FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Diplomarbeitspräsentation

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

Masterstudium:
Computer Architecture and Compiler Design

Bernhard Urban

Technische Universität Wien
Institut für Computersprachen
Arbeitsbereich: Programmiersprachen und Übersetzer
Betreuer: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall

Real World Application: A JIT Compiler

- Nowadays JVMs are inherently **complex** in order to provide fast execution [1] comparable to native applications. They are traditionally implemented in languages such as C or C++
- **Correctness** in a compiler is extremely important with respect to bugs in general and **security** in particular
- Haskell provides language features to enable **abstraction**: strong type system, the Monad construct, type classes and composable code.

"Dirty" low-level tasks in a JIT compiler

- Run-time machine code generation
- Transitions between Haskell world and native code
- Interruption of native code execution to enable run-time services and code patching

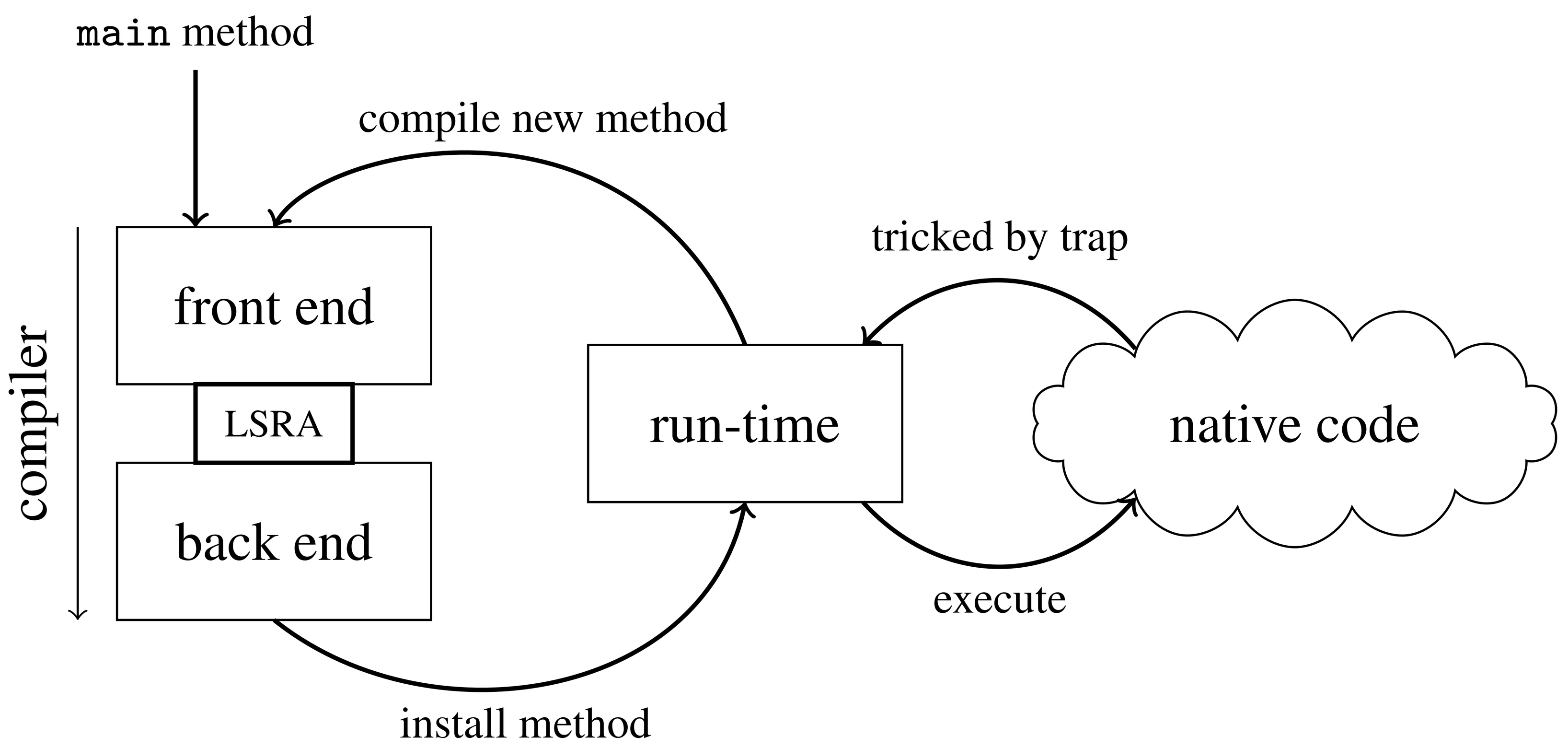
Are those requirements compatible with a high-level **purely functional language** such as Haskell? **Yes!**

Tackling the pitfalls

- Harpy [2] generates code **at run-time** for the `x86`-architecture
- **Haskell → native code**: the `dynamic import` wrapper provided by the *function foreign interface* of the Glasgow Haskell Compiler—comparable to **function pointers** in C
- **Native code → Haskell**: Solved via **traps** (aka. hardware exception or signal), that requires minimal C code.

MateVM [3]: The prototype

The origin of the name was inspired by the caffeine-contained drink called *Club-Mate*, to stay in tradition with other JVM implementations (e.g. Kaffe or CACAO). **Architecture Overview**:



Compiler: Front End

- Intermediate language: Register based, polymorphic regarding register type and implements the notion of basic blocks via **invariants on type level**
- The latter enables usage of Hoopl [4], a Haskell library for **data-flow analysis** (also used by GHC)
- In order to create a **control-flow graph** from JVM bytecode, the JVM stack has to be eliminated and jumps must be resolved
- **Liveness Analysis** is implemented in order to compute live ranges needed for register allocation.

Compiler: Linear Scan Register Allocation (LSRA) [5]

- Few registers available: `ecx`, `edx`, `ebx`, `esi` and `edi` → good **spilling decision** is important
- QuickCheck [6] is a library for testing properties on **random generated instances**. Property for register allocation:
  *For all virtual register there is no other virtual register that has an overlapping live range **and** the same hardware register assigned.*

Front end and LSRA are pure, i.e. code **without side-effects** in the programmer model. The rest is effectful code, but explicitly encapsulated in an I/O-Monad.

Compiler: Back End

- Harpy provides a **domain specific language** similar to Intel syntax for `x86`-assembly to generate machine code
- Custom **combinators** are used to circumvent quirks on ISA level of `x86`, such as `div` that clobbers `eax` and `edx`

Run-time system

- Back end intentionally places traps in generated code, therefore **code patching** is required at run-time
- The run-time system is responsible for class loading, resolve method lookup, dynamic type check, exception handling etc.

Results

- slower than mainstream JVMs (as expected)
- generated code quality is good, but can be certainly improved
- however, the compiler is rather slow and will probably never get faster than implementations in C/C++. That is the price for using a high-level language.

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

References

[1] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney, "A Survey of Adaptive Optimization in Virtual Machines," 2005.

[2] M. Grabmüller and D. Kleeblatt, "Harpy: Run-Time Code Generation in Haskell," 2007.

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

[4] N. Ramsey, J. a. Dias, and S. Peyton Jones, "Hoopl: A Modular, Reusable Library for Dataflow Analysis and Transformation," 2010.

[5] M. Poletto and V. Sarkar, "Linear Scan Register Allocation," 1999.

[6] K. Claessen and J. Hughes, "QuickCheck: a lightweight tool for random testing of Haskell programs," 2000.

Kontakt: lewurm@gmail.com