (which is a Java bytecode instruction) through the pipeline to the back end. For example `INSTANCEOF` is such an instruction, that should be handled by the back end directly. Though, it expects an object reference on the stack and pushes the result as a boolean, that is, it has a source register and a destination register. Therefore the register allocator has to take care of it.

`IRPrep`, `IRInvoke` and `IRPush` are used for executing a method call. `IRPrep` contains a list of registers needed to be saved on the stack before executing a call. This is obviously important for the back end, but also important for the garbage collector, see Section 4.2.4.2.

`CallingConv` denotes if the registers should be saved or restored. `IRPush` "pushes" arguments for method invocations on the stack. This notion is actually convenient for the calling convention used for integers on `x86`. However, `IRPush` also contains the order of the parameter and thus it is easily possible to implement other calling conventions in the back end. `IRInvoke` does the actual method invocation: It has a reference to the run-time constant pool to resolve the target method. Also, it has *maybe* a return register, which has to be allocated if used—this is represented by the type `Maybe t`. Last, it has a description what type of call it is: static, special, virtual or interface.

Note the type regarding entry and exit of `IRInvoke`: It does not end a basic block, but it is considered as fall through instruction. We consider a method invocation as an operation, that does not change the control flow of a program (i.e. what `O O` actually says). However, Java has exceptions which can change the control flow of the caller. But, for easier processing the intermediate language, we do not want to map exception behaviour to the intermediate language and ignore exceptions on that level. In fact, the control flow graph would explode, since many instructions could trigger a runtime exception.

Instructions changing the control flow (shape `O C`)

`IRJump` is used for two scenarios: First for `GOTO` and second on block splitting. The latter case occurs when an instruction is a branch target or it is the start of a try block. Therefore, `IRJump` obviously just contains one label.

`IRIfElse` is the classical operation of the well-known conditional construct. The Java bytecode has several encodings for this operation, which can be all described by `IRIfElse`. This instruction contains:

CMP this data type is defined in `hs-java` and encodes the wanted type of comparison, e.g. `equals` or `less than`.

two registers the two operands on that the comparison is applied to.

two labels referring the two possible target blocks, depending on the result of the comparison.

`IRExHandler` is a fake instruction, which is just used to reference exception handlers. This is needed, since handler blocks are not refered by any jump or whatsoever in the non-exceptional case, hence Hoopl would consider to remove such a basic block, cf. Section 3.4.

In order to implement the Java bytecode instructions `TABLESWITCH` and `LOOKUPSWITCH`, `IRSwitch` is used. It contains a register, which is used for comparison, and a list with values and its target block (denoted as label). Note the type `Maybe` of the value in the tuple: While everything with a concrete value is denoted with `Just ...`, the `default` case of the switch statement is embodied by `Nothing`, but still having a label assigned.

`IRReturn` *maybe* has a return value (again, denoted by `Maybe t`), but has not any successors and therefore contains no label in its constructor, despite being an instruction of shape `O C`.

Miscellaneous fact about `MateIR`: The actual memory encoding is left to the Haskell compiler, which contributes to further abstraction.

3.1.2 Combinators for `MateIR`

Often a situation requires to apply a certain manipulation to an instruction. The following functions are introduced for this purpose:

1. get a list of registers which are used by the instruction
2. get a list of registers which are defined by the instruction
3. map all registers of type `a` of this instruction to a register of type `b`

For the last case, consider the example in Listing 3.4: Every virtual register is mapped[3] to the hardware register `eax`, and that for each instruction of the graph.

Those combinators are used in liveness analysis (see Section 3.3.1) or in the register allocator (see Section 3.6). It enables to write simple and readable code.

```
mapIR :: (t -> r) -> MateIR t e x -> MateIR r e x

everythingEax :: [MateIR Var O O] -> [MateIR HVarX86 O O]
everythingEax instructions =
  map (mapIR (\_ -> HIReg eax)) instructions
```

Listing 3.4: Assign each virtual register to `eax` with `mapIR`

3.2 Java bytecode to `MateIR`

In order to gain a list of Java bytecode instructions, the code section for the corresponding method has to be extracted from the class file. This representation should be transformed to a graph containing `MateIR`-instructions. In order to identify basic blocks, two passes are needed over the Java bytecode stream:

1. resolve each jump offset of the given code segment. This "preparation" pass is necessary to resolve backward references, e.g. for some kind of loops. Also, exception handlers and try blocks[4] are marked as block boundaries in this pass.
2. building basic blocks by translating Java bytecode instructions to `MateIR` instructions. Since block boundaries are known from the first pass, basic blocks can be easily determined.

[3] in fact it does not matter what the type is, as it is ignored anyways (denoted by _)
[4] i.e. that is visible from the bytecode level

3.2.1 Elimination and type analysis of the Java stack

The stack representation should be translated to a register based representation, therefore the stack temporaries have to be mapped to registers somehow.

Although many Java bytecode instructions are typed (e.g. `IADD`), some are not, for example `DUP` and `POP`. However, this would be a valuable information for the back end, for instance the code generator has to know the byte width of the element to duplicate.

Both problems can be tackled in one step: When an instruction like `BIPUSH` is encountered, the constant (as a register) and its type (in this case `int`) is pushed on a *simulation stack*. On an operation, e.g. `IADD`, two registers are popped from the simulation stack and create a new virtual register to store the result of the operation. The destination register is pushed to the simulation stack, so it is usable for further instructions. As a result, `IROp` instruction is equipped with the mentioned registers.

```
BIPUSH 22
DUP
IADD
```

Figure 3.2: Java bytecode snippet

Consider Figure 3.2 for a short example. Listing 3.5 shows how the instructions are handled by the translator. The stack effect and result is depicted in Listing 3.6.

```
{- stack effect:
        stack before IADD: (constant 22) (constant 22)
        stack after IADD:  (virtual reg 1) -}

IROp Add (VReg 1 JInt) (JIntValue 22) (JIntValue 22)
```

Listing 3.6: Stack effect and result as `MateIR` of Figure 3.2

The implementation for `apop`, `apush` and `newvar` is trivial: The so-called *`State`-monad* is used in order to *emulate* imperative computation. The state consists of a Haskell list (representing the stack) and a counter for generating new virtual registers, as a virtual register is denoted by an unique identifier and its type. This example is of course simplified, for the actual implementation consult `./Compiler/Mate/Frontend/MkGraph.hs` of the source code.

Note that this is some kind of verifier for free here: Every register pair is type checked if it has the expected type, e.g. in the example above both parameters are checked if they match regarding their types. If they do not match, the given bytecode is not valid according to the JVM specification and the execution of the virtual machine is aborted.

3.2.2 Non-empty stack on block boundaries

Occasionally it happens that the Java stack is not empty on a basic block boundary. We chose a similar solution as in [Kra98, Section 3.2]: When the stack contains elements on boundaries, those elements are popped from the stack until it is empty. For every possible basic block successor, the mentioned elements are stored. When processing such a successor block, the elements are pushed right at the beginning of the block to the simulation stack.

```haskell
data SimulationStack = SimulationStack
        { vregCounter :: Integer
        , simStack :: [(Integer, VarType)] }

translate :: Instruction -> State SimulationStack [MateIR O O]
translate (BIPUSH w16) = do
        apush (JIntValue w16)
        return []

translate DUP = do
        var <- apop
        dup <- newvar (varType var)
        apush var
        apush dup
        -- move content of var into a fresh register
        return [IROp Add dup var (nullConst (varType var))]

translate IADD = do
        op1 <- apop
        op2 <- apop
        when (typeOf op1 /= JInt || typeOf op2 /= typeOf op1) $
                error "type mismatch"
        nv <- newvar JInt
        apush nv
        return [IROp Add nv op1 op2]

run :: [MateIR Var O O]
run = evalState translate (SimulationStack 0 []) [BIPUSH 22, DUP, IADD]
```

Listing 3.5: Simulating stack effects

3.3 Hoopl passes

Due to the special crafting of `MateIR` with GADTs, data flow passes are ready to be implemented in Hoopl. Data flow problems are common in compiler construction and there are plenty of them ready to implement. But, when implementing a just-in-time compiler, requirements are different to a static compiler with respect to compile-time. While being able to use Hoopl in MateVM, it cannot be used at its full glance yet, in order to meet those requirements. In the future, however, thinking of recompilation, Hoopl will certainly pay off, i.e. compiling hot methods with more expensive passes resulting in better code.

3.3.1 Liveness analysis

A well known analysis is the liveness analysis of variables: The used data structure is a set of virtual registers and the merge operation is the union operator. The analysis is a backward pass and the *bottom* is an empty set. Every instruction is inspected for two aspects, consider the following example:

```
IROp Add (VReg 10 JInt) (VReg 5 JInt) (VReg 4 JInt)
```

gen-set variables are used by it (the virtual registers 5 and 4)

kill-set variables are defined by it (virtual register 10)

The transfer function is depicted in Listing 3.7. Due to the application of the combinators from Section 3.1.2, the code is short. While instructions of the shape `C O` and `O O` are handled in the same manner, `O C` needs special care: On such instructions it is possible that several control flows merge together. Since the liveness analysis is backwards, the facts are simply looked up for every successor (via the `successors` function from the `NonLocal` type class, defined in Section 3.1) and apply the union operator via the well-known higher-order function `fold` [Wik12]. The liveness information will be used as a basis for three tasks in MateVM:

- Computing live ranges, which are designated as input for register allocation (see Section 3.6).

- Dead-code elimination. This optimization would be free, as liveness analysis is needed anyway for register allocation. However, the JVM specification has some pitfalls; consider Listing 3.8: Although `a` is dead in this example (as would be `b` then), `a` and the division cannot be deleted, because it would not trigger the arithmetic exception anymore. That is, the JVM specification [LYBB13, § 2.10.] requires *precise* exceptions regarding the bytecode representation. Of course one can carefully consider all exceptional cases, but this can be cumbersome in context of lazy class loading and probably other things, hence dead-code elimination is disabled at the moment.

  However, dead assignments have some side effects in the current implementation: The live range for a dead variable cannot be computed as it has no single use. Therefore, the register allocator will never assign a register. In such cases, the unassigned register is mapped to a spill location on the stack. Thus, dead variables do not consume valuable hardware registers at least.

- Liveness information is used for defining *garbage collection points* at certain places in the generated code, see Section 4.2.4.2. Despite having a bit different requirement, the same information could be adapted to be used for *replacement points* [SKT07, Section 4.3] too in the future.

```haskell
type LiveSet = Set VirtualReg

bot :: LiveSet
bot = Set.empty

factLabel :: FactBase LiveSet -> Label -> LiveSet
factLabel f l = fromMaybe bot (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 bot
                    (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
```

Listing 3.7: Transferfunction for the liveness pass

3.4 Linearise the graph

For further processing, namely register allocation and code generation, a linearised representation is useful. Hoopl provides some helpers to do this, however there is a problem with the type system: Lists of this kind cannot be built

```haskell
[IRPush 1 (VReg 1 JInt), IRJump (Label 2)]
```

as they have a different shape (`O O` and `O C`), so the type system *refuses* to compile.

```
public static void main(String []args) {
        int b = 1234;
        try {
                int a = b / 0;
                System.out.printf("goodbye\n");
        } catch (Exception _) {
                System.out.printf("div / 0\n");
        }
}
```

Listing 3.8: Eliminating a would lead to wrong behaviour according to the JVM specification

A simple wrapper data type can be defined, see Listing 3.9. Again, LinearIns is polymorphic regarding the used type for registers like MateIR is. Now, the example from above can be represented as a list:

```
data LinearIns t
  = Fst (MateIR t C O)
  | Mid (MateIR t O O)
  | Lst (MateIR t O C)
```

Listing 3.9: Linear representation of MateIR

```
[Mid (IRPush 1 (VReg 1 JInt)), Lst (IRJump (Label 2))]
```

Basic blocks of exception handlers

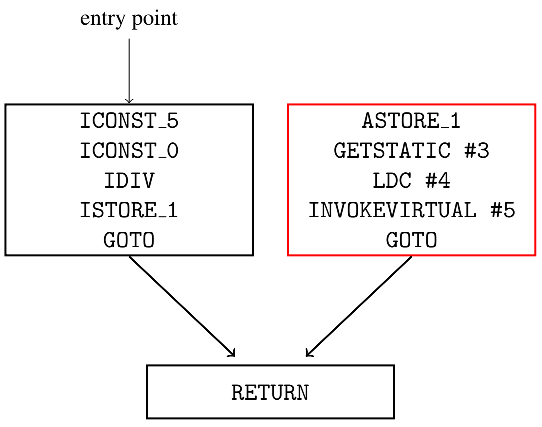
Usually a basic block which is a part of an exception handler will never be referenced by an instruction of the non-exceptional control flow. The problem is depicted in Figure 3.3. Hoopl is designed to remove unused blocks, as this can be a side-effect of an optimization—think of constant folding, which enables to evaluate an expression in an if-statement at compile-time. Hence, it must be ensured that exception handler blocks will not be removed on flattening the control flow graph. In Section 3.1, the pseudo instruction IRExHandler is introduced, which is used to reference all exception handlers at the beginning of a method.

