Improving the Lazy Krivine Machine

Daniel P. Friedman, Abdulaziz Ghuloum, Jeremy G. Siek,
and Lynn Winebarger
Indiana University

Abstract. Krivine presents the $\mathcal{K}$ machine, which produces weak head normal form results. Sestoft introduces several call-by-need variants of the $\mathcal{K}$ machine that implement result sharing via pushing update markers on the stack in a way similar to the TIM and the STG machine. When a sequence of consecutive markers appears on the stack, all but the first cause redundant updates. Improvements related to these sequences have dealt with either the consumption of the markers or the removal of the markers once they appear. Here we present an improvement that eliminates the production of marker sequences of length greater than one. This improvement results in the $\mathcal{C}$ machine, a more space and time efficient variant of $\mathcal{K}$.

We then apply the classic optimization of short-circuiting operand variable dereferences to create the call-by-need $\mathcal{S}$ machine. Finally, we combine the two improvements in the $\mathcal{CS}$ machine. On our benchmarks this machine uses half the stack space, performs one quarter as many updates, and executes between 27% faster and 17% slower than our $\mathcal{L}$ variant of Sestoft's lazy Krivine machine. More interesting is that on one benchmark $\mathcal{L}$, $\mathcal{S}$, and $\mathcal{C}$ consume unbounded space, but $\mathcal{CS}$ consumes constant space. Our comparisons to Sestoft's Mark 2 machine are not exact, however, since we restrict ourselves to unpreprocessed closed lambda terms. Our variant of his machine does no environment trimming, conversion to deBruijn-style variable access, and does not provide basic constants, data type constructors, or the recursive *let*. (The Y combinator is used instead.)

Keywords: lambda calculus, abstract machine, call by need, lazy evaluation

1. Introduction

We present three call-by-need $\mathcal{K}$ (Krivine) machines, the first two based on Sestoft's [15] characterization of the call-by-need $\mathcal{K}$ machine but limited to unpreprocessed closed lambda terms. The third machine combines the improvements of the first two. First, we present the $\mathcal{C}$ machine, which prevents the production of adjacent (update) markers, as all but the first are redundant. Update markers are presented in Fairbairn and Wray's Three Instruction machine [6] and are redescribed in Sestoft's development of his Mark 2 machine and appear in the Spineless Tagless G-machine (STG) [7].

Although Fairbairn and Wray briefly refer to placing markers in the same frame, they do not address avoiding the production of marker sequences. Guy Argo [2] makes improvements to the TIM, including mechanisms for avoiding updates to the heap while consuming markers.

© 2003 *Kluwer Academic Publishers. Printed in the Netherlands.*

The STG shares results among closures in the same way we do. However, the STG still pushes redundant markers on the stack and relies on the garbage collector to remove them. In contrast, our $\mathcal{C}$ machine prevents the production of sequences of markers to begin with.

On page 24 of T.B. Steel [16], van Wijngaarden makes the following incredibly insightful observation:

> I suppose you have a certain implementation of a procedure call in mind when you say that. But this implementation is only so difficult because you have to take care of the goto statement. However, if you do this trick I devised, then you will find that the actual execution of the program is equivalent to a set of statements; no procedure ever returns because it always calls for another one before it ends, and all of the ends of all the procedures will be at the end of the program; one million or two million ends. If one procedure gets to the end, that is the end of all; therefore you can stop. That means you can make the procedure implementation so that it does not bother to enable the procedure to return. That is the whole difficulty with procedure implementation. That's why this is so simple; it is exactly the same as a goto, only called in other words.

His observation, however, is not directly applicable to call-by-need procedures. Why? At the end of evaluating a term, when it stops with a value, the value must be stored, *and* the evaluation must move on to other terms. But, the idea is still germane. It just needs a little tweaking. Thinking about continuations represented as a stack, the markers on a stack machine correspond to the returns of van Wijngaarden's device. Thus, a sequence of markers can be collapsed into one marker just as a sequence of returns can be collapsed into one return. This prevention of marker sequences is a bit more subtle than that, but the essence of the idea of avoiding the building of marker sequences can be found in this notion: although we do not transform the program into continuation-passing style, the machine, itself, behaves as if the program has been so transformed. In some sense, we can think of the program as generating its own continuations as it computes, rather than having them built by some preprocessor.

The second machine we discuss, the $\mathcal{S}$ machine, modifies the $\mathcal{K}$ machine to use a short-circuiting optimization used in an early implementation of Algol 60 [13]. Though the optimization was originally intended for the call-by-name setting of Algol 60, it applies equally well to the call-by-need $\mathcal{K}$ machine and has significant advantages in both time and space resource consumption.

The third machine, the $\mathcal{CS}$ machine, combines the improvements of the $\mathcal{C}$ machine and the $\mathcal{S}$ machine, resulting in a machine that, for our benchmarks, uses half the stack space, performs one quarter as many updates, and executes between 27% faster and 17% slower than our $\mathcal{L}$ machine variant of Sestoft's Mark 2 machine. In one benchmark,

the $\mathcal{CS}$ machine consumes constant space, whereas $\mathcal{C}$, $\mathcal{S}$, and $\mathcal{L}$ each consume unbounded space.

The paper's structure is as follows. In Section 2 we present the call-by-name Krivine machine and our variant $\mathcal{L}$ of Sestoft's Mark 2 machine. In Section 3, we present the $\mathcal{C}$ machine, which prevents the creation of marker sequences. We prove the correctness of this machine with respect to the $\mathcal{L}$ machine via two correctness preserving transformations. The first transformation constructs a throw-away machine, $\underline{\mathcal{C}}$, which adds a level of indirection, and the second transformation collapses marker sequences by sharing result locations. We then consider properties of this machine. In Section 4, we introduce a call-by-need machine, $\mathcal{S}$, with short-circuiting (of operand variable dereferencing) and observe why short-circuiting in the call-by-need machine has advantages similar to those of the call-by-name machine. In Section 5 we combine the collapsed markers improvement with short-circuiting in the machine $\mathcal{CS}$. In Section 6, we show some experimental results. Section 7 summarizes the related work and is followed by the conclusion.

2. Background

The language of lambda terms is given by the following grammar.

$x, y, z \in \text{Var}$	Variables
$M, N \in \text{Exp} ::= x \mid (MN) \mid \lambda x.M$	Terms

The goal is to reduce closed lambda terms to *weak head normal form*, i.e., to terms of the form $\lambda x.M$. During execution, subterms may contain free variables; we pair terms with environments to bind the free variables and call these pairs *closures*. A closure whose term is of the form $\lambda x.M$ is a *value*. The set of values is Val and we let v range over Val.

The $\mathcal{K}$ machine is a simple stack-based call-by-name evaluator (to weak head normal form) restricted to lambda terms. The key feature that makes $\mathcal{K}$ call-by-name rather than call-by-value, is that the App rule pushes the operand N on the stack unevaluated. $\mathcal{K}$ then evaluates N each time the associated variable is referenced. Figure 1 shows the single-step operational semantics of $\mathcal{K}$ ($[]$ is the everywhere undefined function and $()$ is the empty stack.).

We use the phrase *execution sequence of* M to mean a sequence of states where the first is an *initial state*, a state of the form $(\langle M, []\rangle, ())$,

$$\longrightarrow_{\mathcal{K}}: \text{State} \to \text{State}$$
$$\text{State} = \text{Clos} \times \text{Stack}$$
$$\rho \in \text{Env} = \text{Var} \to \text{Clos}$$
$$c \in \text{Clos} = \text{Exp} \times \text{Env}$$
$$\sigma \in \text{Stack} = \mathsf{List}[\text{Clos}]$$

$$\frac{\rho(x) = c}{(\langle x, \rho\rangle, \sigma) \longrightarrow_{\mathcal{K}} (c, \sigma)} \quad (\textsc{Var})$$

$$(\langle (MN), \rho\rangle, \sigma) \longrightarrow_{\mathcal{K}} (\langle M, \rho\rangle, \langle N, \rho\rangle :: \sigma) \quad (\textsc{App})$$

$$(\langle \lambda x.M, \rho\rangle, c :: \sigma) \longrightarrow_{\mathcal{K}} (\langle M, \rho[x \mapsto c]\rangle, \sigma) \quad (\textsc{Call})$$

Figure 1. Definition of the $\mathcal{K}$ (Krivine) Machine

and where for every adjacent pair of states s, t there is a $\mathcal{K}$ rule such that $s \longrightarrow_{\mathcal{K}} t$. If an execution sequence is finite and no rules apply to the last state, then the last state is *final* and thus has the form $(v, ())$ for some value v. As we present other machines, we refer to an execution sequence *on* a particular machine. Initial and final states for these machines are analogous to those of $\mathcal{K}$. Initial states contain a closure and an empty stack, where the closure is composed of a closed term and an empty environment. Final states contain a value and an empty stack.

The $\mathcal{K}$ machine, however, is horribly inefficient (See Section 6.) because of the repeated evaluations of the same operand. A lazy, or call-by-need, machine solves this problem by evaluating an operand at most once and then sharing its value among the occurrences of the same variable. A common mechanism to accomplish this is to add a level of indirection in the environment through pointers into a heap and then use a marker on the stack [6, 15, 7]. The marker is placed on the stack when a variable is referenced and its associated closure is not a value. When the closure is a value, the marker will once again be at the top of the stack to indicate that its value should be placed in the heap so that it can be shared.

