A Typed Calculus Supporting Shallow Embeddings of Abstract Machines

Aaron Bohannon
Zena M. Ariola
Amr Sabry

April 23, 2005

1 Overview

The goal of this work is to draw a formal connection between steps taken by abstract machines and reductions in a system of proof terms for a version the sequent calculus. We believe that by doing so we shed light on some essential characteristics of abstract machines, proofs in sequent calculus systems, and weak normalization of λ-terms. The machines that we consider are the (call-by-name) Krivine machine and a call-by-value machine that may be called a "right-to-left CEK machine" but with some modifications can be seen as a proto-ZINC machine.

The formal connection we exhibit is, in fact, an embedding of the machines into the term calculus. We embed run-time data structures, such as the control stack and environment, in such a way that the operational semantics of the machine corresponds to reduction steps in the calculus. The abstract machine state, including the code that it executes, is captured as a term; the abstract machine transitions are captured as term reductions.

This is in contrast to specifying the operational semantics on top of the calculus. In other words, our goal is to provide a *shallow embedding* of an abstract machine in a calculus/logic, as opposed to a *deep embedding*. This allows reasoning about the machine *inside* the logic itself instead of on *top* of it. The logical formulae that are assigned to proof terms provide a type system for the term language via the Curry-Howard isomorphism, and because of the method of embedding the machines into the terms, this type system can be directly lifted to the machine code and machine states, thereby allowing an elegant and simple formulation of safety, based on the subject reduction theorem of the calculus.

2 Tail-Recursive Evaluators

Plotkin [9] showed how abstract machines could be seen as an implementation of an evaluation function for a functional programming language. It can be noted that a basic evaluation function, whether implementing call-by-name or call-by-value semantics, is not tail-recursive. This corresponds to the fact that the small-step operational semantics has a recursive definition, relying on congruence rules. An implementation of these rules naturally involves a process of searching for the next redex, which may be arbitrarily deep in a term. This search must be managed with care when computing a sequence of reductions, since the cost of computing a single reduction step is linear in the size of the term [4].

An implementation of an abstract machine, on the other hand, can be directly written down as a tail-recursive function. One view of abstract machines is that they are simply tail-recursive evaluators. The process of constructing such an evaluator, as presented by Reynolds [11] and more recently explored by Ager *et al.* [2], consists of defunctionalizing the continuations in a CPS interpreter. A defunctionalized continuation is actually just a data structure representing an evaluation context. As shown by Herbelin [7], proof terms for sequent calculi have a computational interpretation as evaluation contexts. Thus, it is very natural to believe that sequent calculi would have a relationship with abstract machines. A simple connection between the $\overline{\lambda}\mu\tilde{\mu}$-calculus and the Krivine machine was observed by Curien and Herbelin [3].

3 Calculi for Machines

If a machine is correctly implements evaluation of λ-terms, then it will certainly be possible to prove a correspondence between the machine and the λ-calculus. However, there are various calculi that may be much closer to abstract machines, *i.e.* have a more direct statement and proof of correspondence. The closest correspondence would occur when we could define a translation from machine states to terms (or vice-versa) in a purely compositional manner, such that the transitions of the machine could be matched up with the steps of a particular reduction strategy on the term so that neither one would ever take more than a statically fixed number of steps for a given step in the other system.

In order to achieve a correspondence at this level, the term calculus must possess certain features. First, it must be able to simulate weak β-reduction without performing arbitrarily deep searches for the next redex. Otherwise, one step in the calculus would correspond to an arbitrary number of steps of the machine. The $\overline{\lambda}\mu\tilde{\mu}$-calculus [3] described by Curien and Herbelin has this property. They define translations from λ-terms to $\overline{\lambda}\mu\tilde{\mu}$-terms, such that the computational reduction rules of the $\overline{\lambda}\mu\tilde{\mu}$-calculus can be used without any congruence rules to simulate weak β-reduction of the λ-terms. Moreover, by making a simple choice of which way to resolve a critical pair, the same computational rules can be used to simulate either call-by-name or call-by-value reduction on λ-terms.

Abstract machines are also designed to break down the process of substitution into small steps, and they generally carry out these substitutions in a lazy manner. Thus, the other important feature of a good calculus for our simulations is an explicit notion of substitution. Calculi with explicit substitutions were investigated by Abadi *et al.* [1] and certain variants have been used to prove the correctness of abstract machines, *e.g.* the λ_{env}, which was used by Leroy [8] when the ZINC machine was introduced. Hardin, Maranget, and Pagano [6] proposed the $\lambda\sigma_w$-calculus as a "calculus of closures" for proving the correctness of abstract machines and representing the output of compilers. While they succeeded in providing an elegant calculus specialized for weak β-reduction, the calculus was still based upon the structure of natural deduction, and therefore cannot satisfy the requirement of the last paragraph. On the other hand, the $\overline{\lambda}\mu\tilde{\mu}$-calculus does not provide any notion of explicit substitution, so it is not immediately satisfactory, either.