3.5 Computing live ranges from liveness information

After linearising the Hoopl-graph, live ranges can be computed from the liveness information for each instruction. In fact, the actual instruction is ignored and the liveness information is taken for this task, but the sequence must be preserved though. In order to enable fast access to needed information in the linear scan algorithm, the data structure in Listing 3.10 is used. PC is an unique identifier for each MateIR instruction in an ascending manner. In LiveStart every program counter is mapped to a list of virtual registers which are defined at this instruction. On the other hand, the last use of every virtual register is stored in LiveEnd.

```java
public static void main(String []args) {
        try {
                int a = 5 / 0;
        } catch (Exception _) {
                System.out.println("bye");
        }
}
```

(a) Java source code



(b) Control flow graph

```
public static void main(java.lang.String[]);
  Code:
   0:   iconst_5
   1:   iconst_0
   2:   idiv
   3:   istore_1
   4:   goto    16
   7:   astore_1
   8:   getstatic     #3; //Field System.out
   11:  ldc     #4; //String bye
   13:  invokevirtual #5; //Method println
   16:  return
  Exception table:
   from   to  target type
     0     4     7   Class java/lang/Exception
```

(c) Bytecode

Figure 3.3: Illustration of an unreferenced exception handler block

```haskell
type PC = Int

type LiveStart = M.Map PC [VirtualReg]
type LiveEnd = M.Map VirtualReg PC

data LiveRanges = LiveRanges LiveStart LiveEnd
```

Listing 3.10: Data structure to represent live ranges of registers in a method

The implementation is a single pass over the liveness information, adding information to the map on each instruction in the input list.

3.6 Linear Scan Register Allocation (LSRA)

LSRA [PS99] enables register allocation in one linear pass over the instructions, in contrast to more sophisticated algorithms e.g. graph coloring. Since LSRA is linear, it enables *fast* register allocation and therefore is suitable for a just-in-time compiler.

We implemented the algorithm presented in [PS99], see Listing 3.11. The data structures for `liverangeStarts` and `liverangeEnds` are given in Listing 3.10. The algorithm iterates over the computed live ranges of the instructions, where each instruction has a virtual program counter assigned.

```
lsra = do
  for [0 .. lastPC] $ \pc -> do
    case pc `lookup` liverangeStarts of
      Nothing -> return ()
      Just newregs -> do
        freeRegs pc
        for newregs $ \reg -> do
          if hardwareregAvailable
            then assignHWReg to reg
            else spillReg reg
    collect information for gc points

freeRegs pc = do
  for activeRegs $ \reg -> do
    when ((liverangeEnds ! reg) < pc) $ do
      delete from activeRegs
      add to available hardware regs

spillReg reg = do
  if some assigned HWReg has a longer live range
    then spill that reg and assign freed HWReg to reg
    else spill reg
```

Listing 3.11: Algorithm of LSRA (pseudocode)

If there is at least one new register, the active set of registers is considered first and registers are removed if their live range has passed (`freeRegs`). After that, registers are assigned in no certain order to hardware registers. If no hardware register is available, the register is spilled to the stack.

However, registers with shorter live ranges are prioritized opposed to registers with long live ranges. Therefore, before doing the actual spill, it is checked if a hardware register is assigned to a register having a lower priority than the current register. If so, assignments are swapped: The low-priority register is spilled and the now available hardware register is assigned to the current register with the shorter live range. This results in much lesser spills overall, as registers with high register pressure are more likely to get spilled.

As our target architecture is `x86`, `ecx`, `edx`, `ebx`, `esi` and `edi` are used as initial pool of available hardware registers. `eax` is reserved as a scratch register and used for passing a return value in the calling convention. `esp` and `ebp` are used for stack operations, hence they are not available to the register allocator. Readers who are familiar with the `x86` architectures will remember that there are instructions which clobber certain hardware registers, for example `div` clobbers `ebx` and `eax`. We considered two solutions to this problem:

- integrate support for *pre colored* registers in the register allocator, and annotate each intermediate instruction that could clobber hardware registers (hence requires knowledge about the code generator in the front end)

- save registers on the stack (or somewhere else) before emitting the instruction clobbering certain registers, in the code generator

We decided to use the latter approach, since the first one would be only required for `x86` and we would like to move on to another architecture in the future, and therefore avoid ugly design decisions. Thus, the problem is moved to the code generator, cf. Section 4.2.1, which is `x86`-specific anyway.

While we were developing the new front end, LSRA was not implemented yet. Instead, register allocation was done in a *very* simple way: Just assign each register in sequence to virtual registers. After all hardware registers are assigned, continue with spills on the stack and ignore any information about liveness of registers. Compared to the temporary approach, the presented way of LSRA above does a good job even on `x86` with few registers. In fact, a spill occurs rarely.

Instead improving LSRA with respect to live range splitting [WM05], we think that the original problem should be eliminated in the first place: Avoiding register duplication, i.e. suppress move instructions where possible. One approach would be register coalescing before liveness analysis and register allocation, but a better solution would be to delay the duplication decision while transforming Java bytecode to `MateIR` (Section 3.2), as done in [Kra98].

Pre-assigned registers

Some virtual registers have a special purpose, such as arguments in a method. Those registers are not assigned by the register allocator, but are pre-colored. For instance an `int` return value of a method will be always assigned to `eax`, as this is defined by the calling convention (cf. Section 4.2.4.1).

The module described in Section 3.2 is responsible for creating new virtual registers. Since a virtual register is defined by a plain integer (and its type), ranges with special meaning are defined in order to have a portable design across different target architectures, see Table 3.1. “locals” are registers that are local to the method, which are also visible at Java bytecode level.

description	range
locals	1–49999
temporary	50000–99998
return (`int`)	99999
return (`float`)	100000
arguments (`int`)	200000–299999
arguments (`float`)	300000–599999
block interfaces	600000–∞

Table 3.1: Ranges of virtual registers

“temporary” registers are generated by the stack simulation for different purposes. Both registers have their type encoded in the data type of the virtual registers. The next groups are designated to their type, since it allows easier handling: E.g. on `int` parameters each register is mapped to its right position on the stack (think of the `x86` calling conventions), in order to avoid mixing it with `float` parameters in the first place. This behaviour is functional for every other popular architecture too, because they separate integer registers from floating-point registers.

Registers in the group “block interfaces” belongs to basic block boundaries, as explained in Section 3.2.2.

The knowledge to which group a virtual register belongs, can be useful for Hoopl-passes too, e.g. registers of the block interface group can play a critical role in terms of optimizations.

Admittedly this design decision is more like a hack and is not really Haskell idiomatic (that is, defining an own data type for that purpose). Furthermore, although the ranges are big and should be sufficient for any real-life application, it does not feel like a clean solution. A redesign of this implementation should be considered.

CHAPTER 4

Back End

In this chapter we present the architecture specific back end. First, we give a short introduction of how to handle native code in Haskell. Then we present the parts of the code generator designed for `x86`.

4.1 How to call native code from Haskell

Until now, the presented methods are all solvable in pure Haskell without interaction with raw memory or foreign functions. Since the back end has to generate *native code* and execute it, we have to leave the beautiful and pure world of Haskell and have to mess around with low-level details of the target architecture `x86`.

In Listing 4.1 a small example is shown. In line 26 actions of `mNativeAdd` are executed, which is encapsulated in a `CodeGen`-monad. Basically, it allocates a code buffer and emits the given instructions to that buffer. The effect returns the entry address of the code buffer and the disassembly.

In line 27 the `main`-function calls into the code buffer, namely by using `code_int` defined in line 12. For the definition in line 12, a GHC extension for the *foreign function interface* (FFI) is needed. With FFI imported functions of different languages (e.g. C) can be used by Haskell code. This is usually done statically, i.e. the job is delegated to the linker at compile-time. But in certain situations, the address is not available to the compiler or linker. FFI provides the `foreign import dynamic` primitive for that task [FLMPJ99, Section 3.4]. With that, a function can be defined that gets a function pointer of type `BinaryOperation`, which in turn returns a function of the same type but without the `FunPtr`-wrapper. It enables to use FFI-functions as normal Haskell functions, as done in line 27.

After the result of the addition is printed by the program, it also dumps the disassembly of the generated machine code for demonstration purpose. Assuming the example is stored in the file `harpy.hs`, one can compile[1] and execute the Haskell program as depicted in Listing 4.2.

[1]harpy must be installed via `cabal install harpy`

```haskell
{-# LANGUAGE ForeignFunctionInterface #-}
module Main where
import Foreign
import Foreign.C.Types
import Control.Applicative
import Harpy
import Harpy.X86Disassembler
import Text.Printf

type BinaryOperation = CInt -> CInt -> IO CInt

foreign import ccall "dynamic"
   call_int :: FunPtr BinaryOperation -> BinaryOperation

mNativeAdd :: CodeGen () () (FunPtr BinaryOperation, [Instruction])
mNativeAdd = do
  mov eax (Disp 0x4, esp)
  add eax (Disp 0x8, esp)
  ret
  fp <- castPtrToFunPtr <$> getEntryPoint
  d <- disassemble
  return (fp, d)

main :: IO ()
main = do
  (_, Right (nativeAdd, disasm)) <- runCodeGen mNativeAdd () ()
  result <- (call_int nativeAdd) 5 10
  printf "5 + 10 = %d\n" (fromIntegral result :: Word32)
  mapM_ (printf "%s\n" . showAtt) disasm
```

Listing 4.1: How to call into a memory blob containing code

Note, that the type system relies on the given type signature of `BinaryOperation` and cannot verify its correctness. If a native function is called which does not match the type signature, the run-time behaviour is undefined.

Issues with the foreign function interface

Apparently, a few Haskell programmers works on such a low level:

- While we switched to a new environment (Ubuntu 10.04 to 12.04) we encountered strange problems when executing code from a code buffer at run-time [Ove12b]. While we were not able to identify the original problem, we were able to work around the issue using `mprotect(3)` and declare the code buffer explicitly as executable memory region.

```
$ ghc harpy.hs
[1 of 1] Compiling Main             ( harpy.hs, harpy.o )
Linking harpy ...
$ ./harpy
5 + 10 = 15
0914c000  8b 44 24 04                movl   4(%esp),%eax
0914c004  03 44 24 08                addl   8(%esp),%eax
0914c008  c3
```

Listing 4.2: Compile and execute the example from Listing 4.1

- GHCi, the interactive mode provided by GHC, has problems using FFI. Our question regarding this problem is still unanswered on the popular stack overflow platform [Ove12c]. However, recently the GHC team disabled the custom run-time system-linker by default and relies now on the system linker [Mar12], which could among others solve some issues regarding GHCi.

We have shown in this section how a program can switch from Haskell world into native code and return regularly. However, in the situation of a just-in-time compiler, the execution of a program should be interruptible, that is get back to the run-time system (Haskell world) from native code, for example to patch small junks of native code or handle exceptions. MateVM solves that by using *hardware traps*, as presented in Section 4.3.

4.2 Code generation for x86

The function for emitting native code gets a list of type

```
[LinearIns HVarX86]
```

where HVarX86 indicates that each register of an instruction is already assigned to a hardware register.

For emitting each MateIR-instruction, pattern-matching is used. IRLabel is the only instruction being of shape O O. The CodeGen-monad provided by Harpy offers a label mechanism, similar to the labels known from assembly languages. Thus, for each IRLabel a new label is defined in Harpy. Also, the definition of that label is stored in the state of the CodeGen-monad.

Instructions of the shape O C heavily rely on the label mechanism by the CodeGen-monad. For IRIfElse the given registers are compared and the comparison flag from hs-java is mapped to the x86 architecture. Handling of IRSwitch is shown in Listing 4.3. For simplicity we did not consider to implement a jump table, but generating a series of cmp-instructions. For each je- and jmp-instruction respectively, a label obtained from the state of the CodeGen-monad is provided. The label contained in the table is actually a label defined in Hoopl-context and therefore something different. Hence, a lookup-table (lmap) is used where each Hoopl-label is mapped to a label in the sense of Harpy.

Note, that a label can be created in the `CodeGen`-monad, but can be *defined* later on. That is, Harpy is able to resolve unknown labels afterwards and patches them correctly.

