Abstract Machine Project

Simon Altschuler and Markus Færevaag, Spring 2015

Supervisor: Sven Karlsson, DTU Compute

Vision
Abstract machines (also known as virtual machines) have become a popular method of enabling cross-platform portability of programming languages and a way of unifying the target languages of compilers.

However, the industry standard abstract machines, such as the JVM and Microsoft's CIL, are not well fitted for expressing functional and dynamically typed languages. They are mostly developed with object-oriented languages in mind, which burdens code generation of other language paradigms.

Goals
We want to address this issue by designing an instruction set architecture (ISA) and a corresponding abstract machine that can express both imperative, object-oriented and functional as well as dynamically and strictly typed languages in an intuitive way. A different approach than today's abstract machines is therefore needed. We want to design a language that can support all the mentioned types of languages, without being tightly coupled to one specific. This requires a fundamentally more abstract and powerful design.

Scope
Creating a turing-complete abstract machine in its simplest form is not a complex task, but to implement one that can challenge the current standards is a huge undertaking. We want to create a solid core of features that will enable comfortable code generation for most language paradigms, though we realize that we are not going to be able to implement a production ready product.

Use cases
The use case of the abstract machine is obviously to execute the instruction set. Technically we will strive to develop a portable program that can be used on everything from desktop machines to mobile devices and microprocessors. This requires an implementation that does not depend on architecture-specific functionality and one that is optimized for low performance processors.

We have chosen a set of languages that we feel represents the full spectrum of capabilities that we need to support. The idea is that if we can intuitively generate code for these languages, then we will have a solution that covers the defined needs.

These languages are:

Language	Characteristics
C	Small and simple, pointer-heavy, structures/unions
Java	Class-based (with inheritance), object-oriented, strictly typed
Haskell	Purely functional, type inference
Python	Dynamic types, scripting language

Prototype
As documented in our plan below, we will shortly after the initial design phase, implement a small proof-of-concept. This will be used as a starting point for future development, and will also help with pin pointing practical issues for future design.

Risks
It is possible that creating a wide-ranging abstract machine like this, is too difficult and that managing it is too complex be worth the effort. It is also entirely possible that, in the end, a unified assembly language to express multiple paradigms will not ultimately ease code generation. However, we feel that this is an acceptable risk, and that no matter the outcome, there is valuable knowledge to be gained.

Plan
We are going to manage the project using the agile development process. That means that both the design and implementation will run in iterations and that improvements will happen incrementally. It also means that we might make breaking change during the development if we find that a feature was wrong or missing.

The crude plan of the project is as follows:

Task	Purpose	Phase
Define vision, scope, goals and risks	To have a plan for the project, defining our goals and discussing eventual pitfalls	Inception
Design basic outline for ISA	To design the fundamental ISA, the backbone for the future of the design and	Inception, Design

	implementation. This will continually be developed when adding new features.	
Setup documentation framework and version control	To be able to easily document code and enable code versioning	Inception, Implementation, Environment
Implement a proof-of-concept abstract machine	To have a basis for further development and to get a working code environment setup	Implementation, Environment
Setup test mechanism	To have a test framework capable of both unit- and integration testing	Implementation, Environment
Implement the full ISA	To develop the actual abstract machine, implementing the designed ISA iteratively.	Implementation (iterative)
Thesis	To write the final thesis report, documenting our results	Documentation, Final