Sestoft describes how to derive a call-by-need machine from a call-by-need natural semantics. His derivation can be mechanized: first convert the semantics to continuation-passing style and then replace the continuation functions with an equivalent data-structure representation, namely a stack. The $\mathcal{L}$ machine defined in Figure 2 is the result

$$\longrightarrow_{\mathcal{L}}\colon \mathrm{State} \to \mathrm{State}$$
$$\mathrm{State} = \mathrm{Clos} \times \mathrm{Stack} \times \mathrm{Heap}_{\mathsf{cl}}$$
$$\rho \in \mathrm{Env} = \mathrm{Var} \to \mathrm{Loc}_{\mathsf{cl}}$$
$$c \in \mathrm{Clos} = \mathrm{Exp} \times \mathrm{Env}$$
$$\sigma \in \mathrm{Stack} = \mathsf{List}\left[\mathrm{Loc}_{\mathsf{cl}} + \mathrm{Clos}\right]$$
$$l \in \mathrm{Loc}_{\mathsf{cl}} \qquad \mu_{\mathsf{cl}} \in \mathrm{Heap}_{\mathsf{cl}} = \mathrm{Loc}_{\mathsf{cl}} \to \mathrm{Clos}$$

$$\frac{l = \rho(x) \qquad \mu_{\mathsf{cl}}(l) = v}{(\langle x, \rho\rangle, \sigma, \mu_{\mathsf{cl}}) \longrightarrow_{\mathcal{L}} (v, \sigma, \mu_{\mathsf{cl}})} \quad (\textsc{Var1})$$

$$\frac{l = \rho(x) \qquad \mu_{\mathsf{cl}}(l) = c \notin \mathrm{Val}}{(\langle x, \rho\rangle, \sigma, \mu_{\mathsf{cl}}) \longrightarrow_{\mathcal{L}} (c, \mathsf{mark}(l) :: \sigma, \mu_{\mathsf{cl}})} \quad (\textsc{Var2})$$

$$(\langle (MN), \rho\rangle, \sigma, \mu_{\mathsf{cl}}) \longrightarrow_{\mathcal{L}} (\langle M, \rho\rangle, \mathsf{arg}(\langle N, \rho\rangle) :: \sigma, \mu_{\mathsf{cl}}) \quad (\textsc{App})$$

$$\frac{l \notin \mathrm{dom}(\mu_{\mathsf{cl}}) \qquad \mu'_{\mathsf{cl}} = \mu_{\mathsf{cl}}[l \mapsto c]}{(\langle \lambda x.M, \rho\rangle, \mathsf{arg}(c) :: \sigma, \mu_{\mathsf{cl}}) \longrightarrow_{\mathcal{L}} (\langle M, \rho[x \mapsto l]\rangle, \sigma, \mu'_{\mathsf{cl}})} \quad (\textsc{Call})$$

$$\frac{\mu'_{\mathsf{cl}} = \mu_{\mathsf{cl}}[l \mapsto v]}{(v, \mathsf{mark}(l) :: \sigma, \mu_{\mathsf{cl}}) \longrightarrow_{\mathcal{L}} (v, \sigma, \mu'_{\mathsf{cl}})} \quad (\textsc{Update})$$

Figure 2. Definition of the $\mathcal{L}$ (lazy) machine

of such a derivation. It differs from Sestoft's Mark 2 machine in that it is restricted to lambda terms. In addition, we use the Var1 rule, an optimization that Sestoft mentions. This rule avoids pushing a marker on the stack when the variable is associated with a value. The shading in Figure 2 highlights the differences between $\mathcal{K}$ and $\mathcal{L}$. There are two kinds of things on the stack, so we use a disjoint union with mark (= inl) for markers and arg (= inr) for operand closures.

2.1. Properties of the $\mathcal{L}$ machine

In $\mathcal{L}$ there is no way to create circularity within terms and the environment, i.e., there is no way for a variable to be bound to a term that contains a reference to the same variable. This is because there is no recursive *let* or the equivalent.

Proposition 1. Given any state s of the form $(\langle x, \rho\rangle, \sigma, \mu_{\text{cl}})$ in an execution sequence on $\mathcal{L}$, if $\rho(x) = l$ and $\mu_{\text{cl}}(l) = \langle N, \rho'\rangle$, then there is no variable y in N such that $\rho'(y) = l$.

Lemma 1. For any state s in an execution sequence on $\mathcal{L}$, there is at most one marker on the stack containing location l.

Proof. If $\mathsf{mark}(l)$ is on the stack, the closure in $\mu_{\text{cl}}(l)$ is currently being evaluated. By Proposition 1, there is no variable x in $\mu_{\text{cl}}(l)$ such that $\rho(x) = l$. Thus l cannot be pushed on the stack during the evaluation of $\mu_{\text{cl}}(l)$. Once the evaluation of $\mu_{\text{cl}}(l)$ is finished, $\mathsf{mark}(l)$ is popped from the stack.

In the CALL rule, a new location in the heap is allocated and initialized. Later on, the location may be written to in the UPDATE rule. There is at most one write to each location, which corresponds to $\mathcal{L}$ being lazy.

Lemma 2. Given any execution sequence, for each $l \in \text{dom}(\mu_{\text{cl}})$, there is at most one state s' in the sequence of the form $(v, \mathsf{mark}(l) :: \sigma, \mu_{\text{cl}})$.

Proof. Suppose state s' is the first state in the execution sequence of the form $(v, \mathsf{mark}(l) :: \sigma, \mu_{\text{cl}})$. The next step is an UPDATE that places v in $\mu_{\text{cl}}(l)$. Thereafter, when l is accessed through some $\rho(x)$, rule VAR1 will apply and not VAR2. Thus l is never again in a marker that is pushed on the stack and therefore never again updated. But what about the case when there has already been a marker with l on the stack when l was first updated? This can not happen due to Lemma 1.

3. Collapsed Markers

Execution traces of $\mathcal{L}$ reveal situations in which the stack contains sequences of markers. For example, the term $(\lambda z.(\lambda y.z(yz))z)(\lambda x.x)$ demonstrates this behavior. The following is the execution of this term on $\mathcal{L}$. We use the abbreviations $\rho_0 \equiv [z \mapsto l_0]$, $\rho_1 \equiv \rho_0[y \mapsto l_1]$, $\mu_0 \equiv [l_0 \mapsto \langle \lambda x.x, []\rangle]$, $\mu_1 \equiv \mu_0[l_1 \mapsto \langle z, \rho_0\rangle]$, and $\mu_2 \equiv \mu_1[l_2 \mapsto \langle yz, \rho_1\rangle]$.

$(\langle(\lambda z.(\lambda y.z(yz))z)(\lambda x.x), []\rangle, (), [])$
$\to_{\mathcal{L}} (\langle\lambda z.(\lambda y.z(yz))z, []\rangle, \mathsf{arg}(\langle\lambda x.x, []\rangle) :: (), [])$
$\to_{\mathcal{L}} (\langle(\lambda y.z(yz))z, \rho_0\rangle, (), \mu_0)$
$\to_{\mathcal{L}} (\langle\lambda y.z(yz), \rho_0\rangle, \mathsf{arg}(\langle z, \rho_0\rangle) :: (), \mu_0)$
$\to_{\mathcal{L}} (\langle z(yz), \rho_1\rangle, (), \mu_1)$
$\to_{\mathcal{L}} (\langle z, \rho_1\rangle, \mathsf{arg}(\langle yz, \rho_1\rangle) :: (), \mu_1)$
$\to_{\mathcal{L}} (\langle\lambda x.x, []\rangle, \mathsf{arg}(\langle yz, \rho_1\rangle) :: (), \mu_1)$
$\to_{\mathcal{L}} (\langle x, [x{=}l_2]\rangle, (), \mu_2)$
$\to_{\mathcal{L}} (\langle yz, \rho_1\rangle, \mathsf{mark}(l_2) :: (), \mu_2)$
$\to_{\mathcal{L}} (\langle y, \rho_1\rangle, \mathsf{arg}(\langle z, \rho_1\rangle) :: \mathsf{mark}(l_2) :: (), \mu_2)$
$\to_{\mathcal{L}} (\langle z, \rho_0\rangle, \mathsf{mark}(l_1) :: \mathsf{arg}(\langle z, \rho_1\rangle) :: \mathsf{mark}(l_2) :: (), \mu_2)$
$\to_{\mathcal{L}} (\langle\lambda x.x, []\rangle, \mathsf{mark}(l_1) :: \mathsf{arg}(\langle z, \rho_1\rangle) :: \mathsf{mark}(l_2) :: (), \mu_0[l_1 \mapsto \langle z, \rho_0\rangle, l_2 \mapsto \langle yz, \rho_1\rangle])$
$\to_{\mathcal{L}} (\langle\lambda x.x, []\rangle, \mathsf{arg}(\langle z, \rho_1\rangle) :: \mathsf{mark}(l_2) :: (), \mu_0[l_1 \mapsto \langle\lambda x.x, []\rangle, l_2 \mapsto \langle yz, \rho_1\rangle])$
$\to_{\mathcal{L}} (\langle x, [x{=}l_3]\rangle, \mathsf{mark}(l_2) :: (), \mu_0[l_1 \mapsto \langle\lambda x.x, []\rangle, l_2 \mapsto \langle yz, \rho_1\rangle, l_3 \mapsto \langle z, \rho_1\rangle])$
$\to_{\mathcal{L}} (\langle z, \rho_1\rangle, \mathsf{mark}(l_3) :: \mathsf{mark}(l_2) :: (), \mu_0[l_1 \mapsto \langle\lambda x.x, []\rangle, l_2 \mapsto \langle yz, \rho_1\rangle, l_3 \mapsto \langle z, \rho_1\rangle])$
$\to_{\mathcal{L}} (\langle\lambda x.x, []\rangle, \mathsf{mark}(l_3) :: \mathsf{mark}(l_2) :: (), \mu_0[l_1 \mapsto \langle\lambda x.x, []\rangle, l_2 \mapsto \langle yz, \rho_1\rangle, l_3 \mapsto \langle z, \rho_1\rangle])$
$\to_{\mathcal{L}} (\langle\lambda x.x, []\rangle, \mathsf{mark}(l_2) :: (), \mu_0[l_1 \mapsto \langle\lambda x.x, []\rangle, l_2 \mapsto \langle yz, \rho_1\rangle, l_3 \mapsto \langle\lambda x.x, []\rangle])$
$\to_{\mathcal{L}} (\langle\lambda x.x, []\rangle, (), \mu_0[l_1 \mapsto \langle\lambda x.x, []\rangle, l_2 \mapsto \langle\lambda x.x, []\rangle, l_3 \mapsto \langle\lambda x.x, []\rangle])$

When a sequence of markers is popped from the stack, the same value is assigned to each heap location pointed to by the markers. In $\mathcal{L}$ there is a one-to-one correspondence between these heap locations and the closures created for operands in the CALL rule. So these closures receive the same value. The optimization is to avoid creating sequences of markers in the first place by sharing the first marker and result location among closures that receive the same value.

In the interest of simplifying the proof of correctness, we implement this optimization in two steps. The first step adds a level of indirection between environments and closures. The second step introduces sharing in place of pushing redundant markers.