Details about instructions of shape `O O` are presented in the remaining section.

```
IRSwitch src table -> do
  r2r eax src
  forM_ table $ \x -> case x of
    (Just val, label) -> do cmp eax (i32Tow32 val)
                            je  (lmap M.! label)
    -- "Nothing" is always the last entry in the table
    (Nothing, label)  ->    jmp (lmap M.! label)
```

Listing 4.3: Code generation for `IRSwitch`

4.2.1 Combinators

Haskell offers various opportunities to eliminate or at least minimize repetitive tasks. Often the situation occurs, that a register of type `HVarX86` has to be moved to a scratch register, mostly being `eax`. However, `HVarX86` is defined like this:

```
data HVarX86 = HIReg Reg32
             | HIConstant Int32
             | SpillIReg Disp
```

that is, `mov` defined by Harpy cannot be used in this context, but a case distinction would be necessary for each constructor. In order to simplify things for the scratch register `eax`, the following function would be appropriate:

```
hvar2eax :: HVarX86 -> CodeGen s e ()
```

Then, a function to copy back the content of `eax` to a register of type `HVarX86` would be convenient too.

In order to generalize that task (and avoid implementing helpers for each combination), a type class `RegisterToRegister`[2] is introduced. It consists of only one function `r2r`, see Listing 4.4. An instance for each useful combination is defined, e.g. consider the example in Listing 4.3 again: As the incarnation of `src` is not known (without looking at the constructor), `r2r` is used in order to move the content of `src` to the scratch register `eax`.

While we would be certainly able to optimize things, i.e. if `src` is a hardware register, we argue with better readability here though[3].

[2]this definition requires the `MultiParamTypeClasses` extension

[3]and the fact, that nowadays `x86`-processors optimize such cases anyway

```
class RegisterToRegister a b where
  r2r :: a -> b -> CodeGen e s ()

instance RegisterToRegister HVarX86 Word32 where
  r2r (HIReg reg) src       = mov reg src
  r2r (SpillIReg disp) src = mov (disp, ebp) src
  r2r i _ = error $ "r2r HVarX86 Word32: " ++ show i

instance RegisterToRegister HVarX86 Reg32 where
  r2r (HIReg reg) src       = mov reg src
  r2r (SpillIReg disp) src = mov (disp, ebp) src
  r2r i _ = error $ "r2r HVarX86 Reg32: " ++ show i

instance RegisterToRegister Reg32 HVarX86 where
  r2r dst (HIReg reg)       = mov dst reg
  r2r dst (SpillIReg disp)  = mov dst (disp, ebp)
  r2r dst (HIConstant i32)  = mov dst (i32Tow32 i32)
```

Listing 4.4: Type class `RegisterToRegister`

Free a hardware register for a region
Consider code generation for multiplication, the `mul`-instruction of `x86` *clobbers* certain registers. The register allocator does not respect this fact (cf. Section 3.6), hence the problem is delayed to the code generator. In order to address this issue, the combinator `freeRegFor` is introduced and its definition is listed in Listing 4.5. `freeRegFor` gets a hardware register (`Reg32`). For an usage example see Listing 4.7 on Page 32.

```
freeRegFor :: Reg32 -> HVarX86 -> CodeGen e s r -> CodeGen e s r
freeRegFor r32 dst body = do
  let isNotDst = case dst of
                   HIReg dst' -> dst' /= r32
                   _ -> True
  push r32    -- store hardware register on the stack
  res <- body -- execute given actions
  if isNotDst then pop r32                   -- restore hardware register
              else add esp (4 :: Word32) -- or delete backup entry
  return res
```

Listing 4.5: Free a hardware register for a certain region

4.2.2 Emitting `IROp` (binary operator)

For integer operations, the JVM specification [LYBB13, § 2.11.1.] defines some binary operators (cf. Table 4.1). The unary operator `INEG` can be substituted by `ISUB`. While one would assume to use the same pattern to emit code for binary operations, this is not possible with `x86`: For example, all shift instructions assume the amount of bits to shift to be in the register `ecx`. Therefore, the instructions are separated into groups in order to avoid code duplication, see Table 4.1.

Operation	default	shift	mul	div
`add`	X			
`sub`	X			
`mul`			X	
`div`				X
`rem`				X
`and`	X			
`or`	X			
`xor`	X			
`shr`		X		
`sar`		X		
`sal`		X		

Table 4.1: Grouping of binary operators, respecting `x86` quirks

```
type DefaultOp = forall a b.
    (Add a b, Sub a b, And a b, Or a b, Xor a b) =>
    a -> b -> forall e s. CodeGen e s ()

gd :: DefaultOp -> HVarX86 -> HVarX86 -> HVarX86 -> CodeGen e s ()
gd opx dst src1 src2 = do
  r2r eax src2
  case src1 of
    HIConstant c -> opx eax (i32Tow32 c)
    HIReg r      -> opx eax r
    SpillIReg d  -> opx eax (d, ebp)
  r2r dst eax
```

Listing 4.6: Emitting code for the default group

Emitting code for the default group is rather easy, see Listing 4.6. The operation is given as a function `opx`, the destination register and both source registers as `HVarX86`. For readability, the presented combinator `r2r` is used to move the content of the second source register into the scratch register `eax`. For the first source a case distinction has to be done in order to unpack needed information, where also the given operation is emitted with its arguments. After that, the result is stored to the destination, again by using r2r.

Admittedly the type signature is quite hard to understand. It requires knowledge about:

1. how Harpy defines instructions on a type level

2. RankNTypes, an extension to the type system of Haskell

Harpy defines instruction via type classes in order to be able to reflect the different use scenario of x86-instructions to the programmer. For example, consider the type signature of add:

```
add :: Add a b => a -> b -> CodeGen e s ()
```

where Add[4] is a type class, required to be implemented for the type variables a and b in order to use the function add. As an example consider

```
add eax ebx             -- a = Reg32, b = Reg32
add eax (Disp 8, ebp)   -- a = Reg32, b = (Disp, Reg32)
```

That is, an instance has to be defined for the type class Add for each type pair (a, b) if it should be usable from the CodeGen-monad. Given that, it is a similar pattern as for Register2Register. This explains the type class constraint in DefaultOp: It has to be ensured on a type level, that each pair of types is defined for the used instructions. Also, this determines the requirement, that all possible instructions in this groups have to implement the same kind of addressing modes in a type safe way. More generally, this concept is also called a class constraint in Haskell.

RankNTypes

In Haskell98 each polymorphic parameter is unified with the complete function it occurs in i.e. they are universally quantified at top level. Consider the following example:

```
foo :: (a -> a) -> (Char, Bool)
foo f = (f 'c', f True)
```

When the type parameter a is instantiated by the type checker it must conform to all expressions the parameter occurs in. Therefore a cannot be Char and Bool at the same time.

[4]Haskell is case sensitive, whereas identifiers beginning with upper case denotes a type, type class or constructor

A type extension, called *Higher-Ranked-Types* or `RankNTypes` [VWJ08] lifts this limitation and permits universal quantifiers to occur nested within type-expressions, but requires explicit type annotations. The example now writes:

```
foo :: (forall . (a -> a)) -> (Char, Bool)
foo f = (f 'c', f True)
```

A function matching the requirements of `f` in `foo` is for example `id`:

```
id :: forall a. a -> a
id x = x
> foo id
('c', True)
```

Now back to Listing 4.6 (Page 30): `opx` is used in different contexts, namely:

- `(Reg32, Word32)`
- `(Reg32, Reg32)`
- `(Reg32, (Disp, Reg32))`

Therefore, there are three possible incarnations of `opx` regarding the type. I.e., `opx` has three different *ranks*, or more general, a higher rank. This feature is not available in Haskell98, and therefore requires to enable the `RankNTypes` extension. `forall` also works on type class constraints as it is done in `DefaultOp`.

Other groups

Another example is presented in Listing 4.7, emitting code for shift instructions. The combinators `r2r` and `freeRegFor` are used. The remaining groups have a similar implementation, while respecting the requirements of the `x86` architecture.

```
gs :: (forall a b. (Shr a b, Sar a b, Sal a b)
                => a -> b -> CodeGen e s ())
      -> HVarX86 -> HVarX86 -> HVarX86 -> CodeGen e s ()
gs so dst src1 src2 = do
  freeRegFor ecx dst $ do
    r2r eax src2
    r2r ecx src1
    so eax cl
    r2r dst eax
```

Listing 4.7: Emitting code for the shift group

4.2.3 Object- and array layout

For creating objects the layout of an object has to be considered. In Figure 4.1 the layout is depicted, also with its relationship to the method table and interface method table. In order to create an object

- the object size (that is, the amount of fields) has to be requested from the run-time system
- emit a call to the memory management unit to allocate memory.

More details about how offsets are generated are discussed in Section 5.1. Allocating memory

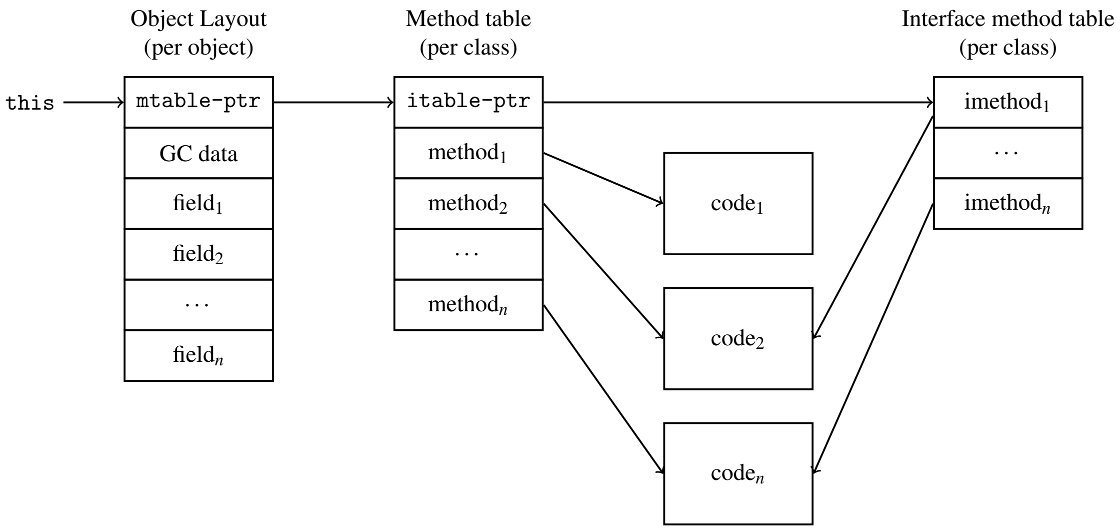


Figure 4.1: Memory layout of an object

from the run-time system is guarded by a trap (see Section 4.3), since it could trigger a lazy class loading event.

The memory layout of an array is shown in Figure 4.2. Although it is not a Java object, it contains a magic number at the position of the `mtable-ptr` field, in order to be recognizable as array by the garbage collector. There are two different magic numbers defined, since there are arrays for primitives and references. For the latter one, the garbage collector has to follow its references.

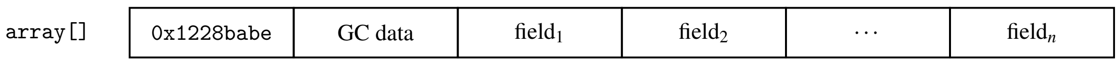


Figure 4.2: Memory layout of an array

Requesting memory from the run-time system

On creating an array or an object, memory has to be allocated. The memory manager needs to know the base pointer and the instruction pointer of the caller, in order to do a stack walk and to look up garbage collection points (Section 4.2.4.2). That information is usually available to the caller implicitly, due to the calling convention, as `eip` and `ebp` of the caller is stored on the current stack frame.

However, the function allocating memory is written in Haskell. Although it is callable from native code (via FFI), internally the generated code by GHC violates the calling convention, therefore the garbage collector cannot rely on the stack layout defined by the used `x86` calling convention. Thus, the Haskell world is considered as a black box and a different solution to gain `ebp` and `eip` is used: The current `ebp`, `eip` and the requested size of memory are passed as arguments to the allocating function, hence having the following type signature:

```
mallocObjectGC :: CPtrdiff -> CPtrdiff -> Int -> IO CPtrdiff
```

where `CPtrdiff` represents the C type `ptrdiff_t` in Haskell.

4.2.4 Method calls

Consider that a method `f` which calls an other method `g`. At compile-time of method `f` it is generally not assumed that method `g` is already compiled, that is, the entry address is not available. Instead, a trap is placed in order to switch to the run-time system on the first invocation of the method, which handles resolution of method `g` and patches the trap accordingly.

