On the Correctness and Efficiency of the Krivine Machine

Mitchell Wand
Northeastern University

Daniel P. Friedman
Indiana University

February 12, 2003

Abstract

We provide a short derivation of the Krivine machine by showing how it simulates weak head reduction. We propose a small variation to the machine that improves its performance dramatically.

Recall that every closed λ-term is either in weak head normal form (that is, an abstraction), or is of the form

$$(\lambda x.M)\ N_1 \ldots N_k$$

where as usual concatenation denotes application and associates to the left.

So weak head reduction is simply the single reduction

$$(\lambda x.M)\ N_1 \ldots N_k \to M[N_1/x]\ N_2 \ldots N_k$$

If we represent the λ-terms $(\lambda x.M), N_1, \ldots, N_k$ as closures, we can avoid doing the substitution. To do this, let us define some syntax:

M	$::=$	$x \mid M\,N \mid \lambda x.M$	Terms
C	$::=$	(M, ρ)	Closures
ρ	$::=$	$[] \mid \rho[x = C]$	Environments
L, R	$::=$	$C_0\, C_1 \ldots C_n$	Configurations

Closures are subject to the condition that $\mathrm{fv}(M) \subseteq \mathrm{dom}(\rho)$, so closures and environments denote only closed terms. In the configuration $C_0 \ldots C_n$, where $C_0 = (M_0, \rho_0)$, we say M_0 is the *head term* of the configuration.

We define a translation U from closures and configurations to closed λ-terms. The translation of a closure is obtained by substitution; the translation of a configuration is the left-associative application of its elements, analogous to concatenation of terms:

$$
\begin{array}{lll}
U(M,[]) & = M & \\
\text{otherwise:} & & \\
U(x,\rho) & = U(\rho(x)) & \text{if } \rho(x) \text{ is defined} \\
 & = x & \text{if } \rho(x) \text{ is undefined} \\
U(MN,\rho) & = U(M,\rho)\,U(N,\rho) & \\
U(\lambda x.M,\rho) & = \lambda x.U(M,\rho - x) & \\
 & & \\
U(C_0 \ldots C_n) & = U(C_0 \ldots C_{n-1})\,U(C_n) & n > 0
\end{array}
$$

where $\rho - x$ denotes ρ with the binding for x, if any, removed. Since all the substituends are closed terms, we need not worry about renaming variables.

So if we represent $(\lambda x.M)\,N_1 \ldots N_k$ by $C_0\,C_1 \ldots C_k$, we can predict the behavior of a weak head reduction by the three rules

$$
\begin{array}{lll}
(x,\rho)\,C_1 \ldots C_k & \to (\rho(x)\,C_1 \ldots C_k & \textbf{var} \\
((M\,N),\rho)\,C_1 \ldots C_k & \to (M,\rho)\,(N,\rho)\,C_1 \ldots C_k & \textbf{app} \\
((\lambda x.M),\rho)\,C_1 \ldots C_k & \to (M,\rho[x = C_1])\,C_2 \ldots C_k \quad (k \geq 1) & \textbf{abs}
\end{array}
$$

which is the Krivine Machine. The first two rules simulate the behavior of U, and the third rule simulates head reduction.

We can make this observation more precise as follows:

Lemma 1 *For each reduction step $L \to R$ of the Krivine machine, either:*

1. *$U(L) = U(R)$ and the head term of R is smaller than the head term of L, or*

2. *$U(L) \to U(R)$ by a weak head reduction.*

Furthermore, if $U(L)$ is not an abstraction, then L is not in a terminal configuration.

Proof: The **var** and **app** rules satisfy case 1; the **abs** rule satisfies case 2. L is in a terminal configuration iff L consists of a single closure whose term is an abstraction, that is, iff $U(L)$ is an abstraction.

Therefore, we have:

Theorem 1 *If M is a closed term, and M weak head-reduces to an abstraction W, then the Krivine machine, started in the configuration $(M, [])$, terminates in a configuration R such that $U(R) = W$. If M does not head-reduce to an abstraction, then the Krivine machine runs infinitely.*

Proof: If the Krivine machine terminates in configuration R, then by the preceding lemma, M head-reduces to $U(R)$. Since both weak head reduction and the Krivine machine halt as soon as they reach a single abstraction, we have $U(R) = W$.

Conversely, if $U(L) = M$ and M takes a weak head reduction step to M', then by the lemma then L is not a terminal configuration, and the machine can take only finitely many **var** and **app** steps before it takes an **abs** step, which simulates the reduction. Hence if M head-reduces infinitely, then so does the Krivine machine.