3.1. The Intermediate $\underline{\mathcal{C}}$ machine

The first step of the transformation is realized in the machine $\underline{\mathcal{C}}$, defined in Figure 3. The difference between this machine and $\mathcal{L}$ is that, in the environment, pointers to closures are replaced with pointers to pointers to closures. This adds a small constant space and time overhead, as discussed in Section 6.

During execution of the same term, $\mathcal{L}$ and $\underline{\mathcal{C}}$ run in lockstep. We capture this idea by defining when a $\mathcal{L}$ state is bisimilar to a $\underline{\mathcal{C}}$ state

$$\longrightarrow_{\underline{\mathcal{C}}} : \text{State} \to \text{State}$$
$$\text{State} = \text{Clos} \times \text{Stack} \times \text{Heap}$$
$$\rho \in \text{Env} = \text{Var} \to \text{Loc}_{\text{Loc}_{\text{cl}}}$$
$$c \in \text{Clos} = \text{Exp} \times \text{Env}$$
$$\sigma \in \text{Stack} = \mathsf{List}[\text{Loc}_{\text{cl}} + \text{Clos}]$$
$$l \in \text{Loc}_{\text{cl}} \quad \mu_{\text{cl}} \in \text{Heap}_{\mathsf{cl}} = \text{Loc}_{\text{cl}} \to \text{Clos}$$
$$r \in \text{Loc}_{\text{Loc}_{\text{cl}}} \quad \mu_{\text{loc}} \in \text{Heap}_{\text{Loc}_{\text{cl}}} = \text{Loc}_{\text{Loc}_{\text{cl}}} \to \text{Loc}_{\text{cl}}$$
$$\mu \in \text{Heap} = \text{Heap}_{\mathsf{cl}} \times \text{Heap}_{\text{Loc}_{\text{cl}}}$$

$$\frac{r = \rho(x) \qquad l = \mu_{\text{loc}}(r) \qquad \mu_{\text{cl}}(l) = v}{(\langle x, \rho\rangle, \sigma, \mu) \longrightarrow_{\underline{\mathcal{C}}} (v, \sigma, \mu)} \quad \text{(Var1)}$$

$$\frac{r = \rho(x) \qquad l = \mu_{\text{loc}}(r) \qquad \mu_{\text{cl}}(l) = c \notin \text{Val}}{(\langle x, \rho\rangle, \sigma, \mu) \longrightarrow_{\underline{\mathcal{C}}} (c, \mathsf{mark}(l) :: \sigma, \mu)} \quad \text{(Var2)}$$

$$(\langle (MN), \rho\rangle, \sigma, \mu) \longrightarrow_{\underline{\mathcal{C}}} (\langle M, \rho\rangle, \mathsf{arg}(\langle N, \rho\rangle) :: \sigma, \mu) \quad \text{(App)}$$

$$\frac{\begin{array}{cc} l \notin \text{dom}(\mu_{\text{cl}}) & \mu'_{\text{cl}} = \mu_{\text{cl}}[l \mapsto c] \\ r \notin \text{dom}(\mu_{\text{loc}}) & \mu'_{\text{loc}} = \mu_{\text{loc}}[r \mapsto l] \end{array}}{(\langle \lambda x.M, \rho\rangle, \mathsf{arg}(c) :: \sigma, \mu) \longrightarrow_{\underline{\mathcal{C}}} (\langle M, \rho[x \mapsto r]\rangle, \sigma, \langle \mu'_{\text{cl}}, \mu'_{\text{loc}}\rangle)} \quad \text{(Call)}$$

$$\frac{\mu'_{\text{cl}} = \mu_{\text{cl}}[l \mapsto v]}{(v, \mathsf{mark}(l) :: \sigma, \mu) \longrightarrow_{\underline{\mathcal{C}}} (v, \sigma, \langle \mu'_{\text{cl}}, \mu_{\text{loc}}\rangle)} \quad \text{(Update)}$$

Figure 3. Definition of the $\underline{\mathcal{C}}$ (intermediate) machine (for proof of correctness, only)

(Figure 4). For such a small difference between $\mathcal{L}$ and $\underline{\mathcal{C}}$, it is perhaps overkill to spell out this bisimulation in the proof of correctness. However, a couple interesting issues are brought to light and similar ideas are used in a more complicated setting in the next section.

Theorem 1. The relation $\sim_{\underline{\mathcal{C}}}$ is a bisimulation. If $s \sim_{\underline{\mathcal{C}}} s'$, then

1. If $s \longrightarrow_{\mathcal{L}} t$, then $s' \longrightarrow_{\underline{\mathcal{C}}} t'$ and $t \sim_{\underline{\mathcal{C}}} t'$.
2. If $s' \longrightarrow_{\underline{\mathcal{C}}} t'$, then $s \longrightarrow_{\mathcal{L}} t$ and $t \sim_{\underline{\mathcal{C}}} t'$.

$$\frac{c \sim^{c}_{\underline{\mathcal{C}}} c' \qquad \sigma \sim^{\sigma}_{\underline{\mathcal{C}}} \sigma'}{(c, \sigma, \mu_{\mathrm{cl}}) \sim_{\underline{\mathcal{C}}} (c', \sigma', \langle \mu'_{\mathrm{cl}}, \mu'_{\mathrm{loc}} \rangle)}$$

$$\frac{\rho \sim^{\rho}_{\underline{\mathcal{C}}} \rho'}{\langle M, \rho \rangle \sim^{c}_{\underline{\mathcal{C}}} \langle M, \rho' \rangle} \qquad \frac{\mu_{\mathrm{cl}}(l) \sim^{c}_{\underline{\mathcal{C}}} \mu'_{\mathrm{cl}}(l')}{l \sim^{\mu}_{\underline{\mathcal{C}}} l'}$$

$$\frac{\mathrm{dom}(\rho) = \mathrm{dom}(\rho') \qquad \forall x \in \mathrm{dom}(\rho).\ \rho(x) \sim^{\mu}_{\underline{\mathcal{C}}} \mu'_{\mathrm{loc}}(\rho'(x))}{\rho \sim^{\rho}_{\underline{\mathcal{C}}} \rho'}$$

$$\frac{}{() \sim^{\sigma}_{\underline{\mathcal{C}}} ()} \qquad \frac{l \sim^{\mu}_{\underline{\mathcal{C}}} l' \qquad \sigma \sim^{\sigma}_{\underline{\mathcal{C}}} \sigma'}{\mathsf{mark}(l) :: \sigma \sim^{\sigma}_{\underline{\mathcal{C}}} \mathsf{mark}(l') :: \sigma'} \qquad \frac{c \sim^{c}_{\underline{\mathcal{C}}} c' \qquad \sigma \sim^{\sigma}_{\underline{\mathcal{C}}} \sigma'}{\mathsf{arg}(c) :: \sigma \sim^{\sigma}_{\underline{\mathcal{C}}} \mathsf{arg}(c') :: \sigma'}$$

Figure 4. Bisimilarity relation $\sim_{\underline{\mathcal{C}}}$ between $\mathcal{L}$ and $\underline{\mathcal{C}}$

Proof. Since $s \sim_{\underline{\mathcal{C}}} s'$, the rule with the same name for each machine gives us t and t', respectively. Then by cases on the rules of $\mathcal{L}$ and $\underline{\mathcal{C}}$, it is easy to see that $t \sim_{\underline{\mathcal{C}}} t'$.

The correctness of $\underline{\mathcal{C}}$ follows immediately from the theorem.

Corollary 1. $\underline{\mathcal{C}}$ is correct with respect to $\mathcal{L}$. That is, if the final state in an execution sequence of expression M on $\mathcal{L}$ is t, then there exists a final state t' of an execution sequence of M on $\underline{\mathcal{C}}$ such that $t \sim_{\underline{\mathcal{C}}} t'$.

We formulate correctness in terms of the relation $\sim_{\underline{\mathcal{C}}}$ because the result of an execution sequence is a closure and the expression part of the closure may contain variables that refer to the environment part. These in turn contain pointers into the heap. Therefore the equivalence of the results cannot be expressed solely in terms of the syntax of the resulting expressions. It depends on the whole machine state.

3.2. The $\mathcal{C}$ machine

The second step in the transformation focuses on the Var2 rule, where markers are pushed on the stack. If the top of the stack is not a marker, we proceed as before and push a marker on the stack. If the top is a marker, we introduce sharing between the environment and the closure associated with the marker. Thus, in Figure 5, we split the Var2 rule into the following two cases.

$\longrightarrow_{\mathcal{C}}$: State $\to$ State
State is the same as for $\underline{\mathcal{C}}$.

$$\frac{r = \rho(x) \quad l = \mu_{\text{loc}}(r) \quad \mu_{\text{cl}}(l) = v}{(\langle x, \rho\rangle, \sigma, \mu) \longrightarrow_{\mathcal{C}} (v, \sigma, \mu)} \tag{Var1}$$

$$\frac{\begin{array}{c}\sigma = () \ \vee \ \sigma = \mathsf{arg}(-) :: \sigma' \\ r = \rho(x) \quad l = \mu_{\text{loc}}(r) \quad \mu_{\text{cl}}(l) = c \notin \text{Val}\end{array}}{(\langle x, \rho\rangle, \sigma, \mu) \longrightarrow_{\mathcal{C}} (c, \mathsf{mark}(l) :: \sigma, \mu)} \tag{Var2a}$$

$$\frac{\begin{array}{c}r = \rho(x) \quad l = \mu_{\text{loc}}(r) \quad \mu_{\text{cl}}(l) = c \notin \text{Val} \\ \mu'_{\text{loc}} = \mu_{\text{loc}}[r \mapsto l']\end{array}}{(\langle x, \rho\rangle, \mathsf{mark}(l') :: \sigma, \mu) \longrightarrow_{\mathcal{C}} (c, \mathsf{mark}(l') :: \sigma, \langle \mu_{\text{cl}}, \mu'_{\text{loc}}\rangle)} \tag{Var2b}$$

$$(\langle (MN), \rho\rangle, \sigma, \mu) \longrightarrow_{\mathcal{C}} (\langle M, \rho\rangle, \mathsf{arg}(\langle N, \rho\rangle) :: \sigma, \mu) \tag{App}$$

$$\frac{\begin{array}{cc}l \notin \text{dom}(\mu_{\text{cl}}) & \mu'_{\text{cl}} = \mu_{\text{cl}}[l \mapsto c] \\ r \notin \text{dom}(\mu_{\text{loc}}) & \mu'_{\text{loc}} = \mu_{\text{loc}}[r \mapsto l]\end{array}}{(\langle \lambda x.M, \rho\rangle, \mathsf{arg}(c) :: \sigma, \mu) \longrightarrow_{\mathcal{C}} (\langle M, \rho[x \mapsto r]\rangle, \sigma, \langle \mu'_{\text{cl}}, \mu'_{\text{loc}}\rangle)} \tag{Call}$$

$$\frac{\mu'_{\text{cl}} = \mu_{\text{cl}}[l \mapsto v]}{(v, \mathsf{mark}(l) :: \sigma, \mu) \longrightarrow_{\mathcal{C}} (v, \sigma, \langle \mu'_{\text{cl}}, \mu_{\text{loc}}\rangle)} \tag{Update}$$

Figure 5. Definition of the $\mathcal{C}$ (collapsed markers) machine

Var2a: The stack is empty or the top of the stack is not a marker. Just as in the Var2 rule, a new marker is pushed on the stack.