The four different types of method invocation as defined in JVM specification [LYBB13, § 3.7.] are discussed in this section. In order to show the core techniques of method calls, we ignore the used trap mechanism and assume everything is patched correctly. The trap mechanism is presented in Section 4.3.

invokestatic

Since the receiver of the call is known at compile-time, this is an ordinary relative call in terms of `x86`.

```
eip <- getCurrentOffset
target <- lookupMethod methodinfo
call (target - eip)
```

Therefore, `invokestatic` is the most efficient call type in MateVM.

invokespecial

Similar to `invokestatic`, but it has a further argument on the first position, namely `this`. This type of call is usually used for object constructors.

invokevirtual

For dynamic dispatch, the receiver of the call has to be determined at run-time. Fortunately, Java has only single inheritance of classes enabling simpler handling of dynamic dispatch. For that, a so-called *method table* (or `mtable`) is used which is computed by the class pool (see Section 5.1).

By dereferencing the pointer to an object, the `mtable` is obtained. In each `mtable`-slot a pointer to the method entry is stored, see also Figure 4.1. Hence, a virtual dispatch is done by first dereferencing the object pointer and then by dereferencing the result with a certain offset.

```
mov ebx this
mov eax (Disp 0, ebx)    -- mtable in eax
call (Disp offset, eax) -- indirect call
```

The offset is obtained from the run-time system (cf. Section 5.1).

invokeinterface
Since interfaces allow multiple inheritance in Java, this type of invocation is more complicated (that is, it requires a further table as shown in Section 5.1) and requires a further indirection.

```
mov ebx this
mov ebx (Disp 0, ebx)    -- mtable in ebx
mov eax (Disp 0, ebx)    -- itable in eax
call (Disp offset, eax) -- indirect call
```

A complete example is presented in Listing 5.1 (Page 47).

4.2.4.1 Calling convention

`cdecl`, a common calling convention on `x86`, is used. That is, arguments are passed on the stack to the callee in reverse order. Furthermore, `eax` is used to return a value from the callee to the caller. The caller is responsible to remove the arguments from the stack after the call. This decision enables easy calling of native functions, as it is the same calling convention as the host operating system uses.

On further investigation we probably can get rid of `esp` and use it as general purpose register in the register allocator, but that would require extra care when calling native functions and requires to eliminate all uses of `push`- and `pop`-instructions in the back end.

4.2.4.2 Savepoints for garbage collection

On calling the garbage collector, information has to be provided about the current state of used *references* located in registers or spill locations. Originally, that information is provided by the register allocator, but this information is associated with the program counter of the intermediate language. On code generation, a table has to be constructed that maps the instruction pointer from native code to the set of references. Savepoints are placed on

- claiming memory from the run-time system (objects and arrays).
- after a method call, that is the return point in the caller where the callee returns.

While the first is obvious, the latter is needed for walking the calling stack, since the garbage collector has to walk through the calling tree (see Section 4.4) to create an exact representation of used objects currently in use.

Savepoints are needed for *exact* garbage collection. A first version is implemented by Steinlechner [Ste13], which is relying on this mechanism.

4.3 Trap handling and code patching

Traps, e.g. invalid instructions, are placed by the code generator intentionally. Such traps causes a signal, caught by a handler which in turn calls into the run-time system of MateVM and does the appropriate actions. Hence, traps realize the edge from native code into the run-time system as depicted in Figure 2.1 (Page 5). At the moment traps are used for:

static calls Consider the example from above (method `f` and `g`), linkage is done at run-time. Due to the return into the run-time system, it is able to resolve and, if needed, to trigger compilation of the method `g`. After that, the entry address is patched to the `call` instruction accordingly.

This trap is active at most once.

virtual calls `invokevirtual` and `invokeinterface` are similar, since both call types require an offset and an appropriate entry in the `mtable` or `itable`. The example in Listing 5.1 (Page 47) explains that type of traps in more detail.

This trap consists actually of two parts: First, patching the right offset in the call instruction itself and second patching the entry in the `mtable` or `itable`. Offset patching is needed at most once, but for the latter case this trap can be triggered frequently, as the receiver can have a different type (e.g. a sub class and hence having an unpatched `mtable` entry).

object creation As the class of the object that should be instantiated may not have been loaded yet, the target object size and the `mtable`-identifier is not known at compile-time. Actually, we could come up with a different solution in the class pool, having the object size and `mtable` identifier available to the compiler. However, the trap is still necessary as the static class initializer has to be executed exactly at this point, in order to fulfill the lazy class loading requirement.

At patch-time the class pool is queried for object size and `mtable` identifier, which maybe loads the class and triggers class initialization. After that, the object size is patched into the assembly stream, which is an argument to the memory allocation function. The `mtable` identifier is patched too, by adjusting a `mov`-instruction.

This trap is active at most once.

object fields The situation is similar to above: a field access requires an offset, thus requires the class to be loaded which would trigger class initialization.

This trap is active at most once.

static fields Static fields are allocated at class initialization. According to the JVM specification [LYBB13, § 5.5.], class initialization takes place on the first access of a class.

At code generation, an instruction sequence is emitted which will eventually cause a signal:

```
mov eax (Addr 0)
```

At patch-time, the address `null` is replaced with the actual memory location of the corresponding static field.

This trap is active at most once.

dynamic type check In the current state, MateVM does an inefficient type check in the run-time system rather than in native code. Traps are placed to call back to the run-time system and compute the type relation, cf. Section 5.3.

It does so, by patching a `mov`-instruction, which, depending on the result of the type check, copies 0 or 1 to the scratch register `eax` (representing the result as boolean).

This trap will not be patched away and is always active.

`ATHROW` Exceptions are handled by the run-time system, as presented in Section 4.4. When an exception occurs, the code generator places a trap in order to give control to the run-time system which handles the exception accordingly.

This trap will not be patched away and is always active.

`NullPointerException` Basically similar to the semantic of `ATHROW`, however it is never triggered explicitly. If a signal is not registered to a native code snippet, and it represents a signal of type `SIGILL`, it is recognized automatically as null pointer access. More details are discussed in Section 4.4.

After patching, the run-time system returns to the execution of native code and restarts the instruction that caused the signal.

Regarding the implementation: Handling hardware traps, or on operating system level *signals*, is not doable in pure Haskell at the moment. Albeit GHC is able to handle simple signals, access to the `ucontext_t` structure is needed in order to dump the contents of the register in the signal handler.

Discussion

Probably we could have used other techniques to switch back to the "Haskell World", but this approach has some advantages:

- The minimum byte sequence needed to cause a signal is two, namely `0xffff` which triggers `SIGILL`. After that, an arbitrary number of `no-op` instructions (`0x90`) can be placed in order to have the exact space needed for the instruction to be patched.
- Null dereferences can be placed intentionally (like done for virtual calls having unpatched `mtable` entries) and are cheap to handle. On the other hand, `NullPointerExceptions` can be handled fast and guarding certain operations with nullchecks is not required.

4.4 Exception handling

Exception handling itself is done at run-time, but it requires preparations at compile-time. First, it heavily depends on the used stack layout as *stack walking* is required for exceptions. Second, a special pointer, called `StablePtr`, is placed on the stack, which encapsulates necessary information as a Haskell data type. `StablePtrs` are provided by the GHC-run-time: If a `StablePtr` is requested for a value, the GHC-run-time assigns a unique number and maps it to the value. Although the value is a Haskell expression, the garbage collector of the GHC-run-time ignores this value. Later in time, one can obtain the value with the `StablePtr`. With this, arbitrary Haskell values can be saved in foreign storages, such as global variables in C or, as in the case of MateVM, on the machine stack.

The stack frame layout used for generated methods in MateVM is depicted in Figure 4.3. Basically, it represents the layout given by the `cdecl` calling convention with little modifica-

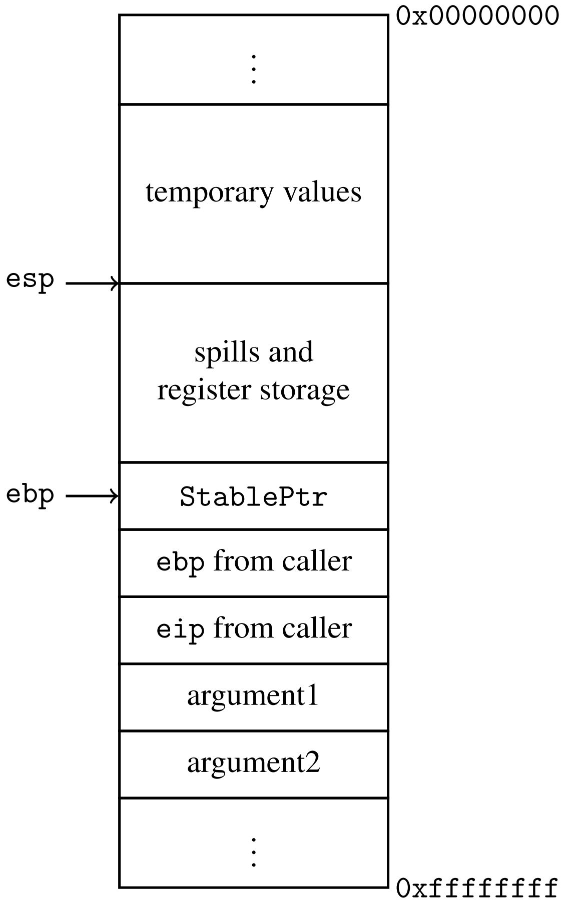


Figure 4.3: Stack frame layout of a generated method

tions. On the function prologue `ebp` is saved to the stack and also a copy of the `StablePtr` for this method. Then, `esp` will be assigned to `ebp` and `esp` will be updated according to the space requirements for spilled registers plus some slots for saving active hardware registers to a fixed location on the stack. The required space is computed by the register allocator, since it knows how much registers have to be spilled in this method.

The value of the stored `StablePtr` has the type `RuntimeStackInfo`. It contains

- the method name, mainly for debugging purpose at the moment. In future work, i.e. implementing a sophisticated base library, it is also needed for `StackTraceElement` in order to provide the user a correct stack trace.
- An exception-map, originally obtained from the class file stored in the exception-section. However, the given offsets refer to the program counter of the Java bytecode. Therefore, the back end has to update those values and map them to the hardware program counter of the target machine accordingly. This map maintains information necessary to find the appropriate handler for an exception at run-time.
- Savepoints for garbage collection, as described in Section 4.2.4.2

At run-time, the exception handling unit can be initiated by an `ATHROW`-instruction. In MateVM, `ATHROW` is realized by hardware traps as described in Section 4.3. When the trap handler identifies an `ATHROW`-trap, the exception, referenced by the pointer in `eax`, is inspected and handled as follows:

1. By observing the stack via `ebp`, the `StablePtr` can be obtained. The value referred by `StablePtr` contains a map of all possible handlers for this method. Let $\mathbb{H}$ be all handlers having the current `eip` in their responsible range. An element $x \in \mathbb{H}$ is a tuple, consisting of the type matching this handler and its entry point.
2. The run-time type of the exception stored in `eax` and the type of each handler in $\mathbb{H}$ are compared, using dynamic type checking. $\mathbb{H}$ has a certain order, representing the priority of each handler. That is, a handler for a more specific type regarding the class hierarchy, should match first than a handler for a more general type. On the first match, the algorithm continues with Case 4. If no handler matches, the stack is unwinded, as described in Case 3.
3. Unwinding the stack means, that the current stack frame is popped and `eip`, `ebp` and `esp` are restored, such as when the callee would return normally. Also caller-save registers have to be restored, which are stored at a fixed position in the stack frame. After that, Case 1 is considered again, as a different `StablePtr` is referenced now, thus a different set of handlers that could potentially catch the exception accordingly.
4. Having a matching handler, the handler can be executed by updating `eip` to the entry point of the handler. The trap handler is responsible to write back the updated registers to the processor and continues execution.

For certain cases, the virtual machine is responsible to create an exception object, for example `NullPointerException` or `ArithmeticException`. In such cases, the trap handler creates an object by first allocating memory for the object itself. Then the `<init>`-method is looked up, i.e. the constructor, of the target class and is called with the just allocated object (that is, execution of native code of this method, which can potentially has further side-effects such as exceptions).

Run-time exceptions

Some common run-time-exceptions are implemented, for example `NullPointerException`: MateVM uses two types of traps, namely `SIGILL` and `SIGSEGV` in generated code (cf. Section 4.3) Consider the following snippet:

```
mov 16(%eax), ebx
```

Suppose that `eax` contains an object reference and 16 is the offset to a field of that object. The JVM specification [LYBB13, § 2.4.], allows `null`-references, hence a nullcheck would be necessary before each field access. In fact, this check has to be done even before virtual calls, as they require an valid reference to `mtable`. If the check fails, an object of type `NullPointerException` has to be created.

