Towards an Atomic Abstract Machine

David R. Sherratt

December 5, 2016

Abstract

The familiar *call-by-need* evaluation strategy (*as used in Haskell compilers*), also known as *lazy evaluation* shows best performance when terms are duplicated. When we have more control over the duplication of the term, such that we can choose which segments of the term to duplicate, then we can implement *full laziness*. Full laziness means we duplicate only what is necessary, and the maximal free subexpressions (the largest subterms such that, if they contain a bound variable x, they also contain its binder λx) remain.

Haskell compilers can perform full laziness at compile time. However, during runtime the implementation becomes too expensive. The idea behind this project is to further study the process of implementing a fully lazy evaluation strategy.

The purpose of sharing in the λ-calculus is to have better control over duplication of terms. Sharing is the use of a single representation for multiple instances of a common subterm. During evaluation, instead of duplicating a term, we can share it. This allows us to evaluate all the copies of the subterm simultaneously, by evaluating their shared representation [6].

In the atomic λ-calculus [2], we implement two types of closures. The usual notion of closure is a term with an associated sharing implemented as a let-construct. The second notion of closure introduces the *distributor* construct, which allows for duplication of a term atomically. The distributor uses a more restricted notion of *unsharing* similar to Lamping's sharing graphs [4] which performs optimal reductions as defined in [5]. This is exactly what allows this calculus to perform fully lazy evaluation.

An *abstract machine* is a theoretical implementation of a strategy for a language on an abstract notion of computer architecture, made from directly implementable constructs. An abstract machine for evaluating atomic λ-terms would be a fully lazy abstract machine. For my PhD I aim to develop an *atomic abstract machine* based off the atomic λ-calculus. To achieve this, we make all the reduction rules in the calculus *local* in the sense of [3]. To achieve this, we construct a new calculus based on the atomic λ-calculus that makes the variable scopes explicit while maintaining fully lazy sharing.

This *directed atomic λ-calculus* uses a variant of director strings [1] to make the scopes of terms explicit. When performing substitution, instead of traversing the complete term, we instead only follow the path of the variable we want to replace which is highlighted through the annotations. An application can indicate the location of a variable with a 0 (in the left hand side) or with a 1 (in the right hand side), and similar for sharing. The scoping information of terms can then be used so that all the reduction rules can be considered local. One can then start work on an abstract machine for this calculus, that performs fully lazy λ-evaluation.

References

[1] Maribel Fernández, Ian Mackie, and François-Régis Sinot. *Lambda-calculus with director strings*. Applicable Algebra in Engineering, Communication and Computing. 2005.

[2] Tom Gundersen, Willem Heijltjes, and Michel Parigot. *Atomic lambda calculus: A typed lambda-calculus with explicit sharing.* Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science. IEEE Computer Society, 2013.

[3] Yves Lafont. *Interaction nets*. Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 1989.

[4] John Lamping. *An algorithm for optimal lambda calculus reduction*.Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 1989.

[5] Jean-Jacques Lévy. *Optimal reductions in the lambda-calculus*. To HB Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism. 1980.

[6] Christopher Peter Wadsworth. *Semantics and Pragmatics of the Lambda-Calculus*. Diss. University of Oxford, 1971.