A version of the $\overline{\lambda}\mu\tilde{\mu}$-calculus with explicit substitutions has been studied and found to be very well-behaved [10]. Unfortunately, the inclusion of explicit substitutions is not, in itself, enough to guarantee that a calculus has the properties that we desire. The reason is that when a term has multiple substitutions at the outermost level, the next redex must (eventually) be a propagation of the innermost substitution. The search for this redex, which may be arbitrarily deep, would not mirror the operation of an abstract machine. One way around this is to take the approach of the $\lambda\sigma_w$-calculus and use *simultaneous* explicit substitutions. However, we take another approach and represent environments within the calculus. This approach was inspired by Douence and Fradet [5], but instead of working abstractly at the level of combinators, we provide a very concrete embedding of environments in the calculus, which gives us the benefit of being able to apply the type system of the calculus to the environments in a direct way.

4 Our Development

The calculus into which we embed the abstract machines is a slightly modified version of the $\overline{\lambda}\mu\tilde{\mu}$-calculus with explicit substitutions that was studied by Polonovski [10]. Our first modification is the addition of an explicit weakening construct that acts as a method of garbage collection. This is necessary for simulating the mutable machine registers that abstract machines generally have. The other modification involves restricting the allowable terms by constraining the contexts in the typing judgments. This modification is not technically necessary but allows us to assert that the typable α-equivalent terms are in a one-to-one correspondence with the typable terms viewed without α-equivalence, thereby highlighting the successful elimination of the need to keep track of names in the calculus. The restricted set of terms are allowed to use only a single term variable—which is used to encode an accumulator register—and two context variables—one is used for encoding the run-time stack pointer and the other for encoding the environment pointer. We call this the $\overline{\lambda}\mu\tilde{\mu}\mathtt{r}\uparrow$-calculus.

The $\overline{\lambda}\mu\tilde{\mu}\mathtt{r}\uparrow$-calculus imposes a useful structure on terms that is closer to the level of an abstract machine. In fact, the individual reduction steps in this system are much more fine-grained that one would see in most abstract machines. In order to show how the reduction steps of this calculus correspond to abstract machines, such as the Krivine machine, it is useful to develop a sort of toolkit of "macros" for commands and terms in the $\overline{\lambda}\mu\tilde{\mu}\mathtt{r}\uparrow$-calculus. Thereafter, we exhibit a set of reduction steps on these macro-terms that correspond to multiple reduction steps at the raw term level. These macro-reductions implement a specific strategy of small-step reductions; hence, we present one set implementing call-by-name and one set implementing call-by-value, with a concrete description of the strategies that they implement on the pure $\overline{\lambda}\mu\tilde{\mu}\mathtt{r}\uparrow$-terms.

It becomes apparent that these coarse-grained systems are, in a literal sense, abstract machines themselves built directly out of the $\overline{\lambda}\mu\tilde{\mu}\mathtt{r}\uparrow$-calculus. It is then a very small (almost trivial) step to draw the correspondence with the traditional Krivine machine and a call-by-value machine that is similar to the ZINC machine, and we see how these machines arise out of the duality of the calculus. The typing rules of the calculus are then also lifted in the obvious way to give a type system to the macro-terms and, by extension, the abstract machines themselves, thus allowing an elegant statement of safety at the level of machines.

References

[1] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 31–46, New York, 1990. ACM Press.

[2] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. From interpreter to compiler and virtual machine: A functional derivation. Research Series RS-03-14, BRICS, March 2003. 36 pp.

[3] Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 233–243, New York, 2000. ACM Press.

[4] Olivier Danvy and Lasse R. Nielsen. Syntactic theories in practice. In Mark van den Brand and Rakesh Verma, editors, *Electronic Notes in Theoretical Computer Science*, volume 59. Elsevier, 2001.

[5] Rémi Douence and Pascal Fradet. A systematic study of functional language implementations. *ACM Trans. Program. Lang. Syst.*, 20(2):344–387, 1998.

[6] Thérèse Hardin, Luc Maranget, and Bruno Pagano. Functional back-ends within the lambda-sigma calculus. Technical Report RR-3034, INRIA, November 1996.

[7] H. Herbelin. A lambda-calculus structure isomorphic to Gentzen-style sequent calculus structure. In *Proc. Annual Conference of the European Association for Computer Science Logic, Kazimierz, Poland*, volume 933 of *Lecture Notes in Computer Science*, Berlin, 1994. Springer-Verlag.

[8] Xavier Leroy. The ZINC experiment : an economical implementation of the ML language. Technical Report RT-0117, INRIA, February 1990.

[9] G. D. Plotkin. Call-by-name, call-by-value, and the lambda-calculus. *Theoretical Computer Science*, 1(2):125–159, December 1975.

[10] Emmanuel Polonovski. *Substitutions explicites, logique et normalisation*. PhD thesis, Université Paris 7, 2004.

[11] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference*, pages 717–740. ACM Press, 1972.