This proof extends to some variants of the Krivine machine. For example, consider the following variant of rule **app**:

$((M\ x), \rho)\ C_1 \ldots C_k \to (M, \rho)\ \rho(x)\ C_1 \ldots C_k$		**app-var**
$((M\ N), \rho)\ C_1 \ldots C_k \to (M, \rho)\ (N, \rho)\ C_1 \ldots C_k$	N not a variable	**app**

Since **app-var** satisfies the conditions of the lemma, the variant machine is also sound and complete for weak head reduction.

Since only $\rho(x)$ is kept on the stack, rather than all of ρ, this modification can drastically reduce the storage requirements of the machine. To illustrate the

difference, consider the reduction of $(\lambda x.xx)\ (\lambda x.xx)$. Let $\Delta = (\lambda x.xx)$, let $\rho_1 = [x = (\Delta, [])]$, $\rho_2 = [x = (x, \rho_1)]$, *etc*. Then the original machine reduces as follows:

$$
\begin{array}{l}
(\Delta\ \Delta, [\,]) \\
\rightarrow (\Delta, [\,])\ (\Delta, [\,]) \\
\rightarrow (x\ x, \rho_1) \\
\rightarrow (x, \rho_1)\ (x, \rho_1) \\
\rightarrow (\Delta, [\,])\ (x, \rho_1) \\
\rightarrow (x\ x, \rho_2) \\
\rightarrow (x, \rho_2)\ (x, \rho_2) \\
\rightarrow (x, \rho_1)\ (x, \rho_2) \\
\rightarrow (\Delta, [\,])\ (x, \rho_2) \\
\rightarrow (x\ x, \rho_3) \\
\ldots
\end{array}
$$

The machine loops, as required, and uses only a bounded number of stack positions, but the environments grow unboundedly, even though the head reduction of $\Delta\,\Delta$ requires only constant space. Furthermore, the machine requires $n+1$ steps to get from $(x\,x, \rho_n)$ to $(x\,x, \rho_{n+1})$, so it requires $\Theta(n^2)$ steps to complete n iterations.

The new machine reduces as follows:

$$
\begin{array}{l}
(\Delta\ \Delta, [\,]) \\
\rightarrow (\Delta, [\,])\ (\Delta, [\,]) \\
\rightarrow (x\ x, \rho_1) \\
\rightarrow (x, \rho_1)\ (\Delta, [\,]) \\
\rightarrow (\Delta, [\,])\ (\Delta, [\,]) \\
\ldots
\end{array}
$$

It simulates the reduction using only constant space. Although this illustratation involves an infinite reduction, similar behavior occurs for any term involving the fixpoint combinator Y: in general, the maximum nesting depth of environment is proportional to the number of iterations through Y. For example, consider the following term, which creates an exponential computation in continuation-passing style:

$$
\begin{array}{l}
\lambda n.(Y\,(\lambda f.\lambda x.\lambda k. \\
\qquad IF\ (Z?\ x) \\
\qquad\quad (k\ T) \\
\qquad\quad (f\ (P\ x) \\
\qquad\qquad (\lambda b.IF\ b\ (f\ (P\ x)\ k)\ (k\ F)))) \\
\quad n \\
\quad (\lambda x.x))
\end{array}
$$

Here Y is the usual fixpoint combinator, T, F, and IF are the usual Church booleans, and $Z?$ and P are the zero-test and predecessor operations on the Barendregt numerals [Bar81, Section 6.2]. Given a numeral n, this term always returns T, but does so only after exploring the entire numeral (a list of length n) exponentially many times. Since the computation is tail-recursive, it iterates through the Y combinator exponentially many times.

We compared the behavior of the old and new machines experimentally on these terms. As expected, we found that the depth of the environment grows exponentially in the original machine but only linearly in the new one. (The linear nesting depth is due to the representation of $(P\,(P\,\ldots\,(P\,n)))$, can be represented only as a closure with an environment of nesting depth k, where k is the number of P's in the term).

Furthermore, the old machine takes approximately 13% more steps than the new one, because of the steps needed to traverse the extra environment links.

Historical Note

Krivine's machine was unpublished until this issue. We believe that we learned about it from [Cré90]. Wand's notes indicate that he figured out this explanation for his semantics class in the Winter of 1992. Friedman discovered the space leak in response to the call for papers for this special issue.

The idea of short-circuiting variable operands is derived from modelling call-by-reference and improves the execution for languages that use call-by-name and call-by-need. The short-circuiting appears in Section 3.8 of [FWH01].

Thanks to Olivier Danvy for urging us to submit this note.

References

[Bar81] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, Amsterdam, 1981.

[Cré90] P. Crégut. An abstract machine for the normalization of λ-terms. In *Proc. 1990 ACM Symposium on Lisp and Functional Programming*, pages 333–340. ACM, June 1990.

[FWH01] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. MIT Press, Cambridge, MA, second edition, 2001.