Bootstrapping a Lisp compiler

Anton Kindestam
antonki@kth.se

February 12, 2012

Project specification

Goals The goal of the project is to develop a simple compiler targeting the JVM (or optionally other architectures given time; at the very least care should be taken to make targetting another machine viable in the future) for a simple Lisp dialect with influences from Common Lisp, Clojure and Scheme. The compiler should be implemented in the Lisp dialect it targets, or one very close to it so porting the compiler to the targetted Lisp language will be trivial, so bootstrapping can be performed and a self-hosting compiler achieved.

The primary goal of the project is to study techniques for bootstrapping a self-hosting compiler for a dynamic language such as LISP. An earlier goal during the design process will be to have the compiler compile an interpreter for the implemented language[1].

Another goal of the project will be to investigate the possibility of implementing Tail Call Optimization[2] on top of the JVM and, if at all viable, implement it in the compiler to allow for boundless tail recursion for the compiled code. This is tricky since the JVM lacks the ability to jump between functions and is something that e.g. Clojure lacks (though arguably in part by design as working around the JVM for Tail Call Optimization can make interaction with Java libraries tricker).

Host For hosting the compiler during development and bootstrapping either an interpreter for a small and simple Lisp language implemented in Java that I have previously written (currently named LJSP) and that can be easily adapted to better fit the Lisp dialect for this project or a full-blown Lisp environment already implemented in Java or on top of the JVM such as Clojure or ABCL (a Common Lisp implementation).

The dialect The dialect implemented will most likely be based on the dialect implemented by the LJSP interpreter (which is in turn mostly influenced by

[1] A Lisp interpreter is a fairly trivial program and being able to compile it will be a good measuring stick for compiler progress. In fact the language can even be considered partially bootstrapped at this point (The compiler can generate the interpreter which can run the compiler. The step to having the compiler compile the compiler is not far away.).

[2] When the last thing a function does is call a function, another or itself, there is no longer a need to keep the stack frame of the caller function and it is removed and the last function is jumped to rather than called with the stack. This allow boundless tail recursion without using up all of the stack and getting to a stack overflow.

Common Lisp, Scheme and Maclisp[3]) but with a bit of changes to make it more viable for compilation along with other fixes (for instance LJSP currently has only dynamic scoping, lexical scoping would be highly desirable both as a language feature and for being more compilable) and, if time permits, additions and changes inspired by Clojure. Due to time constraints some changes to limit the size and complexity of the language might be neccesary (e.g. removing datatypes not used by the compiler and so on).

Using the LJSP Lisp dialect as a base is reasonable since it is very simple (the interpreter clocks in at no more than 1000 lines of Java code) and it is obviously very familiar to me. If using the LJSP interpreter as the host for initial bootstrapping will be useful in the way that the interpreter can easily be adjusted to the compiler and the Lisp dialect being implemented.

Another approach would be to target a subset of Clojure, or another popular Lisp dialect, and write the compiler in that dialect. Interestingly enough a subset of LJSP can in turn be considered in turn a (fairly small) subset of Common Lisp if one draws the lines very carefully. What this means is that the choice of host language matters little to the dialect implemented as long as the host language is Lisp and the target language is Lisp most differences can be reconciled with carefully choosing language subsets and smart usage of language macros.

Implementation The project will focus mostly on the topic of bootstrapping a self-hosting compiler and thus very little energy will be focused on different compiler implementation techniques such as optimization and more on actually implementing a simple compiler prototype unless time permits.

On the extreme end of things a function might compile to almost nothing but branches and calls to other functions since such a compiler is very simple to implement yet still permits adding some optimizations later on.

If possible, however, the compiler should aim for more intelligence then that. E.g. expanding built-ins, expanding arithmetic operations (type inference and/or simple optional explicit type declarations permitting) etc.

Naturally Lisp macros will be expanded at compile-time.

Using a Lisp language to implement another Lisp language has the nice property of skipping the steps of lexing and parsing since you start out with the code encoded in lists in a format that is basically already the Abstract Syntax Tree of the language being implemented.

If the LJSP interpreter is used as the original host a ways of implementation that seems interesting is to perform bootstrapping incrementally by gradually replacing parts of the original environment, starting at having the compiler capable of compiling the interpreter for the language, until the runtime environment of the system depends on as little Java code as possible. Some traces of Java left, in particular for some basic datastructures, will be inevitable however.

Generally existing free software Lisp/Clojure/Scheme to JVM compilers will be used for inspiration for the compiler implementation.

Litterature on the subject of compiler design, and Lisp compilers in particular, will be consulted throughout the process so as to not having to reinvent the wheel as well as giving a general direction the task. For instance

[3]An elderly Lisp dialect developed at MIT in the 1970s

`http://scheme2006.cs.uchicago.edu/11-ghuloum.pdf` provides an overview of a task similar to mine.

Risk assesment It is possible that, due to the limited time frame, a compiler truly capable of bootstrapping itself will not have been produced by the end.

However since the developing, and bootstrapping, will be done incrementally in steps the end result will not be something entirely unusable and rather a fully functioning compiler.

Even without having successfully having bootstrapped the compiler the subject of bootstrapping a compiler for a dynamic language will have been extensively studied and thus the goal of investigating applicable techniques will have been accomplished. More than likely the subject of investigating possible workarounds for implementing Tail Call Optimization atop the JVM will have been studied.

Limiting the language to be small enough will make the task manageable and will increase the likelihood of success.