However, nullchecks are expensive as they occur often [KP98]. One approach would be to eliminate nullchecks by static analysis, which would be easily doable with Hoopl. But, since MateVM already uses hardware traps, it can fall back to signals of the operating system and the target machine respectively, which signals the executed program with `SIGSEGV` if an invalid memory location is accessed, such as `0x00000000`. Therefore, nullcheck elimination is for free: The generated code does not have to check for null-references and the trap mechanism knows, if no other trap is registered for the program counter issuing the signal, it has to be a `NullPointerException`[5].

For further stability, MateVM should determine (like CACAO does [KP98]), which instruction or method caused the invalid access, thus determine if the signal originates from the Java program or something else. In the latter case, the virtual machine should propagate the signal to the operating system.

4.5 Example: A classic Fibonacci implementation

In this section the life cycle of a method through MateVM is demonstrated. The input as Java source code is shown in Listing 4.8. Figure 4.4a represents the output after using `javac`, that is Java bytecode. In Figure 4.4b the debug output after the front end is shown. Note the

```
        public static int fib(int n) {
                if (n <= 1) return 1;
                else        return fib(n - 1) + fib(n - 2);
        }
```

Listing 4.8: A naive implementation of Fibonacci

[5]furthermore, it can distinguish between the different types of signals

usage of virtual registers as described in Section 3.6. Also, as the actual register assignment has not happened yet, to be saved registers are not determined yet, represented by the empty list in `call preps` constructor. The pattern `Add <reg0>, <reg1>, 0x00000000` represents a copy instruction. This substitution is useful for analysis, since it is covered by the general case (the add operation).

Consider the first line of Figure 4.4b: The register copy could be avoided and all future uses of the temporary register 50000 should be replaced with the source, the virtual register 200000. Unfortunately, since `MateIR` is not in static single assignment form, this kind of transformation cannot be done safely in every case. Apart from that, as already discussed in Section 3.6, the copy elimination should be done beforehand.

```
0:    iload_0
1:    iconst_1
2:    if_icmpgt       7
5:    iconst_1
6:    ireturn
7:    iload_0
8:    iconst_1
9:    isub
10:   invokestatic    #2; //Method fib:(I)I
13:   iload_0
14:   iconst_2
15:   isub
16:   invokestatic    #2; //Method fib:(I)I
19:   iadd
20:   ireturn
```

(a) Java bytecode of `fib()`

```
L1:
  Add   JInt(50000), JInt(200000), 0x00000000
  if (0x00000001 ‘C_GT‘ JInt(50000)) then L2 else L3
L2:
  Add   JInt(50001), JInt(200000), 0x00000000
  Sub   JInt(50002), 0x00000001, JInt(50001)
  call preps (SaveRegs): []
  push(0) JInt(50002)
  invoke RTCall(02) (Just JInt(99999)) [CallStatic]
  call preps (RestoreRegs): []
  Add   JInt(50003), JInt(99999), 0x00000000
  Add   JInt(50004), JInt(200000), 0x00000000
  Sub   JInt(50005), 0x00000002, JInt(50004)
  call preps (SaveRegs): []
  push(0) JInt(50005)
  invoke RTCall(02) (Just JInt(99999)) [CallStatic]
  call preps (RestoreRegs): []
  Add   JInt(50006), JInt(99999), 0x00000000
  Add   JInt(50007), JInt(50006), JInt(50003)
  return (Just JInt(50007))
L3:
  return (Just 0x00000001)
```

(b) `MateIR` debug output

Figure 4.4: Java bytecode and representation in `MateIR` before linearisation

Next, Listing 4.9 shows the intermediate representation after linearisation and register allocation. Note that Hoopl put the basic block `L3` directly after `L1`. All virtual registers are replaced by hardware registers or spill locations. Otherwise no optimizations, such as dead-code elimination, are done, thus the structure remains the same.

Finally, the back end emits native code, as shown in Listing 4.10. As `cdecl` is used, the function layout should be familiar to the reader: Line 1–7 is the method prologue, slightly modified due to the use of the `StablePtr`. Block `L1` represents the comparison used to determine the base case, while block `L3` is the base case itself, returning 1 plus the function epilogue. From line 23–36 the first call to `fib(n-1)` is done, followed by the second call to `fib(n-2)` in line 37–50. Note that the result of `fib(n-1)` is saved to the stack in line 36. In the block from line 52–54 the results from both method calls are added, and finally returned to the caller in 56–63.

```
L1:
  Add      ecx, 0x0c(ebp), 0x00000000
  if (0x00000001 `C_GT` ecx) then L2 else L3
L3:
  return (Just 0x00000001)
L2:
  Add      ecx, 0x0c(ebp), 0x00000000
  Sub      edx, 0x00000001, ecx
  call preps (SaveRegs): [(ecx,JInt),(edx,JInt)]
  push(0) edx
  invoke RTCall(02) (Just eax) [CallStatic]
  call preps (RestoreRegs): []
  Add      0xffffffe8(ebp), eax, 0x00000000
  Add      ecx, 0x0c(ebp), 0x00000000
  Sub      edx, 0x00000002, ecx
  call preps (SaveRegs): [(ecx,JInt),(edx,JInt)]
  push(0) edx
  invoke RTCall(02) (Just eax) [CallStatic]
  call preps (RestoreRegs): []
  Add      edx, eax, 0x00000000
  Add      ecx, edx, 0xffffffe8(ebp)
  return (Just ecx)
```

Listing 4.9: Representation after linearising and register allocation

```
  push ebp
  push 21H      # StablePtr
  mov ebp, esp
  sub esp, 1cH
  push ebx      # callee saved registers
  push esi
  push edi
L1:
  mov ecx, dword ptr [ebp+12]
  cmp ecx, 1H
  jg  L2
  jmp L3
L3:
  mov eax, 1H   # epilogue for 'return 1'
  pop edi       # restore callee saved regs
  pop esi
  pop ebx
  mov esp, ebp
  pop ebp       # destroy StablePtr
  pop ebp
  ret
L2:
  mov ecx, dword ptr [ebp+12] # copy arg1
  mov eax, ecx
  sub eax, 1H
  mov edx, eax
  mov dword ptr [ebp-4], ecx  # save regs
  mov dword ptr [ebp-8], edx
  push edx                    # push arg
  # trap for static call
  (invalid opcode, byte=255)
  nop
  nop
  nop
  add esp, 4H
  mov dword ptr [ebp-24], eax # store result
  mov ecx, dword ptr [ebp+12] # copy arg1
  mov eax, ecx
  sub eax, 2H
  mov edx, eax
  mov dword ptr [ebp-4], ecx  # save regs
  mov dword ptr [ebp-8], edx
  push edx                    # push arg
  # trap for static call
  (invalid opcode, byte=255)
  nop
  nop
  nop
  add esp, 4H
  mov edx, eax

  mov eax, dword ptr [ebp-24]
  add eax, edx                # add results
  mov ecx, eax

  mov eax, ecx                # epilogue
  pop edi
  pop esi
  pop ebx
  mov esp, ebp
  pop ebp
  pop ebp
  ret
```

Listing 4.10: Emitted Code

CHAPTER 5

Run-time environment

One of the main differences between a static compiler and a dynamic compiler is, among others, to shift certain parts from compile-time to run-time[1]. For example, the JVM specification requires to be able to load classes at run-time[2], instead of loading all classes at startup of the virtual machine. In this chapter we present what happens on class loading, how a method lookup, dynamic type checking, string allocation and garbage collection works.

5.1 Classpool

This part of MateVM is responsible for everything regarding Java classes. It serves as a database for field- and method-offsets. Also, it loads classes and interfaces if requested. Loading a class could be triggered by:

- method lookup (see Section 5.2)
- object creation
- static or object field access
- dynamic type check

All those events require to execute the static initializer of the target class. Interfaces are loaded by loading a class implementing this interface or by resolving a super interface.

A class file is searched in all given class paths. Before the class itself is inspected, the super class of it is resolved recursively, until it loads `java/lang/Object`. Afterwards all interfaces are loaded that are implemented by this class. Loaded classes or interfaces are cached by the classpool in a map as Haskell values, in order to avoid filesystem access on further requests to this class (e.g. on extracting the code segment when compiling a new method of this class). Finally, the class itself can be used to build the following lookup-tables:

[1]strictly speaken, everything is run-time in a dynamic compilation system

[2]in fact it even requires to be able to *replace* loaded classes. MateVM does not support this feature yet

static fields When calculating offsets for static fields, static fields of the super class have to be respected too. Thus, the classpool duplicates the lookup-table of the super class. The classpool allocates memory for each new field entries and stores the address in the lookup-table for each offset.

Note, that the requirement of the Java language specification [GJS+13, Example 8.3.1.1-2], i.e. static fields could be overridden by inheritance, has to be respected.

object field offsets Object fields are similar to static fields, except that no absolute addresses are stored, but offsets, as they are per object.

method offsets As Java implements single inheritance of classes, method offset generation is similar to field offsets. As calculated method offsets can be shared across different objects of this class, a so-called *method table*, or short `mtable`, is allocated. On object creation, the address of the `mtable` is stored as the first entry in the object layout (cf. Figure 4.1 on Page 33). Note, that the offset is only unique *locally* with respect to its class hierarchy, which differs to offsets for interface methods.

interface method offsets These offsets are generated by loading interfaces. When loading an interface, each method will be assigned to a *globally and unique offset*. A globally unique mapping is needed, in order to have consistent offests across different classes which implement the *same* interfaces. The uniqueness requirement is an implication of multiple inheritance of interfaces.

On class initialization, memory is allocated for a table according to the current size of the global offset map. This table is called *interface method table*, or short `itable`. The global offset map guarantees that it contains all offsets needed by the class and therefore is big enough (a precondition is, that all implemented interfaces are loaded at this point, therefore the exact size can be obtained). In fact, the table will contain many slots for interface methods never used by this class, as some interfaces are not even implemented by it, but already added to the global offset map. As a result, interface method tables could be large and reserve unnecessary memory. There are techniques addressing this issue [ACFG01].

An advantage of our approach is that we can skip an expensive dynamic type check on `invokeinterface`, thus having a rather efficient and simple implementation (see Section 4.2.4).

Each offset, when mapped (see below), contains a pointer to the implementing method, as depicted in Figure 4.1 on Page 33.

Pointers in the method table and interface method table are initialized with null pointers, thus forcing a segmentation fault (also known as `SIGSEGV`) on access. This signal is captured by the trap handler, causing a method lookup and patching the table entry accordingly (cf. Section 4.3).

Figure 5.1 depicts the memory layout of object `o` after the execution of `main` from the example listed in Listing 5.1[3]. `hashCode()`, `notify()` and so forth does not have a reference

[3]addresses can vary due to different machines and address space layout randomization

```
public class Example extends A {
  public static void main(String []args) {
    Example o = new Example();
    o.method1();
    o.method2();
    for (int i = 0; i < 2; i++)
      ((I2) o).method1();
    ((I1) o).method1();
    ((I2) o).method2();
    System.out.printf("o.f1: %d\n", o.f1);
  }
}

class A implements I1, I2 {
  int f1;
  public void method1() { System.out.printf("method1\n"); }
  public int  method2() { f1 += 2; return f1; }
}

interface I1 {
  void method1();
}

interface I2 {
  void method1();
  int  method2();
}
```

Listing 5.1: Example with interfaces

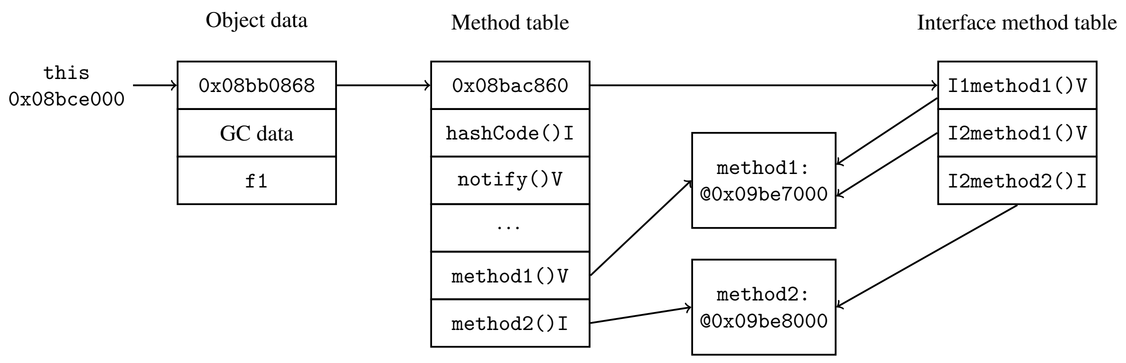


Figure 5.1: Memory layout of object o after execution (cf. Listing 5.1)

to an actual piece of code, since they were never executed and therefore not compiled to native code. The method references of the interfaces are resolved to the same code blob as the class methods itself (i.e., no code duplication). The linkage is done at run-time, for example consider `I2method1`: At code generation (compile-time) a trap is placed at line 7 of Listing 5.1. When the trap occurs, the trap handler knows the type of the call and information about the called method. It then consults the methodpool (see Section 5.2) for the entry point of the method and patches

- the entry point at the correct offset in the `itable`
- the call instruction with the correct offset into the `itable`

Loading the `itable` into the right register is already handled by the code generator. After the trap handler has done its work, execution restarts at the program counter which caused the trap, that is now a call after patching.