Var2b: The top of the stack is a marker. The environment is updated to point to the shared result location.

The second transformation is from $\underline{\mathcal{C}}$ to $\mathcal{C}$. Next, we prove that it preserves correctness. The Var2a rule is similar to the Var2 rule, so we focus on the correctness of the Var2b rule. The difference in the Var2b rule is that instead of pushing a new marker $\mathsf{mark}(l)$ on the stack, the location r, which is $\rho(x)$, is changed to point to the location l' from the current marker. To see why this change preserves correctness, consider the Update rule, which consumes markers. When there are two adjacent markers on the stack, the same value is assigned to both of the associated locations, as stated in the following lemma.

Lemma 3.

$$(v, \mathsf{mark}(l_2) :: \mathsf{mark}(l_1) :: \sigma, \mu) \longrightarrow^2_{\underline{\mathcal{C}}} (v, \sigma, \mu'_{\mathrm{cl}})$$
where
$$\mu'_{\mathrm{cl}}(l_2) = \mu'_{\mathrm{cl}}(l_1) = v$$

Proof. The UPDATE rule is the only rule that matches the above states, and its effect is to assign v to the location l, pop the stack, and continue with v still in the first position of the state.

This lemma can be applied multiple times to show that for a contiguous sequence of markers, the same value is assigned to each of the associated locations.

We define an equivalence relation $\approx$ between locations in μ_{cl} that are in a marker sequence. We use $[l]_{\approx}$ to denote the representative of the equivalence class containing l, which we choose to be the first l in a sequence of markers to be pushed on the stack. We abuse the notation $\approx[l \mapsto l']$ to mean the reflexive, symmetric, transitive closure of $(\approx \cup \{\langle l, l' \rangle\})$ and that the representative of l in $\approx[l \mapsto l']$ is $[l']_{\approx}$. Next, we augment the state definition of $\underline{\mathcal{C}}$ to construct the $\approx$ relation. Most of the rules stay the same, passing along $\approx$ unchanged, but we split the VAR2 rule. When the expression is a variable bound to a non-value and when the top of the stack is not a marker, l is put in an equivalence class by itself.

$$\frac{\sigma = () \vee \sigma = \mathsf{arg}(-) :: \sigma' \qquad r = \rho(x) \qquad l = \mu_{\mathrm{loc}}(r) \qquad \mu_{\mathrm{cl}}(l) = c \notin \mathrm{Val}}{(\langle x, \rho \rangle, \sigma, \mu, \approx) \longrightarrow_{\underline{\mathcal{C}}} (c, \mathsf{mark}(l) :: \sigma, \mu, \approx[l \mapsto l])} \ \textsc{Var2A}{\approx}$$

In the case when the top of the stack is a marker, the relation $\approx$ is extended so that the representative of l is $[l']_{\approx}$.

$$\frac{r = \rho(x) \qquad l = \mu_{\mathrm{loc}}(r) \qquad \mu_{\mathrm{cl}}(l) = c \notin \mathrm{Val}}{(\langle x, \rho \rangle, \mathsf{mark}(l') :: \sigma, \mu, \approx) \longrightarrow_{\underline{\mathcal{C}}} (c, \mathsf{mark}(l) :: \mathsf{mark}(l') :: \sigma, \mu, \approx[l \mapsto l'])} \ \textsc{Var2B}{\approx}$$

Next we connect Lemma 3 to the relation $\approx$.

Lemma 4. In a state s of an execution sequence, if $l \approx l'$ and if $\forall l \in [l]_{\approx} \,.\, \mu_{\mathrm{cl}}(l) \in \mathrm{Val}$, then $\mu_{\mathrm{cl}}(l) = \mu_{\mathrm{cl}}(l')$. Also, $\mu_{\mathrm{cl}}(l) = \mu_{\mathrm{cl}}(l')$ for every state in the execution sequence after s.

Proof. If $l \approx l'$ in some some state s, and if $\forall l \in [l]_{\approx} \,.\, \mu_{\mathrm{cl}}(l) \in \mathrm{Val}$, then the state transitions described in Lemma 3 must have occurred

$$\frac{\approx \vdash c \sim^{c}_{\mathcal{C}} c' \qquad \approx \vdash \sigma \sim^{\sigma}_{\mathcal{C}} \sigma'}{(c, \sigma, \langle \mu_{\text{cl}}, \mu_{\text{loc}} \rangle, \approx) \sim_{\mathcal{C}} (c', \sigma', \langle \mu'_{\text{cl}}, \mu'_{\text{loc}} \rangle)}$$

$$\frac{\approx \vdash \rho \sim^{\rho}_{\mathcal{C}} \rho'}{\approx \vdash \langle M, \rho \rangle \sim^{c}_{\mathcal{C}} \langle M, \rho' \rangle} \qquad \frac{\approx \vdash \mu_{\text{cl}}([l]_{\approx}) \sim^{c}_{\mathcal{C}} \mu'_{\text{cl}}(l')}{\approx \vdash l \sim^{\mu}_{\mathcal{C}} l'}$$

$$\frac{\text{dom}(\rho) = \text{dom}(\rho') \qquad \forall x \in \text{dom}(\rho).\ \approx \vdash \mu_{\text{loc}}(\rho(x)) \sim^{\mu}_{\mathcal{C}} \mu'_{\text{loc}}(\rho'(x))}{\approx \vdash \rho \sim^{\rho}_{\mathcal{C}} \rho'}$$

$$\frac{}{\approx \vdash () \sim^{\sigma}_{\mathcal{C}} ()}$$

$$\frac{\approx \vdash l \sim^{\mu}_{\mathcal{C}} l' \qquad \approx \vdash \sigma \sim^{\sigma}_{\mathcal{C}} \sigma'}{\approx \vdash \mathsf{mark}(l) :: \sigma \sim^{\sigma}_{\mathcal{C}} \mathsf{mark}(l') :: \sigma'} \qquad \frac{\approx \vdash c \sim^{c}_{\mathcal{C}} c' \qquad \approx \vdash \sigma \sim^{\sigma}_{\mathcal{C}} \sigma'}{\approx \vdash \mathsf{arg}(c) :: \sigma \sim^{\sigma}_{\mathcal{C}} \mathsf{arg}(c') :: \sigma'}$$

$$\frac{\approx \vdash l_2 \sim^{\mu}_{\mathcal{C}} l' \qquad \approx \vdash \mathsf{mark}(l_1) :: \sigma \sim^{\sigma}_{\mathcal{C}} \mathsf{mark}(l') :: \sigma'}{\approx \vdash \mathsf{mark}(l_2) :: \mathsf{mark}(l_1) :: \sigma \sim^{\sigma}_{\mathcal{C}} \mathsf{mark}(l') :: \sigma'} \quad (\textsc{Push})$$

$$\frac{\approx \vdash l_2 \sim^{\mu}_{\mathcal{C}} l' \qquad \approx \vdash \mathsf{mark}(l_2) :: \mathsf{mark}(l_1) :: \sigma \sim^{\sigma}_{\mathcal{C}} \mathsf{mark}(l') :: \sigma'}{\approx \vdash \mathsf{mark}(l_1) :: \sigma \sim^{\sigma}_{\mathcal{C}} \mathsf{mark}(l') :: \sigma'} \quad (\textsc{Pop})$$

Figure 6. Bisimilarity relation $\sim_{\mathcal{C}}$ between $\underline{\mathcal{C}}$ and $\mathcal{C}$

prior to s in the execution sequence. Therefore l and l' have been assigned the same value in some previous state s'. There is at most one assignment to each heap location, so $\mu_{\text{cl}}(l) = \mu_{\text{cl}}(l')$ holds for all states after s' including the current state s.

In Theorem 2 we demonstrate a weak bisimilarity between machine states of $\underline{\mathcal{C}}$ and $\mathcal{C}$ (where the state of $\underline{\mathcal{C}}$ has been augmented with the relation $\approx$). The relation is called "weak" because the two machines no longer run in lockstep. During transitions in which $\underline{\mathcal{C}}$ pushes or pops redundant marks, $\mathcal{C}$ makes no transitions.

From Lemma 3, we can make the simplification that location l of a $\underline{\mathcal{C}}$ state is bisimilar to location l' of a $\mathcal{C}$ state when $[l]_{\approx}$ is bisimilar to l'. Also, the stack σ of a $\underline{\mathcal{C}}$ state is bisimilar to the stack σ' of a $\mathcal{C}$ state when the only difference is the presence of redundant markers on σ. The relation $\sim_{\mathcal{C}}$ between states of $\underline{\mathcal{C}}$ and $\mathcal{C}$ is defined in Figure 6.

Theorem 2. The relation $\sim_{\mathcal{C}}$ between states in an execution sequence on $\underline{\mathcal{C}}$ and on $\mathcal{C}$ has the following property. If $s \sim_{\mathcal{C}} s'$, then

1. If $s \longrightarrow_{\underline{\mathcal{C}}} t$, then either $t \sim_{\mathcal{C}} s'$ or there exists a t' such that $s' \longrightarrow_{\mathcal{C}} t'$ and $t \sim_{\mathcal{C}} t'$.
2. If $s' \longrightarrow_{\mathcal{C}} t'$, then there exists a t such that $s \longrightarrow^{+}_{\underline{\mathcal{C}}} t$ and $t \sim_{\mathcal{C}} t'$.

Proof. For all but the states matching the source pattern for VAR2 and UPDATE, it is obvious that the two machines run in lockstep and that the bisimilarity is preserved.

For the VAR2 situation, either both machines push markers to bisimilar locations and bisimilarity is preserved, or $\underline{\mathcal{C}}$ pushes a marker and $\mathcal{C}$ does not. In this case, the top of the stack is a marker. Therefore we can apply the POP rule to show that the stacks are still bisimilar. Also, the update of $\mu'_{\mathrm{loc}}[r \mapsto l']$ for $\mathcal{C}$ does not interfere with the bisimilarity because we know that $l \approx l'$.

For the UPDATE situation, either both machines pop markers from the stack, in which case it is easy to see that bisimilarity is preserved, or else $\underline{\mathcal{C}}$ pops a marker from the stack while $\mathcal{C}$ does nothing. Then, we apply the PUSH rule to show that the stacks are still bisimilar.

Corollary 2. $\mathcal{C}$ is correct with respect to $\underline{\mathcal{C}}$. That is, if the final state in an execution sequence of expression M on $\underline{\mathcal{C}}$ is t, then there exists a final state t' of an execution sequence of M on $\mathcal{C}$ such that $t \sim_{\mathcal{C}} t'$.

3.3. PROPERTIES OF THE $\mathcal{C}$ MACHINE

The $\mathcal{C}$ machine provides the following guarantee.

Theorem 3. For any state in an execution sequence on $\mathcal{C}$, there will never be adjacent markers on the stack.

Proof. The proof is by inspection of the rules of $\mathcal{C}$, specifically, VAR2A and VAR2B.

Another interesting property is that for each location in the heaps μ_{cl} and μ_{loc}, there is at most one assignment (other than initialization). The single assignment for μ_{cl} on $\mathcal{C}$ is a property that is inherited from $\mathcal{L}$. For the μ_{loc}, we state the following theorem.

Theorem 4. Given any execution sequence, for each $r \in \mu_{\mathrm{loc}}$, there is at most one state s' in the sequence of the form $(\langle x, \rho\rangle, \mathsf{mark}(l') :: \sigma, \mu)$ where $r = \rho(x)$, and $\mu_{\mathrm{cl}}(\mu_{\mathrm{loc}}(r)) \notin \mathrm{Val}$.

Proof. Suppose s' is the first state in the execution sequence of the form $(\langle x, \rho\rangle, \mathsf{mark}(l') :: \sigma, \langle \mu_{\mathrm{cl}}, \mu_{\mathrm{loc}}\rangle)$ where $r = \rho(x)$ and $\mu_{\mathrm{cl}}(\mu_{\mathrm{loc}}(r)) \notin \mathrm{Val}$. An assignment to r occurs on the next step, causing r to share the result location with some other closure, which is currently being evaluated. Once the closure is evaluated, $\mu_{\mathrm{cl}}(\mu_{\mathrm{loc}}(r))$ is updated with its resulting value. Thereafter, whenever r is accessed (through an environment lookup), rule VAR1 applies and not VAR2B. For the subsequence of states between when r is first assigned and when the value is assigned to $\mu_{\mathrm{cl}}(\mu_{\mathrm{loc}}(r))$, no reads of r occur because of Lemma 1.

The benefit of the collapsed markers optimization is particularly apparent for the term $((Y(\lambda z.(\lambda y.((\lambda x.x)(zy)))))(\lambda x.x))$. With $\mathcal{L}$, the stack grows by approximately 7 markers per 100 steps, but with $\mathcal{C}$, the stack contains at most 5 items. (See Section 6 for details.)

4. Short-Circuiting applied to the $\mathcal{L}$ machine

In this section we construct a call-by-need machine, based on $\mathcal{L}$, that applies short-circuiting of operand variable dereferencing. This optimization was used in early implementations of Algol 60 [13] and is proved correct when applied to $\mathcal{K}$ in Wand [18]. The key observation is that $\mathcal{K}$ grows the environment unnecessarily in the APP rule. Consider the following sequence of transitions.

$$
\begin{array}{l}
(\langle \lambda x.Mx, [] \rangle, \langle N, [] \rangle :: \sigma) \\
\longrightarrow_{\mathcal{K}} (\langle Mx, [x \mapsto \langle N, [] \rangle] \rangle, \sigma) \\
\longrightarrow_{\mathcal{K}} (\langle M, [x \mapsto \langle N, [] \rangle] \rangle, \langle x, [x \mapsto \langle N, []] \rangle \rangle :: \sigma)
\end{array}
$$

In the stack, the current environment is larger than the environment in the closure associated with x, that is, $[x \mapsto \langle N, [] \rangle]$ is larger than $[]$. So if possible, it would save space to push the closure associated with x on the stack instead of x and the current environment. Intuitively, the reason this is okay is that the lambda calculus is a pure functional language, so the closure associated with x cannot change and therefore it does not matter when it is dereferenced. Therefore we include in $\mathcal{S}$ the APPVAR rule that performs eager lookup of variables in operand position. Here is the same sequence with the new rule:

$$
\begin{array}{l}
(\langle \lambda x.Mx, [] \rangle, \langle N, [] \rangle :: \sigma) \\
\longrightarrow_{\mathcal{S}} (\langle Mx, [x \mapsto \langle N, [] \rangle] \rangle, \sigma) \\
\longrightarrow_{\mathcal{S}} (\langle M, [x \mapsto \langle N, [] \rangle] \rangle, \langle N, [] \rangle :: \sigma)
\end{array}
$$

$$\longrightarrow_{\mathcal{S}} : \text{State} \to \text{State}$$
$$\text{State} = \text{Clos} \times \text{Stack} \times \text{Heap}_{\mathsf{cl}}$$
$$\rho \in \text{Env} = \text{Var} \to \text{Loc}_{\text{cl}}$$
$$c \in \text{Clos} ::= \langle (MN), \rho \rangle \mid \text{Val} \quad \subset \quad \text{Exp} \times \text{Env}$$
$$\sigma \in \text{Stack} = \mathsf{List}\left[\text{Loc}_{\text{cl}} + \text{Loc}_{\text{cl}}\right]$$
$$l \in \text{Loc}_{\text{cl}} \quad \mu_{\text{cl}} \in \text{Heap}_{\mathsf{cl}} = \text{Loc}_{\text{cl}} \to \text{Clos}$$

$$\frac{l = \rho(x) \qquad \mu_{\text{cl}}(l) = v}{(\langle x, \rho \rangle, \sigma, \mu_{\text{cl}}) \longrightarrow_{\mathcal{S}} (v, \sigma, \mu_{\text{cl}})} \quad (\textsc{Var1})$$

$$\frac{l = \rho(x) \qquad \mu_{\text{cl}}(l) = c \notin \text{Val}}{(\langle x, \rho \rangle, \sigma, \mu_{\text{cl}}) \longrightarrow_{\mathcal{S}} (c, \mathsf{mark}(l) :: \sigma, \mu_{\text{cl}})} \quad (\textsc{Var2})$$

$$\frac{l \notin \text{dom}(\mu_{\text{cl}}) \qquad \mu'_{\text{cl}} = \mu_{\text{cl}}[l \mapsto \langle N, \rho \rangle]}{(\langle (MN), \rho \rangle, \sigma, \mu_{\text{cl}}) \longrightarrow_{\mathcal{S}} (\langle M, \rho \rangle, \mathsf{arg}(\, l \,) :: \sigma, \mu'_{\text{cl}})} \quad (\textsc{App})$$

$$\frac{l = \rho(x)}{(\langle (Mx), \rho \rangle, \sigma, \mu_{\text{cl}}) \longrightarrow_{\mathcal{S}} (\langle M, \rho \rangle, \mathsf{arg}(l) :: \sigma, \mu_{\text{cl}})} \quad (\textsc{AppVar})$$

$$(\langle \lambda x.M, \rho \rangle, \mathsf{arg}(\, l \,) :: \sigma, \mu_{\text{cl}}) \longrightarrow_{\mathcal{S}} (\langle M, \rho[x \mapsto \, l \,] \rangle, \sigma, \mu_{\text{cl}}) \quad (\textsc{Call})$$

$$\frac{\mu'_{\text{cl}} = \mu_{\text{cl}}[l \mapsto v]}{(v, \mathsf{mark}(l) :: \sigma, \mu_{\text{cl}}) \longrightarrow_{\mathcal{S}} (v, \sigma, \mu'_{\text{cl}})} \quad (\textsc{Update})$$

Figure 7. Definition of the $\mathcal{S}$ (short-circuiting) machine derived from the $\mathcal{L}$ machine

An implication of short-circuiting is that the term in a closure is restricted to lambda abstractions and applications; variables do not appear. This follows by inspection of the rules of the $\mathcal{S}$ machine (Figure 7). Closures are created in the Update and App rules and in neither case can the term be a variable.

There is, however, one complication in the call-by-need version of short-circuiting. The AppVar rule, which implements short-circuiting, transfers closures from the environment back to the stack. Without this rule, closures flow from the stack to the environment, but not back again (We are not concerned with the closures pointed to by the markers.). Closures in the environment may be shared. Therefore the stack must also hold pointers to closures to preserve this sharing.

For the proof of correctness we again develop a bisimulation, this time between the $\mathcal{L}$ machine and the $\mathcal{S}$ machine. The bisimulation is shown in Figure 8. The main difference between the two machines is that in the $\mathcal{L}$ machine there can appear chains of closures whose code part is a variable, whereas in the $\mathcal{S}$ machine these chains are collapsed and do not appear. The following is an example of such a chain.

$$\begin{aligned} & \langle x_1, \rho_1 \rangle \\ \mu_{\text{cl}}(\rho_1(x_1)) &= \langle x_2, \rho_2 \rangle \\ \mu_{\text{cl}}(\rho_2(x_2)) &= \langle x_3, \rho_3 \rangle \\ & \vdots \\ \mu_{\text{cl}}(\rho_n(x_n)) &= \langle N, \rho_{n+1} \rangle \qquad \text{where } N \notin \text{Var} \end{aligned}$$

Theorem 5. The relation $\sim_{\mathcal{S}}$ between states in an execution sequence on $\mathcal{L}$ and on $\mathcal{S}$ has the following property. If $s \sim_{\mathcal{S}} s'$, then exactly one of the following holds:

1. $s \longrightarrow_{\mathcal{L}} t$, $s' \longrightarrow_{\mathcal{S}} t'$, and $t \sim_{\mathcal{S}} t'$.
2. $s \longrightarrow_{\mathcal{L}}^{+} t$, $s' \longrightarrow_{\mathcal{S}} t'$, and $t \sim_{\mathcal{S}} t'$.
3. both s and s' are stuck.