On the second iteration at line 7 the trap is removed due to the patching above. Also, the entry in the `itable` is patched accordingly, therefore the method call is done without any interaction of the run-time system.

Assuming a different object would pass this loop (not the case in this example), and it has not a patched `itable` yet, it would also trigger a signal and would be handled by the trap handler.

Security concerns

The segmentation fault would not occur at `0x00000000`, which is not mapped on any sane system nowadays, but at `0x00000000`$+i_{\text{offset}}$ (in this case being $i_{\text{offset}} = 0x4$). Apart from security implications, this could lead to an unwanted faulty behaviour when hitting a valid memory segment if i_{offset} is big enough.

5.2 Methodpool

If the run-time system looks up a method, it consults the so-called *methodpool*. It maintains a map of already compiled methods and its entry points. If the requested method is not in the map, the methodpool starts the compiler for it. After compilation, it stores the result (that is, the entry point) into the map. Note, that the key of the mentioned map is the full name of the method, i.e. the class name, method name and its type signature (same notation as used by the JVM specification [LYBB13, § 4.3.2.]) encoded as a byte string.

5.2.1 Native methods

The methodpool is also responsible to resolve *native* methods. In Java a programmer can declare a method as `native` and omitting the method body, i.e. no code section is provided via the class file. The virtual machine has to find a proper method implementation for it. For example basic IO-operations are native, since they need to access interfaces of the operating system. In MateVM native methods can be implemented in C or Haskell:

C Dynamic look up is done with a Haskell package called `plugins` [Ste12]. It can determine a symbol (and hence its entry point) in an object file and link it to the current executed program. However, the operation itself costs about $300ms$ which conflict with the requirement of a fast startup. Therefore, it is suggested to implement native methods critical to execution time with the Haskell variant.

Haskell Functions in Haskell can be exported with FFI, therefore they are also visible from the outside of Haskell and are callable with the used calling convention. Further, it is possible to obtain the entry point of such a function at compile-time [Ove12a]. Those functions must be declared as `IO` and should use C types as provided by the FFI module.

Since both variants can obtain the entry point of a method, the same techniques for code patching applies as for normal methods, just the lookup is different. Due to the same calling convention for methods emitted by MateVM and the operating system, no property mismatch occurs.

The difference between the two presented approaches is, that native methods implemented with Haskell are compiled and linked to the MateVM-executable. With the C variant, one has to provide an object file (`native.o`), which can be exchanged without recompilation of MateVM. Therefore, the C variant will be interesting for adding support for the Java Native Interface [Mic03]. MateVM does not implement JNI yet, which simplifies the current native interface a lot.

5.3 Dynamic type checking

In order to determine a subtype-relation between two classes or interfaces, a *hierarchy* has to be built first, which represents the relations between classes or interfaces. On class or interface loading, the class pool adds an entry to the global hierarchy data structure. The *class* hierarchy consists of a map, where the key represents the `mtable`-pointer and the value its super class (again as `mtable`-pointer) and all implementing interfaces as a list of byte strings. The *interface* hierarchy is a map, with a byte string as key (representing the interface name) and all super interfaces as list of byte strings as value.

The transition between native code and run-time system is already described in Section 4.3. The run-time system queries the class hierarchy with the object reference and the given class or interface to examine, extracted from the `INSTANCEOF`- or `CHECKCAST`-instruction.

Since Java has single inheritance for classes and multiple inheritance for interfaces, two cases have to be distinguished:

Class Let m_{obj} be the `mtable`-pointer of the given object. First the `mtable`-pointer m_{query} of the requested class name is obtained (the classpool maintains a map for that). Then, a function compares m_{obj} with m_{query}. On a match, the subtype-relation is given and the algorithm terminates successfully. Otherwise, the super class of m_{obj} is obtained and the function calls itself recursively. If no type match occurs, the algorithm will eventually end up with `java/lang/Object` as m_{obj}, which aborts the algorithm and it returns `false`, meaning no subtype-relation is given.

Interface Let $\mathbb{I}$ be the set of implementing interfaces of the given object. i_{query} is the requested interface name. A function checks if $i_{\text{query}} \in \mathbb{I}$. If this operation succeeds, the subtype-relation is `true`. Otherwise, the super interfaces of all interfaces in $\mathbb{I}$ are obtained from the interface hierarchy and added to $\mathbb{I}$, while the already checked interfaces are removed from $\mathbb{I}$. The algorithm terminates with `false`, if $\mathbb{I}$ represents the empty set.

Although, the module operates functional correct and is easily implementable in Haskell, the implementation is rather inefficient regarding execution time (cf. Section 7.2): The class or interface hierarchy is iterated on every query. Even worse, the algorithm needs rather long to determine that no subtype-relation for the given query exists. Likewise, the used data structure is bloated, as the presented algorithm operates on interface names. Another issue is, that class reloading is not considered at all.

For future work we suggest to implement displays [Coh91, BEK09], as the implementation effort is moderate and is suitable for native code since the required data structure is simple. Also, Vitek et al. [VHK97] presents sophisticated algorithms for dynamic type checking.

5.4 Handling of `String` literals

`Strings` have a rather unique behaviour in the Java language specification [GJS+13, §3.10.5]. Consider the example in Listing 5.2. Usually, in order to compare the content of two objects, `equals()` is used in Java. Although `Strings` are regular objects in Java, they can be defined at compile-time, i.e. as a `String` literal. Further, the specification requires that two

```java
public class Stringhandling extends {
  public static void main(String []args) {
            String a = "abc";
            String b = "abc";
            if (a == b)
                    System.out.printf("a and b are equal\n");
            else
                    System.out.printf("never happens\n");
  }
}
```

Listing 5.2: Behaviour of interned `Strings` in Java

`String` literals, describing the same content, must be the *same* object.

In MateVM, this is solved via a map: As key the `String` content is used, and the value represents the reference to the object. If the run-time system requests a yet unknown `String`, it has to be allocated and added to the map.

5.5 Garbage Collection

As mentioned in Section 1.4 Steinlechner [Ste13] contributed two garbage collectors to MateVM, which are described shortly in this section. Which garbage collector is used, is controlled by the compile-time-option

```
usePreciseGC :: Bool
```

in `./Compiler/Mate/Runtime/RtsOptions.hs`.

BoehmGC
The Boehm-Demers-Weiser garbage collector [Boe13] is conservative, and can be used as transparent replacement for `malloc(3)`. Hence, `free(3)` is not necessary explicitly anymore, therefore matching the semantics of the Java language [LYBB13, § 2.5.3.].

BoehmGC is implemented in C, thus, to be usable from Haskell, bindings via FFI had to be done. The interface to MateVM is rather easy, since it already used `malloc(3)`[4] for allocating memory for objects and arrays.

Exact garbage collection
Being a conservative garbage collector has of course drawbacks, e.g. memory leaks occurs by design. Exact garbage collection requires a so-called *root-set* containing references, which is represented by savepoints, as already discussed in Section 4.2.4.2. Simplified, on a garbage collection, it builds an object graph using the root-set combined with stack walking and marks all references reachable from that root-set. All marked objects are copied to a new heap space. The old heap space is omitted, thus all unreferenced objects are implicitly deallocated.

From an implementation point of view, everything is done in Haskell, that is, no FFI is required. It introduces a type class `RefObj`, which essentially represents functions that can be used to operate on a single reference in the object graph. The actual module relies heavily on `IO`, but since the graph is realized as a type class, it can be tested with pure code, resulting in more reliable tests.

[4]but without deallocating memory afterwards

CHAPTER 6
Pearls

In this section we want to highlight some parts of MateVM, which shows the strength of implementing a virtual machine in Haskell. We already covered some functional pearls in the previous chapters, such as Hoopl in Section 3.3 or `RankNTypes` in Section 4.2.2.

6.1 Register allocation and QuickCheck

Section 3.6 presents register allocation. In order to do the actual allocation, live ranges are needed. From that, a register assignment can be computed. The following property can be defined:

Property 1. *For all virtual register there is no other virtual register that has an overlapping live range and the same hardware register assigned.*

In the debug build of MateVM we check the result of the register allocator with this property—Property 1 is realized with the function

```
noLiveRangeCollision :: LiveRanges -> RegMapping -> Bool
```

QuickCheck [CH00] enables property checking on randomly generated test cases. In order to enable random generation, QuickCheck defines a type class `Arbitrary`, that must be implemented for the desired data type. We have to specify certain boundaries and guide the library how to create an instance. For example, we can define a range of how many virtual registers should a `LiveRange` contain. Also, a virtual register can be guarded by conditions, e.g. the end of a live range must be *after* the start of it. `./Compiler/Mate/Frontend/RegisterAllocation.hs` specifies an instance of `Arbitrary` for `LiveRange`.

Now, QuickCheck can be used to do property checks:

- QuickCheck generates a random `LiveRange` instance
- LSRA is used to obtain a register mapping (i.e., from virtual register to hardware registers)
- verifies the register mapping with Property 1

If a random instance generated by QuickCheck fails, the malicious test case is printed and can be analyzed. Otherwise, this procedure is repeated hundred times per default.

In order to raise the chance of finding a failing test case, the QuickCheck test is included in the build bot, which builds `HEAD` from the repository every night. So far, the only bug found was introduced by refactoring, yielding an `undefined` value due to a missing preprocessor definition.

6.2 Evolution of code patching in MateVM

The presented trap mechanism (cf. Section 4.3) requires code patching.

6.2.1 First generation: C

In the early stages of MateVM, code patching was done in C, cf. Listing 6.1. Note, that `getMethodEntry` is a Haskell function called with FFI, which knows about registered traps, that could be looked up via the current instruction pointer. This method requires knowledge about `x86`-opcodes and how instructions are encoded.

```c
void staticcalltrap(int nSignal, siginfo_t *info, void *ctx) {
  mcontext_t *mctx = &((ucontext_t *) ctx)->uc_mcontext;
  unsigned int eip = (unsigned int) mctx->gregs[REG_EIP];
  unsigned int *to_patch = (unsigned int *) (eip + 1);
  if (*to_patch != 0x909090ff) {
          dprintf("callertrap: something is wrong here. abort\n");
          exit(0);
  }
  /* getMethodEntry queries the method pool */
  unsigned int patchme = getMethodEntry(eip, 0);

  unsigned char *insn = (unsigned char *) eip;
  *insn = 0xe8; /* call opcode on x86 */
  *to_patch = patchme - (eip + 5);
  mctx->gregs[REG_EIP] = (unsigned long) insn;
}
```

Listing 6.1: First generation: Code patching in C for a static method invocation

6.2.2 Second generation: Haskell and pointers

Later on, patching was shifted to the Haskell world (see Listing 6.2), but basically it remains the same: It still requires low-level knowledge about the target architecture. The dirty work, such as obtaining registers or register write back, is still done in C.

```haskell
staticCallHandler :: CPtrdiff -> IO CPtrdiff
staticCallHandler eip = do
  let insn_ptr = intPtrToPtr (fromIntegral eip) :: Ptr CUChar
  let imm_ptr = intPtrToPtr (fromIntegral (eip - 1)) :: Ptr CPtrdiff
  checkMe <- peek imm_ptr
  if checkMe == 0x909090ff then
    do
      entryAddr <- getMethodEntry eip 0
      poke insn_ptr 0xe8 -- call opcode
      poke imm_ptr (entryAddr - (eip + 5))
      return (eip - 2)
    else error "staticCallHandler: something is wrong here. abort\n"
```

Listing 6.2: Second generation: Code patching in Haskell for a static method invocation

6.2.3 Current generation: Patching via the `CodeGen`-monad

In the current implementation, the `CodeGen`-monad is used a second time, namely at patch-time. Consider the example in Listing 6.3: `girStatic` emits the code for a static call. In line 8 a sequence is generated, that eventually triggers `SIGILL`. After that, line 9 defines a *closure* named `patcher`, that is actually a `CodeGen`-monad for itself. Instead of storing relevant information for the trap at this point, the unevaluated closure is stored in the trap map (line 23). At patch-time, the trap handler executes `patcher`, using the current instruction pointer as code buffer. Thus, `patcher` overrides the trap with its emitted code.

This method is more elegant, since compile-time and patch-time are sticked together at a common part in the source code.