Proof. For the most part, $\mathcal{L}$ and $\mathcal{S}$ run in lock step and it is easy to see they are in bisimilar states. However, there are two situations when $\mathcal{L}$ takes several steps for a single step of $\mathcal{S}$. The first situation is when evaluating a variable. In the $\mathcal{L}$ machine, the variable may be the start of a chain of closures whose code part is a variable. In this case, the $\mathcal{L}$ machine will take a series of VAR2 steps, ending in either a VAR2 or VAR1 step to catch up with the $\mathcal{S}$ machine. In performing the extra VAR2 steps, the $\mathcal{L}$ machine will have pushed markers on the stack, which is why the bisimulation includes a rule for shrinking the stack of $\mathcal{L}$. The second situation where the machines will be out of step is when the $\mathcal{L}$ machine is processing the markers pushed on the stack during the extra VAR2 steps. The $\mathcal{L}$ machine executes a series of UPDATE steps to catch up with the $\mathcal{S}$ machine.

4.1. PROPERTIES OF THE $\mathcal{S}$ MACHINE

The main property of the $\mathcal{S}$ machine is that it reduces the size of the environment. Environments appear in closures, and closures appear on

$$\frac{c \sim^c_{\mathcal{S}} c' \qquad \sigma \sim^\sigma_{\mathcal{S}} \sigma'}{(c, \sigma, \mu_{\mathrm{cl}}) \sim_{\mathcal{S}} (c', \sigma', \mu'_{\mathrm{cl}})}$$

$$\frac{x \neq M \qquad \mu_{\mathrm{cl}}(\rho(x)) \sim^c_{\mathcal{S}} \langle M, \rho' \rangle}{\langle x, \rho \rangle \sim^c_{\mathcal{S}} \langle M, \rho' \rangle} \qquad \frac{\rho \sim^\rho_{\mathcal{S}} \rho'}{\langle M, \rho \rangle \sim^c_{\mathcal{S}} \langle M, \rho' \rangle}$$

$$\frac{\mathrm{dom}(\rho) = \mathrm{dom}(\rho') \qquad \forall x \in \mathrm{dom}(\rho).\ \mu_{\mathrm{cl}}(\rho(x)) \sim^c_{\mathcal{S}} \mu'_{\mathrm{cl}}(\rho'(x))}{\rho \sim^\rho_{\mathcal{S}} \rho'}$$

$$\frac{}{() \sim^\sigma_{\mathcal{S}} ()} \qquad \frac{c \sim^c_{\mathcal{S}} \mu'_{\mathrm{cl}}(l) \qquad \sigma \sim^\sigma_{\mathcal{S}} \sigma'}{\mathsf{arg}(c) :: \sigma \sim^\sigma_{\mathcal{S}} \mathsf{arg}(l) :: \sigma'}$$

$$\frac{\mu_{\mathrm{cl}}(l') \sim^c_{\mathcal{S}} \mu'_{\mathrm{cl}}(l'') \qquad \mathsf{mark}(l) :: \sigma \sim^\sigma_{\mathcal{S}} \mathsf{mark}(l'') :: \sigma'}{\mathsf{mark}(l) :: \mathsf{mark}(l') :: \sigma \sim^\sigma_{\mathcal{S}} \mathsf{mark}(l'') :: \sigma'}$$

$$\frac{\mu_{\mathrm{cl}}(l) \sim^c_{\mathcal{S}} \mu'_{\mathrm{cl}}(l') \qquad \sigma \sim^\sigma_{\mathcal{S}} \sigma'}{\mathsf{mark}(l) :: \sigma \sim^\sigma_{\mathcal{S}} \mathsf{mark}(l') :: \sigma'}$$

Figure 8. Bisimilarity relation $\sim_{\mathcal{S}}$ between $\mathcal{L}$ and $\mathcal{S}$

the stack, so we have to take both into consideration when measuring total environment size.

$$\begin{aligned} \#(\rho) &= \Sigma_{x \in \mathrm{dom}(\rho)} \#(\rho(x)) \\ \#(\langle N, \rho \rangle) &= 1 + \#(\rho) \\ \#(\mathsf{arg}(c) :: \sigma) &= \#(c) + \#(\sigma) \\ \#(\mathsf{mark}(l) :: \sigma) &= \#(\mu_{\mathrm{cl}}(l)) + \#(\sigma) \\ \#(()) &= 0 \end{aligned}$$

Lemma 5. If $x \in \mathrm{dom}(\rho)$ and $\rho(x) = \langle N, \rho' \rangle$, then $\#(\rho') < \#(\rho)$.

Proof. The environment ρ' is contained within environment ρ, therefore we have $\#(\rho') < \#(\rho)$.

Theorem 6. Given execution sequences of a term N on $\mathcal{L}$ and on $\mathcal{S}$, if a closure of the form $\langle (Mx), \rho \rangle$ appears in the first position of a state s in the execution sequence on $\mathcal{L}$, and $\langle (Mx), \rho' \rangle$ appears in the first position of state s' in the execution sequence on $\mathcal{S}$, and if $s \sim_{\mathcal{S}} s'$, then for all subsequent states of the parallel execution sequence, we have $\#(\rho') + \#(\sigma') < \#(\rho) + \#(\sigma)$.

Proof. We focus on the situation when the first position of state s and s' is of the form $\langle (Mx), \rho \rangle$ and $\langle (Mx), \rho' \rangle$. We have the following transitions for $\mathcal{L}$ and $\mathcal{S}$.

$$\begin{aligned}(\langle (Mx), \rho \rangle, \sigma, \mu_{\mathrm{cl}}) &\longrightarrow_{\mathcal{L}} (\langle M, \rho \rangle, \mathsf{arg}(\langle x, \rho \rangle) :: \sigma, \mu_{\mathrm{cl}}) \\ (\langle (Mx), \rho' \rangle, \sigma', \mu'_{\mathrm{cl}}) &\longrightarrow_{\mathcal{S}} (\langle M, \rho' \rangle, \mathsf{arg}(\rho'(x)) :: \sigma', \mu'_{\mathrm{cl}}).\end{aligned}$$

Let $\rho'(x) = \langle N, \rho'' \rangle$. By Lemma 5, we know that $\#(\rho'') < \#(\rho')$. This means that whenever a term of the form (Mx) is encountered, $\mathcal{S}$ pushes a closure on the stack whose associated environment is smaller than the current environment. In $\mathcal{L}$, when a term (Mx) is encountered, the closure pushed on the stack contains the current environment. All the other rules are the same for $\mathcal{L}$ and $\mathcal{S}$. Therefore, for all subsequent states, we have $\#(\rho') + \#(\sigma') < \#(\rho) + \#(\sigma)$.

5. Combining Collapsed Markers and Short-circuiting

The $\mathcal{CS}$ machine (Figure 9) combines the improvements of $\mathcal{C}$ and $\mathcal{S}$. That is, redundant markers are not pushed on the stack and variables in operand position are eagerly dereferenced.

Each of the $\mathcal{C}$ and $\mathcal{S}$ optimizations independently improve the memory consumption of the machine. When combined, however, the improvement is significant: the optimizations work together to reduce the memory consumption associated with processing operands. Eager dereferencing reduces the size of the environment associated with an operand when it is pushed on the stack. Redundant marker elimination collapses the memory used for multiple operands into a single operand. The result is that for one of our benchmark programs, the combined machine stays afloat while the other machines sink.

From Section 3.2 we have the correctness of the $\mathcal{C}$ machine. One can view the $\mathcal{CS}$ machine as the $\mathcal{C}$ machine modified to include short-circuiting. Thus we can show correctness of the $\mathcal{CS}$ machine using the same technique with which we proved the $\mathcal{S}$ machine correct with respect to the $\mathcal{L}$ machine in Section 4. The bisimulation $\sim_{\mathcal{CS}}$ relating $\mathcal{C}$ and $\mathcal{CS}$, shown in Figure 10, is nearly identical to $\sim_{\mathcal{S}}$.

Theorem 7. The relation $\sim_{\mathcal{CS}}$ between states in an execution sequence on $\mathcal{C}$ and on $\mathcal{CS}$ has the following property. If $s \sim_{\mathcal{CS}} s'$, then exactly one of the following holds:

1. $s \longrightarrow_{\mathcal{C}} t$, $s' \longrightarrow_{\mathcal{CS}} t'$, and $t \sim_{\mathcal{CS}} t'$.
2. $s \longrightarrow_{\mathcal{C}}^{+} t$, $s' \longrightarrow_{\mathcal{CS}} t'$, and $t \sim_{\mathcal{CS}} t'$.
3. both s and s' are stuck.

$$\longrightarrow_{\mathcal{CS}} : \mathrm{State} \to \mathrm{State}$$
$$\mathrm{State} = \mathrm{Clos} \times \mathrm{Stack} \times \mathrm{Heap}$$
$$\rho \in \mathrm{Env} = \mathrm{Var} \to \mathrm{Loc}_{\mathrm{Loc}_{\mathrm{cl}}}$$
$$c \in \mathrm{Clos} ::= \langle (MN), \rho \rangle \mid \mathrm{Val} \ \subset \ \mathrm{Exp} \times \mathrm{Env}$$
$$\sigma \in \mathrm{Stack} = \mathsf{List}[\mathrm{Loc}_{\mathrm{cl}} + \mathrm{Loc}_{\mathrm{Loc}_{\mathrm{cl}}}]$$
$$r \in \mathrm{Loc}_{\mathrm{Loc}_{\mathrm{cl}}} \quad \mu_{\mathrm{loc}} \in \mathrm{Heap}_{\mathrm{Loc}_{\mathrm{cl}}} = \mathrm{Loc}_{\mathrm{Loc}_{\mathrm{cl}}} \to \mathrm{Loc}_{\mathrm{cl}}$$
$$l \in \mathrm{Loc}_{\mathrm{cl}} \quad \mu_{\mathrm{cl}} \in \mathrm{Heap}_{\mathsf{cl}} = \mathrm{Loc}_{\mathrm{cl}} \to \mathrm{Clos}$$
$$\mu \in \mathrm{Heap} = \mathrm{Heap}_{\mathsf{cl}} \times \mathrm{Heap}_{\mathrm{Loc}_{\mathrm{cl}}}$$

$$\frac{r = \rho(x) \qquad l = \mu_{\mathrm{loc}}(r) \qquad \mu_{\mathrm{cl}}(l) = v}{(\langle x, \rho \rangle, \sigma, \mu) \longrightarrow_{\mathcal{CS}} (v, \sigma, \mu)} \tag{Var1}$$

$$\frac{\begin{array}{c}\sigma = () \ \vee \ \sigma = \mathsf{arg}(-) :: \sigma' \\ r = \rho(x) \qquad l = \mu_{\mathrm{loc}}(r) \qquad \mu_{\mathrm{cl}}(l) = c \notin \mathrm{Val}\end{array}}{(\langle x, \rho \rangle, \sigma, \mu) \longrightarrow_{\mathcal{CS}} (c, \mathsf{mark}(l) :: \sigma, \mu)} \tag{Var2a}$$

$$\frac{\begin{array}{c}r = \rho(x) \qquad l' = \mu_{\mathrm{loc}}(r) \qquad \mu_{\mathrm{cl}}(l') = c \notin \mathrm{Val} \\ \mu'_{\mathrm{loc}} = \mu_{\mathrm{loc}}[r \mapsto l]\end{array}}{(\langle x, \rho \rangle, \mathsf{mark}(l) :: \sigma, \mu) \longrightarrow_{\mathcal{CS}} (c, \mathsf{mark}(l) :: \sigma, \langle \mu_{\mathrm{cl}}, \mu'_{\mathrm{loc}} \rangle)} \tag{Var2b}$$

$$\frac{\begin{array}{c}l \notin \mathrm{dom}(\mu_{\mathrm{cl}}) \qquad \mu'_{\mathrm{cl}} = \mu_{\mathrm{cl}}[l \mapsto \langle N, \rho \rangle] \\ r \notin \mathrm{dom}(\mu_{\mathrm{loc}}) \qquad \mu'_{\mathrm{loc}} = \mu_{\mathrm{loc}}[r \mapsto l]\end{array}}{(\langle (MN), \rho \rangle, \sigma, \mu) \longrightarrow_{\mathcal{CS}} (\langle M, \rho \rangle, \mathsf{arg}(r) :: \sigma, \langle \mu'_{\mathrm{cl}}, \mu'_{\mathrm{loc}} \rangle)} \tag{App}$$

$$\frac{r = \rho(x)}{(\langle (Mx), \rho \rangle, \sigma, \mu) \longrightarrow_{\mathcal{CS}} (\langle M, \rho \rangle, \mathsf{arg}(r) :: \sigma, \mu)} \tag{AppVar}$$

$$(\langle \lambda x.M, \rho \rangle, \mathsf{arg}(r) :: \sigma, \mu) \longrightarrow_{\mathcal{CS}} (\langle M, \rho[x \mapsto r] \rangle, \sigma, \mu) \tag{Call}$$

$$\frac{\mu'_{\mathrm{cl}} = \mu_{\mathrm{cl}}[l \mapsto v]}{(v, \mathsf{mark}(l) :: \sigma, \mu) \longrightarrow_{\mathcal{CS}} (v, \sigma, \langle \mu'_{\mathrm{cl}}, \mu_{\mathrm{loc}} \rangle)} \tag{Update}$$

Figure 9. Definition of the $\mathcal{CS}$ (collapsed markers and short-circuiting) machine

Proof. As in the proof of Theorem 5, we have $\mathcal{C}$ and $\mathcal{CS}$ running in lock step for the most part, except for two situations where $\mathcal{C}$ takes several steps for one step on $\mathcal{CS}$. The first situation is when evaluating a variable. In the $\mathcal{C}$ machine, the variable may be the start of a chain of closures whose code part is a variable. In this case, the $\mathcal{C}$ machine will take a series of Var2a or Var2b steps, ending in either a Var2a,

Notation: μ also denotes $\mu_{\text{cl}} \circ \mu_{\text{loc}}$

$$\frac{c \sim^{c}_{\mathcal{CS}} c' \qquad \sigma \sim^{\sigma}_{\mathcal{CS}} \sigma'}{(c, \sigma, \mu) \sim_{\mathcal{CS}} (c', \sigma', \mu')}$$

$$\frac{x \neq M \qquad \mu(\rho(x)) \sim^{c}_{\mathcal{CS}} \langle M, \rho' \rangle}{\langle x, \rho \rangle \sim^{c}_{\mathcal{CS}} \langle M, \rho' \rangle} \qquad \frac{\rho \sim^{\rho}_{\mathcal{CS}} \rho'}{\langle M, \rho \rangle \sim^{c}_{\mathcal{CS}} \langle M, \rho' \rangle}$$

$$\frac{\text{dom}(\rho) = \text{dom}(\rho') \qquad \forall x \in \text{dom}(\rho).\ \mu(\rho(x)) \sim^{c}_{\mathcal{CS}} \mu'(\rho'(x))}{\rho \sim^{\rho}_{\mathcal{CS}} \rho'}$$

$$\frac{}{() \sim^{\sigma}_{\mathcal{CS}} ()} \qquad \frac{c \sim^{c}_{\mathcal{CS}} \mu'(r) \qquad \sigma \sim^{\sigma}_{\mathcal{CS}} \sigma'}{\mathsf{arg}(c) :: \sigma \sim^{\sigma}_{\mathcal{CS}} \mathsf{arg}(r) :: \sigma'}$$

$$\frac{\mu_{\text{cl}}(l') \sim^{c}_{\mathcal{CS}} \mu'_{\text{cl}}(l'') \qquad \mathsf{mark}(l) :: \sigma \sim^{\sigma}_{\mathcal{CS}} \mathsf{mark}(l'') :: \sigma'}{\mathsf{mark}(l) :: \mathsf{mark}(l') :: \sigma \sim^{\sigma}_{\mathcal{CS}} \mathsf{mark}(l'') :: \sigma'}$$

$$\frac{\mu_{\text{cl}}(l) \sim^{c}_{\mathcal{CS}} \mu'_{\text{cl}}(l') \qquad \sigma \sim^{\sigma}_{\mathcal{CS}} \sigma'}{\mathsf{mark}(l) :: \sigma \sim^{\sigma}_{\mathcal{CS}} \mathsf{mark}(l') :: \sigma'}$$

Figure 10. Bisimilarity relation $\sim_{\mathcal{CS}}$ between $\mathcal{C}$ and $\mathcal{CS}$

VAR2B, or VAR1 step to catch up with the $\mathcal{CS}$ machine. The second situation where the machines will be out of step is when the $\mathcal{C}$ machine is processing the markers pushed on the stack during the extra VAR2A steps. The $\mathcal{C}$ machine executes a series of UPDATE steps to catch up with the $\mathcal{CS}$ machine.

6. Experiments

The four machines were implemented in Scheme. The experiments were compiled and run using Chez Scheme version 6.9 with optimization level 3 on a Sun Blade 100. The wall clock time was obtained using the Chez Scheme `time` function, and for each entry, the time presented is the best time of 6 trials. We report on a number of statistics to give a more detailed picture of the performance, including the number of pushes and pops from the stack and reads and writes to the heap. Also, "Maximum Live Data" field measures the number of objects reachable

from the environment and the stack, including closures, environments, and heap cells. The unit of measurement is a Scheme memory cell, with stacks and environments represented as lists and associative lists.

We use four benchmark programs to evaluate the performance of our four machines: $\mathcal{L}$, $\mathcal{C}$, $\mathcal{S}$, and $\mathcal{CS}$. The first is `(=0 (- (factorial 5) 120))`. The `5` and `120` are represented as Barendregt numerals [3] and `=0` is an appropriate test for zero. All nontrivial functions are written in the style of primitive recursive functions [12] where recursion is supported using the Y combinator. The subtraction and test for zero force `(factorial 5)` to run to completion. Each of the machines returns the term $\lambda x.\lambda y.x$, which is the Church representation for true.

The call-by-name $\mathcal{K}$ machine executes this benchmark in 658,608,487 steps. The performance of the four machines is summarized in Table I. As expected the depth of the stack (including both closure operands and markers) and the number of updates is reduced by the collapsed marker optimization. Somewhat unexpected is the reduction in stack depth and number of updates due to short-circuiting (Its main purpose is to shrink the environment.). Normal evaluation of operand variable references includes placing markers on the stack, whereas short-circuiting does not. Eager variable dereferencing also causes a few unnecessary lookups; the environment references increased by 10%. With the combined optimizations, we have a 27% decrease in execution time, 27% fewer pushes (pops) on the stack, and 55% fewer writes to the heap. On the negative side, there are 35% more reads from the heap, and the $\mathcal{CS}$ performs more dispatching during execution, which partly explains why there is not a greater speedup. There was a 7% decrease in total memory consumption, with some of the space saved on the stack offset by the additional heap cells needed for the indirection.

The second benchmark is `(=0 (- (tak 12 10 6) 7))`, where `tak` is the Takeuchi function implemented in terms of the lambda calculus. Table II shows the results for the rest of the machines. Here the $\mathcal{C}$ machine had the advantage in execution time and $\mathcal{S}$ machine used the least memory. Also, the $\mathcal{CS}$ machine has a slower execution time than $\mathcal{L}$ machine, but saves a little space.

The third benchmark evaluates `(=0 (- (stream-ref primes 8) 23))`, where `primes` is a lazy list of prime numbers generated by the sieve of Eratosthenes [1]. The list is represented via pairing [3] and the remainder is calculated via iterated subtraction. As usual the numbers are represented as Barendregt numerals. In this experiment, again the $\mathcal{C}$ machine had the best execution time and $\mathcal{CS}$ machine used the least memory.