6.2.4 Future generation: Trap combinator for the `CodeGen`-monad

The next step would be to introduce a combinator, that places a trap for a certain region. In the current method, knowledge about the needed space for a trap is necessary. With a combinator, that could be determined automatically. The source code would be like depicted in Listing 6.4, that is a sequence of monadic action is guarded by a trap. The body of the guard is executed not at compile-time, but at patch-time when the placed trap occurs. The body is used to patch at the given program counter.

Emitting the trap with enough padding and storing the `patcher` into the map, is hidden by the combinator `guardWithTrap`.

```haskell
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
        call $ fromIntegral (entryAddr - ((wbr M.! eip) + 5))
        return wbr
  setGCPoint mapping
  let argcnt = methodGetArgsCount (methodNameTypeByIdx cls cpidx)
                  * ptrSize
  when (argcnt > 0) (add esp argcnt)

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

Listing 6.3: Third generation: Code patching via `CodeGen`-monad

```haskell
girStatic cpidx haveReturn ct mapping = do
  {- ... -}
  newNamedLabel (show l) >>= defineLabel
  guardWithTrap $ \wbr -> do
    entryAddr <- liftIO $ lookupMethodEntry l
    call $ fromIntegral (entryAddr - ((wbr M.! eip) + 5))
  setGCPoint mapping
  {- ... -}
```

Listing 6.4: Future generation: combinator in `CodeGen`-monad

CHAPTER 7

Results

7.1 Benchmarks

In order to show the state of the current implementation of MateVM, we present some benchmarks and compare performance with mainstream JVMs. Unfortunately, MateVM does not have a sophisticated base library like GNU Classpath yet, therefore we cannot fall back on elaborated benchmarks such as SPEC JVM98 [Cor98]. Hence, we implemented some synthetically benchmarks. The benchmarks are:

HelloWorld The classic "Hello World" example. This instance is used as baseline, as the other virtual machines uses full-fledged base libraries, and therefore have more initial compile-time or interpreting overhead as MateVM. We subtract the result of this benchmark to get an approximate value of the actual execution time, in order to get a fairer comparison (the following benchmarks does not rely on the base library—except `BenchException`).

Fib A naive implementation of Fibonacci calculating the 37th number.

BenchObjectfield Stressing field access (read and write) of an object in a loop, with two reads and one write per iteration.

BenchStaticfield Stressing field access (read and write) of a class in a loop, with two reads and one write per iteration.

BenchVirtual An example using inheritance for classes and overloads methods. In a loop we call virtual methods in order to examine dynamic dispatch.

BenchInterface Testing `invokeinterface`, with two objects of different classes but implementing a common interface. We then cast them to the same interface. In a loop, we call for both objects the same methods, but, since different implementations in the class, each method refers to a distinct method which the JVMs have to resolve at run-time.

BenchInstanceOf Queries the JVM about class relations in a moderate class hierarchy (eight interfaces and four classes).

BenchArray This benchmark allocates an array with `0x100` elements and initialises it with "random" numbers in a loop. Then, it calculates some kind of sum in a long loop, which includes four array reads and two writes each iteration.

BenchException A nested method hierarchy is called in a loop. In the leaf method an exception is thrown, which is propagated to the `main`-method that has appropriate handlers to catch the exception.

BenchCompiletime An object is allocated, that has 7885 field members which are initialized in the object constructor. However, only two instance variables are accessed in `main`.

The source code of each benchmark can be looked up in the repository, under the directory `./tests/`.

Execution time and memory usage are measured. The command-line utility `time(1)` is used for both tasks, as this is the most generic way to do. While execution time is straight forward, memory usage has its difficulties. The so-called *maximum resident set size* is measured, that is the amount of memory held in RAM of the executing machine. While it is not exact, it has some advantages regarding garbage collection: Usually a garbage collector allocates an initial heap on start up[1]. In the presented test cases, this heap will never be full as the memory requirements are low. Fortunately, the operating system uses lazy page loading, therefore a page is only installed on the first access—thus a page counts to the resident set size on the first access.

Others then that, `time(1)` reports memory in the wrong unit [Pro10] which is respected in the presented results. In order approve the measured results, we analyzed the GHC heap report for MateVM that roughly matches the output by `time(1)`.

How we measure a JVM with a test case

The script `./tools/bench.sh` tests a single test case by first executing it three times in order to fill the file caches of the machine. After that, the test case is executed twelve times, and the best three results are taken. From that, the average is calculated.

Builds of JVMs used

We obtained all JVMs from the Ubuntu 12.04 repositories.

```
java version "1.6.0_24"
OpenJDK Runtime Environment (IcedTea6 1.11.5) (6b24-1.11.5-0ubuntu1~12.04.1)
OpenJDK Server VM (build 20.0-b12, mixed mode)
OpenJDK Client VM (build 20.0-b12, mixed mode, sharing)
CACAO (build 1.1.0pre2, compiled mode)
JamVM (build 1.6.0-devel, inline-threaded interpreter with stack-caching)
```

[1]note, that MateVM runs in fact two GCs: One for the Haskell run-time and one for Java

Regarding CACAO, the wall times were taken with the Ubuntu build in Table 7.1. In order to obtain compile-time in Table 7.3, the option `-enable-statistics`[2] is necessary. This option is not available in the Ubuntu build, therefore a custom build had to be done.

Benchmark machine

The measurements were performed on an Intel Core i7 920 2.67 GHz equipped with 8 GB RAM available. As operating system Ubuntu 12.04 LTS 32 bit is used. Regarding MateVM, GHC 7.4.1 is used, compiled with `-O2 -fasm -static`. See the `Makefile`-target `mate.opt` for the specific build flags.

7.2 Execution time

The wall time is measured in Table 7.1, however `HelloWorld` is used as baseline, that is 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,

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

Table 7.1: Measurements for execution time

MateVM cannot compete with the other just-in-time compilers. Then again, `BenchVirtual` and `BenchInterface` are handicapped by the rather bad layout for `mtable` and `itable`, which simplifies things but requires more dereferencing on a virtual dispatch.

`BenchInstanceOf` is particular bad regarding performance since it involves explicit dynamic type checking, as already discussed in Section 5.3.

In `BenchCompiletime` the compiler is stressed. We analyzed compile-time in more detail for MateVM in Table 7.2. The values are obtained by enabling

```
mateTIME :: Bool
```

in `./Compiler/Mate/Debug.hs`. As a comparison, the compile-time results for CACAO are shown in Table 7.3 for some benchmarks, obtained via `-time`. They are varying between

[2]this enables the `-time` flag on command-line

benchmark	wall time	compiler	amount in %
HelloWorld	8.00ms	5.33ms	66.66%
Fib	474.70ms	2.67ms	0.56%
Objectfield	897.39ms	4.00ms	0.45%
Staticfield	832.05ms	5.33ms	0.64%
Virtual	4976.31ms	5.33ms	0.10%
Interface	621.37ms	5.33ms	0.86%
InstanceOf	1401.42ms	10.67ms	0.76%
Array	1610.77ms	4.00ms	0.25%
Exception	458.70ms	6.67ms	1.45%
Compiletime	956.06ms	757.38ms	79.21%

Table 7.2: Execution time and compile-time for MateVM

20ms and 56ms for CACAO, *regardless* of the benchmark. Note that the startup-time is included, i.e. the compile-time for `BenchCompiletime` is negligible compared to MateVM.

We hope, that performance of the compiler can be improved in future work (cf. Chapter 9), although due to Haskell being a high-level language, it will probably never get faster than CACAO.

benchmark	wall time	compiler	amount in %
HelloWorld	128ms	40ms	31.25%
Exception	320ms	44ms	13.75%
Compiletime	196ms	40ms	20.41%

Table 7.3: Execution time and compile-time for CACAO

7.3 Memory usage

As expected, memory consumption does not vary that much between the benchmarks. From the differences between some test cases, we can conclude memory consumption for the compiler, e.g. consider `BenchCompiletime` at MateVM. We can estimate, that the compiler needs about 30MB to compile the method. Unfortunately, lazy programs are known for unpredictable memory behaviour. Furthermore, MateVM does not use strictness annotations for its data structures yet, supporting bad memory usage.

On the other hand, the garbage collector of the GHC-runtime is not triggered yet, therefore making use of the available heap. The option for the garbage collector can be tuned if memory usage is a problem, for example if the maximum heap size is restricted to 15 MB, the benchmark runs as well, but the garbage collector must be called more often.

This observation is also true for the garbage collectors of the JVMs: By tuning those parameters, different results could be achieved as well. We used the default configuration for each JVM, cf. Table 7.5.

benchmark	server	client	cacao	mate	jamvm
HelloWorld	10.22MB	11.39MB	12.66MB	5.88MB	7.88MB
Fib	11.57MB	11.63MB	12.83MB	5.96MB	8.00MB
Objectfield	11.84MB	11.66MB	12.84MB	5.99MB	8.01MB
Staticfield	11.84MB	11.67MB	12.84MB	5.96MB	8.01MB
Virtual	11.93MB	11.69MB	12.86MB	6.00MB	8.02MB
Interface	12.26MB	11.72MB	12.89MB	6.20MB	8.06MB
InstanceOf	10.37MB	11.76MB	12.92MB	6.45MB	8.11MB
Array	12.18MB	11.66MB	12.84MB	6.05MB	8.01MB
Exception	20.67MB	15.92MB	14.41MB	7.49MB	16.62MB
Compiletime	12.23MB	13.42MB	17.07MB	36.12MB	10.05MB

Table 7.4: Measurements for memory usage

JVM	stack	heap start	heap max
server	320kB	16MB	512MB
client	320kB	16MB	512MB
cacao	64kB	2MB	128MB
jamvm	64kB	1MB	16MB

Table 7.5: Memory Configuration of JVMs

The memory footprint is unexpected low (roughly 6MB): Keep in mind, that for executing a Haskell program, a run-time system for it is necessary. Furthermore, MateVM is still missing a sophisticated base library mainly the reason for "low" memory usage. Consider

```
$ java -cacao -verbose:jit tests/HelloWorld | \
    grep -c 'Generating code done'
1124
```

A classic `HelloWorld`-program requires 1124 methods to be compiled. This is clearly not the case with the minimalistic base library used in MateVM:

```
$ mate tests/HelloWorld # generates logfile "mate.log"
$ grep 'Jit: emit code' mate.log
7
```

Therefore resulting in lower absolute memory consumption.

CHAPTER 8

Related Work

This thesis is influenced by research from the Haskell and JVM community.

8.1 Kafka: Just-In-Time Compiler for a Functional Language

Grabmüller implemented the prototype *Kafka* for his thesis [Gra09]. Kafka is a just-in-time compiler for a fictive functional language. Kafka itself is implemented in Haskell, hence being the first just-in-time compiler implemented in Haskell known to the author. For run-time code generation it uses Harpy (Grabmüller is the author of Harpy [GK07]). In order to store references on the machine stack, it also uses `StablePtrs` as presented in Section 4.4.

Callbacks into the run-time system are solved differently: All run-time related functions are exported via FFI, therefore can be called by native code (such as done for calls into garbage collection in Section 4.2.3). If on a certain point run-time services are needed, a call to it is emitted. At run-time, the run-time service can patch itself away, e.g. an eval-node triggers evaluation, that is the compiler, of a part of the program. After compilation, it replaces the call into the run-time system with a call to the generated function via code patching (similar to the "second generation" Section 6.2.2).

We think that traps as used in MateVM, would make the implementation of Kafka simpler and more readable. However, it has an entirely different execution model, since the input language is of an other language paradigm. The source code of Kafka is not publicly available, but can be obtained by asking the author of Kafka friendly.

8.2 Lambdachine: A Trace-Based Just-In-Time Compiler for Haskell

Schilling implemented a trace-based just-in-time compiler for Haskell [Sch12, Sch13]. Lambdachine is split into two parts:

- In order to have an efficient execution, the Haskell source code is first compiled to a suitable representation for tracing, that is bytecode. It uses the GHC front end to extract the `Core`-representation of an given Haskell file. The compiler translates the `Core` input to its own bytecode, which is used for static analysis and then for register allocation. Lambdachine uses Hoopl as well: Linearisation, as presented in Section 3.4, was inspired by Lambdachine. Hence, it uses the linearised representation for register allocation similar to the presented way in Section 3.6. It is the only Hoopl-client in the wild[1] known to the author.

- A virtual machine, implemented in C++, which executes the generated bytecode files. Since only traces have to be compiled, code generation is simpler than a regular compiler.

8.3 JVMs ported to `x86`

`x86` is known to be an unpopular target for compiler writers (e.g., [ABC$^+$02, Section 3.1]), because it has a non-orthogonal instruction set and is not a load/store architecture. Unfortunately, due to Harpy, MateVM is restricted to `x86` at the moment.

The first target of CACAO was DEC Alpha [Kra98], but `x86` was implemented years later [Tha04]. An other example is the Jikes RVM, which reported their experiences with `x86` regarding the virtual machine in [ABC$^+$02]—the first platform used for Jikes RVM was PowerPC on AIX.

8.4 Just-In-Time Compiler Techniques

Regarding used techniques, MateVM is inspired by CACAO, due to its simplicity. The presented design of traps in Section 4.3 is similar to the implementation of CACAO. CACAO solves `mtable` and `itable` more efficient [BEK09], but the idea of having tables and therefore be able to omit dynamic type checking on an invocation, is the same for MateVM. MateVM implements some ideas from the front end of CACAO, for example interface registers on basic block analysis (cf. Section 3.2.2 [Kra98]).

[1] apart from GHC

CHAPTER 9

Conclusion and Future Work

We have shown that Haskell is suitable for implementing a just-in-time compiler. The strength of Haskell is certainly the explicit seperation of pure and effectful code, which is helpful on designing a complex project. Hoopl turned out to be a powerful framework, that enables further opportunities to integrate optimization passes easily. An other example is the `CodeGen`-monad, which enables a nice level of abstraction to generate native code in a type-safe manner.

The results evidently shows that MateVM is not faster than other just-in-time compilers, but we are satisfied with overall performance. Although performance can be improved, Haskell is a high-level language after all, therefore we could not expect a faster implementation than JVMs in C/C++/etc in the first place. On the other hand, we think that MateVM is "better" on a source code level, albeit this is the subjective opinion of the author and impossible to prove.

We are looking forward to further contributions to MateVM.

Future Work

We have two primary goals for MateVM, with the constraint to be implemented in Haskell:

1. Implement the JVM specification [LYBB13]
2. Implement a fast virtual machine, but keep complexity low

Both goals are big tasks for itself, therefore there are many open tasks and questions left. Here is a list, sorted by priority, of some open tasks:

`x86_64` and other architectures We are limited by Harpy to `x86`, but we would like to move forward, since `x86` is unfriendly to compiler writers. `x86_64` should be rather easy to port, but in principle every architecture supported by GHC is welcome.

Optimizations There are many opportunities, e.g.:

- According to [RDPJ10], students were able to implement sophisticated optimizations such as lazy code motion [KRSA92] with Hoopl.

- Method inlining could be done in the front end of MateVM. An appropriate place would be when translating the Java bytecode of a method to `MateIR`. However, one has to implement a good heuristic to guide inlining decisions. Also, inlining of virtual calls is more complicated, so inlining of static methods should be the first step when implementing such optimization.
- Stack allocation of heap objects. Again, Hoopl should be well suited for escape analysis.

As optimizations increase compile-time-costs, one has to implement some kind of trigger to decide, when a method should be compiled with optimizations. The standard approach for that is profiling. It would be nice to implement a technique, which collects analysis data for a Hoopl-pass at run-time that could be used later on re-compiling a hot method.

The idea can be further evolved to have different kinds of compilers: A fast one (regarding to compile-time) producing slow code and a highly optimized one for hot methods (similar to Jikes RVM [AAC+99]). On the other hand, one might want to implement an interpreter for the sake of simplicity.

Although some optimizations seems to be easy to implement, it turns out to require more effort than expected, since the JVM specification has unpleasant requirements, such as class reloading at run-time. Such side-effects must be considered when implementing optimizations and deoptimization strategies must be developed. Deoptimization is useful for optimistic optimization too. Traps from Section 4.3 could be refined and used to implement the semantics of deoptimizations.

GNU Classpath At the moment MateVM uses a custom and minimalistic base library. GNU Classpath would enable to run real life applications with MateVM. One major feature missing for it, are the concept of class loaders. As GNU Classpath is designed to be easily portable and the exposed interface is simple, it should not be too hard. However, in practice it can be a cumbersome task to debug errors in combination with GNU Classpath. At the moment, executing a simple HelloWorld-program with GNU Classpath loads about 70 classes, but fails when GNU Classpath requests a class-object from MateVM. Although this is already good progress, it would require still some weeks or months to get GNU Classpath running properly.

Overall improvments During the implementation the author improved his Haskell skills from a beginner level. Thus there are some low-hanging fruits:

- implement algorithms more efficient with respect to strictness
- using unboxed types in order to avoid overhead introduced by Haskell data types
- redesign the compilation pipeline in order to enable more interleaved compilation, eventually eliminating iterations on the intermediate representation
- better integration of traps into the `CodeGen`-monad
- other tweaks, that we are not aware of yet

Improve existing run-time services Simple techniques are used in the current state. Improvements are definitely possible at `invokeinterface` and dynamic type check. The latter should be done in native code, as discussed in Section 5.3.

Concurrency One should evaluate what is needed for concurrency. This is a rather difficult task, as it can influence other parts if things are done wrong. Also, concurrency is known to be an optimization barrier, therefore possible optimizations conflicting with concurrency have to be disabled or deoptimization strategies have to be developed.

Bibliography

[AAC$^+$99] Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen, *Implementing Jalapeno in Java*, Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (New York, NY, USA), OOPSLA '99, ACM, 1999, pp. 314–324.

[ABC$^+$02] Bowen Alpern, Maria Butrico, Anthony Cocchi, Julian Dolby, Stephen Fink, David Grove, and Ton Ngo, *Experiences Porting the Jikes RVM to Linux/IA32*, In 2nd Java(TM) Virtual Machine Research and Technology Symposium (JVM'02, USENIX Association, 2002, pp. 51–64.

[ACFG01] Bowen Alpern, Anthony Cocchi, Stephen Fink, and David Grove, *Efficient implementation of Java interfaces: Invokeinterface considered harmless*, Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (New York, NY, USA), OOPSLA '01, ACM, 2001, pp. 108–124.

[AFG$^+$11] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney, *Adaptive optimization in the Jalapeno JVM*, SIGPLAN Not. **46** (2011), no. 4, 65–83.

[BEK09] Manfred Brockhaus, Anton Ertl, and Andreas Krall, *Weiterführender Übersetzerbau*, Institut für Computersprachen. Arbeitsbereich für Programmiersprachen und Übersetzerbau. Technische Universität Wien, 2009.

[Boe13] Hans Boehm, *A garbage collector for C and C++*, 2013, `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`.

[CH00] Koen Claessen and John Hughes, *QuickCheck: a lightweight tool for random testing of Haskell programs*, Proceedings of the fifth ACM SIGPLAN international conference on Functional programming (New York, NY, USA), ICFP '00, ACM, 2000, pp. 268–279.

[Coh91] Norman H. Cohen, *Type-extension type test can be performed in constant time*, ACM Trans. Program. Lang. Syst. **13** (1991), no. 4, 626–629.

[Com12] Haskell Community, *Hackage*, 2012, http://hackage.haskell.org.

[Cor98] The Standard Performance Evaluation Corporation, *SPEC JVM98*, 1998, http://www.spec.org/osg/jvm98/.

[ERU13] Josef Eisl, Sebastian Rumpl, and Bernhard Urban, *JorthVM at github*, https://github.com/JorthVM/JorthVM, 2013, [Online; accessed 10-January-2013].

[FLMPJ99] Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones, *Calling Hell from Heaven and Heaven from Hell*, Proceedings of the fourth ACM SIGPLAN international conference on Functional programming (New York, NY, USA), ICFP '99, ACM, 1999, pp. 114–125.

[GJS+13] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley, *The Java^{TM} Language Specification, Java SE 7 Edition*, Oracle, 2013, http://docs.oracle.com/javase/specs/jls/se7/html.

[GK07] Martin Grabmüller and Dirk Kleeblatt, *Harpy: Run-Time Code Generation in Haskell*, Proceedings of the ACM SIGPLAN workshop on Haskell workshop (New York, NY, USA), Haskell '07, ACM, 2007, pp. 94–94.

[Gra09] Martin Grabmüller, *Dynamic Compilation for Functional Programs*, 2009.

[JWW04] Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich, *Wobbly Types: Type Inference for Generalised Algebraic Data Types*, Tech. report, 2004.

[KP98] Andreas Krall and Mark Probst, *Monitors and Exceptions: How to implement Java efficiently*, IN ACM 1998 WORKSHOP ON JAVA FOR HIGH-PERFORMANCE NETWORK COMPUTING, ACM, 1998, pp. 15–24.

[Kra98] Andreas Krall, *Efficient JavaVM Just-in-Time Compilation*, International Conference on Parallel Architectures and Compilation Techniques, 1998, pp. 205–212.

[KRSA92] Jens Knoop, Oliver Rüthing, Bernhard Steffen, and Rwth Aachen, *Lazy code motion*, 1992, pp. 224–234.

[KWM+08] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox, *Design of the Java HotSpot^{TM} client compiler for Java 6*, ACM Trans. Archit. Code Optim. **5** (2008), no. 1, 7:1–7:32.

[LGC02] Sorin Lerner, David Grove, and Craig Chambers, *Composing Dataflow Analyses and Transformations*, Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (New York, NY, USA), POPL '02, ACM, 2002, pp. 270–282.

[LYBB13] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley, *The Java^{TM} Virtual Machine Specification*, Oracle, 2013, http://docs.oracle.com/javase/specs/jls/se7/html.

[Mar12] Simon Marlow, *Dynamically link GHCi (and use system linker) on platforms that support it*, 2012, http://hackage.haskell.org/trac/ghc/ticket/3658.

[Mic03] Sun Microsystems, *Java^{TM} Native Interface Specification*, http://java.sun.com/j2se/1.5.0/docs/guide/jni/spec/jniTOC.html (2003), http://java.sun.com/j2se/1.5.0/docs/guide/jni/spec/jniTOC.html.

[oG12] University of Glasgow, *Glasgow Haskell Compiler*, 2012, http://www.haskell.org/ghc/.

[Ove12a] Stack Overflow, *get the address of a function (without interfacing C)*, 2012, http://stackoverflow.com/questions/10967598/get-the-address-of-a-function-without-interfacing-c.

[Ove12b] ______, *GHC: segmentation fault under strange conditions*, 2012, http://stackoverflow.com/questions/10341943/ghc-segmentation-fault-under-strange-conditions.

[Ove12c] ______, *GHCi runtime linker issue when using FFI declarations*, 2012, http://stackoverflow.com/questions/10658104/ghci-runtime-linker-issue-when-using-ffi-declarations.

[Pro10] Bob Proulx, *GNU time: incorrect results*, 2010, https://groups.google.com/d/msg/gnu.utils.help/u1MOsHL4bhg/ewaNE5uxLxoJ.

[PS99] Massimiliano Poletto and Vivek Sarkar, *Linear Scan Register Allocation*, ACM TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS **21** (1999), no. 5, 895–913.

[RDPJ10] Norman Ramsey, João Dias, and Simon Peyton Jones, *Hoopl: A Modular, Reusable Library for Dataflow Analysis and Transformation*, Proceedings of the third ACM Haskell symposium on Haskell (New York, NY, USA), Haskell '10, ACM, 2010, pp. 121–134.

[Sch12] Thomas Schilling, *Challenges for a Trace-Based Just-In-Time Compiler for Haskell*, Implementation and Application of Functional Languages (Andy Gill and Jurriaan Hage, eds.), Lecture Notes in Computer Science, vol. 7257, Springer Berlin Heidelberg, 2012, pp. 51–68.

[Sch13] ______, *lambdachine at github*, https://github.com/nominolo/lambdachine, 2013, [Online; accessed 13-January-2013].

[SKT07] Edwin Steiner, Andreas Krall, and Christian Thalinger, *Adaptive Inlining and On-Stack Replacement in the CACAO virtual machine*, Proceedings of the 5th international symposium on Principles and practice of programming in Java (New York, NY, USA), PPPJ '07, ACM, 2007, pp. 221–226.

[Ste12] Don Stewart, *plugins: Dynamic linking for Haskell and C objects*, 2012, http://hackage.haskell.org/package/plugins.

[Ste13] Harald Steinlechner, *Precise Garbage Collection für die MateVM*, 2013, to appear.

[Tha04] Christian Thalinger, *Optimizing and Porting the CACAO JVM*, 2004.

[VHK97] Jan Vitek, R. Nigel Horspool, and Andreas Krall, *Efficient type inclusion tests*, Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (New York, NY, USA), OOPSLA '97, ACM, 1997, pp. 142–157.

[VWJ08] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones, *FPH: First-class Polymorphism for Haskell Declarative, constraint-free type inference for impredicative polymorphism*, 2008.

[Wik12] Wikipedia, *Fold (higher-order function)*, 2012, http://en.wikipedia.org/w/index.php?title=Fold_(higher-order_function)&oldid=518131910.

[WM05] Christian Wimmer and Hanspeter Mössenböck, *Optimized Interval Splitting in a Linear Scan Register Allocator*, Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments (New York, NY, USA), VEE '05, ACM, 2005, pp. 132–141.