The fourth benchmark is the program described in Section 3, which exhibits unbounded stack growth due to marker sequences. The pro-

Table I. Factorial Benchmark

	$\mathcal{L}$	$\mathcal{C}$	$\mathcal{S}$	$\mathcal{CS}$
Wall Clock (milliseconds)	380	320	270	270
Steps of Computation	18,012	16,394	14,790	14,204
Marker Updates	2,911	1,293	1,300	714
Maximum Stack Depth	485	243	364	243
Stack Pushes/Pops	7,966	6,348	6,355	5,769
Reads from μ_{cl}	4,991	4,991	3,380	3,380
Writes to μ_{cl}	2,911	1,293	1,300	714
Reads from μ_{loc}	-	4,991	-	3,380
Writes to μ_{loc}	-	1,618	-	586
Environment References	4,991	4,991	5,584	5,584
Maximum Live Data	3,226	3,000	2,928	3,006

Table II. Takeuchi Benchmark

	$\mathcal{L}$	$\mathcal{C}$	$\mathcal{S}$	$\mathcal{CS}$
Wall Clock (milliseconds)	12,300	10,980	15,920	14,430
Steps of Computation	277,970	257,600	231,206	225,220
Marker Updates	39,200	18,830	15,818	9,832
Maximum Stack Depth	147	90	115	89
Stack Pushes/Pops	119,425	96,043	99,055	90,057
Reads from μ_{cl}	78,320	78,320	54,938	54,938
Writes to μ_{cl}	39,200	18,830	15,818	9,832
Reads from μ_{loc}	-	78,320	-	54,938
Writes to μ_{loc}	-	20,370	-	5,986
Environment References	78,320	78,320	85,103	85,103
Maximum Live Data	779	811	678	707

gram is $((Y(\lambda z.(\lambda y.((\lambda x.x)(zy)))))(\lambda x.x))$. Tables IV and V present the results for running this program for $1,000$ and $2,000$ steps. The surprising result is that the maximum depth of the stack for $\mathcal{C}$ and $\mathcal{CS}$ is a small constant, whereas the depth of the stack for $\mathcal{L}$ and $\mathcal{S}$ grows at a rate of 7 items per 100 steps. Further, the total memory usage is constant for $\mathcal{CS}$, whereas it is unbounded for the other machines.

Implementations of the machines and the benchmark programs as presented in this section are available at

`http://www.osl.iu.edu/~jsiek/fried-hosc-krivine.tar.gz.`

Table III. Sieve Benchmark

	$\mathcal{L}$	$\mathcal{C}$	$\mathcal{S}$	$\mathcal{CS}$
Wall Clock (milliseconds)	5,220	4,600	5,800	5,540
Steps of Computation	142,735	131,857	118,113	115,166
Marker Updates	20,541	9,663	8,230	5,283
Maximum Stack Depth	205	111	157	111
Stack Pushes/Pops	61,646	50,768	49,335	46,388
Reads from μ_{cl}	39,984	39,984	27,673	27,673
Writes to μ_{cl}	20,541	9,663	8,230	5,283
Reads from μ_{loc}	-	39,984	-	27,673
Writes to μ_{loc}	-	10,878	-	2,947
Environment References	39,984	39,984	44,279	44,279
Maximum Live Data	1,429	1,234	1,258	1,190

Table IV. Constant Stack Depth Versus Unbounded Growth

	$\mathcal{L}$	$\mathcal{C}$	$\mathcal{S}$	$\mathcal{CS}$
Steps of Computation	1,000	1,000	1,000	1,000
Marker Updates	131	131	76	76
Maximum Stack Depth	70	5	79	4
Stack Pushes/Pops	467	401	462	386
Reads from μ_{cl}	331	331	306	306
Writes to μ_{cl}	131	131	76	76
Reads from μ_{loc}	-	331	-	306
Writes to μ_{loc}	-	66	-	76
Environment References	331	331	458	458
Maximum Live Data	504	394	229	32

7. Related Work

Crégut [4] gives a succinct introduction to the Krivine machine [8]. He then introduces two machines based on $\mathcal{K}$ for reducing lambda terms to normal form. Sestoft [15] derives a call-by-need variant of $\mathcal{K}$ from the natural semantics of Launchbury [9] and then describes some optimizations, none of which relate to marker sequences. Launchbury et al. [10] use a compile-time analysis to identify when the result of an operand does not need to be shared, and thus no marker is required. The TIM [6] is a simple, lazy abstract machine based on compiling to supercombinators. Fairbairn and Wray observe the problem of marker

Table V. Constant Stack Depth Versus Unbounded Growth

	$\mathcal{L}$	$\mathcal{C}$	$\mathcal{S}$	$\mathcal{CS}$
Steps of Computation	2,000	2,000	2,000	2,000
Marker Updates	264	264	153	153
Maximum Stack Depth	137	5	156	4
Stack Pushes/Pops	934	802	923	771
Reads from μ_{cl}	665	665	613	613
Writes to μ_{cl}	264	264	153	153
Reads from μ_{loc}	-	665	-	613
Writes to μ_{loc}	-	132	-	152
Environment References	665	665	919	919
Maximum Live Data	1,016	765	460	32

sequences and briefly mention an improvement, but their description lacks enough information for us to compare their improvement to ours. Guy Argo [2] makes improvements to the TIM, including mechanisms for avoiding updates to the heap while consuming markers. Wakeling and Dix improve the TIM by reducing environment allocation by sharing frames [17]. Peyton Jones [7] discusses the redundant marker problem in some detail. In his machine, the redundant markers are eliminated by the garbage collector, but only after they appear. The TIM implementation also does this. In Section 3 we have presented a term that produces an unbounded number of markers on the stack. Thus the garbage collector would have to run continually to keep up with the program, thereby slowing overall execution. We suggest that it is better not to push the redundant markers in the first place.

Randell and Russell [13] made the observation that variables should be dereferenced before putting them on the argument stack, and Crégut [5] presents a version of the Krivine machine with this optimization. This eager evaluation of references is a special case of the selective eager evaluation enabled by strictness analysis [11]. In general, space leaks in the presence of tail recursion are a common problem in non-strict languages [14]. Sestoft observes that the natural implementation of a lazy $\mathcal{K}$ machine accumulates excessive environment space. This occurs because the interesting notion of reference for a closure in an environment is a variable reference in a closure containing that environment, which the implementation language's garbage collector does not know. Sestoft's solution, environment trimming, is to copy only the variables referenced in a closure's code into the closure's environment. The TIM

copies arguments but annotates the frame of the closure with a bit pattern denoting which closures can be collected.

8. Conclusion

We have presented three machines, the last of these machines is the combination of the other two. Each of the other two are independent, but both build directly from Sestoft's call-by-need Krivine machine, restricted to closed lambda terms. The first of these machines shows how to avoid producing sequences of markers. This has the property that if the top of the stack contains a marker, the location just below it does not contain a marker. This algorithm is inspired by thinking about the markers as sequences of returns in van Wijngaarden's device. (This device, of course, is one of the first, if not the first, informal characterization of continuation-passing style.) More importantly, since all the *returns* are the same, we introduce sharing to take advantage of this fact. Thus, we only use a single marker, even though there may appear to be many returns.

In the second machine we have observed that eager dereferencing of variables in operand position leads to smaller environments and fewer updates. Since each variable is bound to a closure, we know that we have not adversely affected the correctness of the program; we have only improved on the number of steps the computation runs and on its memory consumption. We have demonstrated a benchmark that consumes an unbounded amount of memory on all except the $\mathcal{CS}$ machine. Thus not only are the two machines independent improvements on $\mathcal{L}$, but they are also synergistic.

The original $\mathcal{K}$ machine processes lambda terms. In keeping with the spirit of this special issue, we have chosen to stay in that framework. That is, we do not include basic primitives and constants and do no static analysis or preprocessing into some variation on the lambda terms. The simplicity of the lambda terms facilitates a straightforward proof of correctness of our $\mathcal{CS}$ machine. Also, by restricting ourselves to lambda terms, attention is focused solely on the issues surrounding call-by-need, making possible the observation of several significant properties of our implementation. If we had weaved our enhancements through some more realistic (and more complex) machine, we would have perhaps produced more convincing evidence of our improvements, but also, there might be questions as to the accuracy of our experiments. We believe that by reducing the discussion to simple lambda terms, our results stand on their own. We look forward to future re-

search that combines these ideas with current implementations of lazy functional languages.

Acknowledgments

We thank Ron Garcia for helpful discussions and Olivier Danvy and Mitchell Wand for advice and encouragement. We also thank the reviewers for their time and comments.

References

1. H. Abelson and G. J. Sussman. *The Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, 1985.
2. G. Argo. Improving the Three Instruction Machine. In *Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 100–112, Reading, MA, September 1989. Addison-Wesley.
3. H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, Amsterdam, 1981.
4. 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.
5. P. Crégut. *Machines à environment pour la réduction symbolique et l'évaluation partielle*. PhD thesis, Université de Paris VII, Paris, France, 1991.
6. J. Fairbairn and S. C. Wray. Tim: A simple lazy abstract-machine to execute supercombinators. In G. Kahn, editor, *IFIP Functional Programming Languages and Computer Architecture*, number 274 in Lecture Notes in Computer Science, pages 34–45. Springer Verlag, 1987.
7. S. L. P. Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.
8. J.-L. Krivine. Un interprète du λ-calcul. 1985.
9. J. Launchbury. A natural semantics for lazy evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 144–154, Charleston, South Carolina, 1993.
10. J. Launchbury, A. Gill, J. Hughes, S. Marlow, S. Peyton Jones, and P. Wadler. Avoiding unnecessary updates. In J. Launchbury and P. M. Sansom, editors, *Functional Programming, Glasgow 1992*. Springer-Verlag, 1992.
11. A. Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs*. PhD thesis, Univerity of Edinburgh, Edinburgh, Scotland, 1981.
12. R. Péter. *Recursive Functions*. Academic Press, 1967.
13. B. Randell and L. J. Russell. *Algol 60 Implementation*. Academic Press, New York, 1964.
14. C. Reinke. Ghood – graphical visualisation and animation of haskell object observations. In R. Hinze, editor, *Haskell Workshop*, 2001.
15. P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, 1997.

16. T. B. Steel, editor. *Formal Language Description Languages for Computer Programming*. North-Holland, Amsterdam, 1966.
17. D. Wakeling and A. Dix. Optimising partial applications in TIM. Technical report, University of York, November 1993.
18. M. Wand. On the correctness of the Krivine Machine. submitted for publication, 2003.