SAND2017-8450C

Cactus Environment Machine

Shared Environment Call-by-Need

George Stelle[1,2], Darko Stefanovic[1], Stephen L. Olivier[2], and Stephanie Forrest[1]

[1] University of New Mexico
{stelleg, darko, forrest}@cs.unm.edu
[2] Sandia National Laboratories
slolivi@sandia.gov

Abstract. Existing machines for lazy evaluation use a *flat* representation of environments, storing the terms associated with free variables in an array. Combined with a heap, this structure supports the shared intermediate results required by lazy evaluation. We propose and describe an alternative approach that uses a *shared* environment to minimize the overhead of delayed computations. We show how a shared environment can act as both an environment and a mechanism for sharing results. To formalize this approach, we introduce a calculus that makes the shared environment explicit, as well as a machine to implement the calculus, the *Cactus Environment Machine*. A simple compiler implements the machine and is used to run experiments for assessing performance. The results show reasonable performance and suggest that incorporating this approach into real-world compilers could yield performance benefits in some scenarios.

1 Introduction

Call-by-need evaluation is a formalization of the idea that work should be delayed until needed, and performed only once. Existing implementations of call-by-need take care in *packaging* a delayed computation, or *thunk*, by building a closure with an array that contains the bindings of all free variables [24, 7]. The overhead induced by this operation is well known, and is one reason existing implementations avoid thunks wherever possible [18]. The key insight of our Cactus Environment ($\mathscr{CE}$) Machine is that this overhead can be minimized by only recording a location in a shared environment.

As an example, consider the application $f\ e$. In existing call-by-need implementations, e.g., the STG machine[24], a closure with a flat environment will be constructed for e. Doing so incurs a time and memory cost proportional to the number of free variables of e. [1] We minimize this packaging cost by recording a location in a shared environment, which requires only two machine words (and two instructions) for the thunk: one for the code pointer, and one for the environment pointer. One way to think about the approach is that it is *lazier* about lazy evaluation: in the case that e is unneeded, the work to package it in a thunk is entirely wasted. In the spirit of lazy evaluation, we attempt to minimize this potentially unnecessary work.

The main contributions of the paper are:

[1] In some implementations, these are lambda-lifted to be formal parameters, but the principle is the same.

- A big-step calculus and small-step abstract machine that formalize the notion of a shared environment for call-by-need evaluation using an explicitly shared environment (Section 4).
- A simple implementation of the abstract machine that compiles to x86 assembly with a preliminary evaluation that shows performance comparable to existing implementations (Sections 6 and 7).

Section 2 reviews relevant background material, and Section 3 discusses the current landscape of environment representations, highlighting the opportunity for combining shared environments with lazy evaluation. We then provide some intuition for why this might be combination might be desirable, and formalize the connection between call-by-need evaluation and shared environments in a calculus (Section 4). Section 5 uses the calculus to derive a novel abstract machine, the $\mathscr{CE}$ machine, explains how $\mathscr{CE}$ uses the shared environment in a natural way to implement lazy evaluation, and gives its formal semantics. We then describe a straightforward implementation of $\mathscr{CE}$ in Section 6, extended with machine literals and primitive operations, and compiling directly to native code. We evaluate the implementation in Section 7, showing that it is capable of performing comparably to existing implementations despite lacking several common optimizations, and we discuss the results. We discuss related work, the limitations of our approach, and some ideas for future work in Section 8, and conclude the paper in Section 9.

2 Background and Motivation

This section provides relevant background for the $\mathscr{CE}$ machine, outlining lambda calculus, evaluation strategies, and Curien's calculus of closures.

2.1 Preliminaries

We begin with the simple lambda calculus [5]:

$$t ::= x \mid \lambda x.t \mid t\ t$$

where x is a variable, $\lambda x.t$ is an abstraction, and $t\ t$ is an application. We also use lambda calculus with deBruijn indices, which replaces variables with a natural number indexing into the binding lambdas. This calculus is given by the syntax:

$$t ::= i \mid \lambda t \mid t\ t$$

where $i \in \mathbb{N}$. In both cases, we use the standard Barendregt syntax conventions, namely that applications are left associative and the bodies of abstractions extend as far as possible to the right [5]. A *value* in lambda calculus refers to an abstraction. We are concerned only with evaluation to weak head normal form (WHNF), which terminates on an abstraction without entering its body.

In mechanical evaluation of expressions, it would be too inefficient to perform explicit substitution. To solve this, the standard approach uses closures [20, 8, 24, 6]. Closures combine a term with an environment, which binds the free variables of the term to closures.

As the formal basis for closures we use Curien's calculus of closures [8], Figure 1. It is a formalization of closures with an environment represented as a list of closures, indexed by deBruijn indices. We will occasionally modify this calculus by replacing the deBruijn indices with variables for readability, in which case variables are looked up in the environment instead of indexed, e.g., $t[x = c, y = c']$) [5]. We also add superscript and subscript markers to denote unique syntax elements, e.g., $t', t_1 \in \text{Term}$.

Syntax

$$t ::= i \mid \lambda t \mid t\ t \qquad \text{(Term)}$$
$$i \in \mathbb{N} \qquad \text{(Variable)}$$
$$c ::= t[\rho] \qquad \text{(Closure)}$$
$$\rho ::= \bullet \mid c \cdot \rho \qquad \text{(Environment)}$$

Semantics

$$\frac{t_1[\rho] \xrightarrow{*}_L \lambda t_2[\rho']}{t_1 t_3[\rho] \rightarrow_L t_2[t_3[\rho] \cdot \rho']} \qquad \text{(LEval)}$$

$$i[c_0 \cdot c_1 \cdot ... c_i \cdot \rho] \rightarrow_L c_i \qquad \text{(LVar)}$$

Fig. 1. Curien's call-by-name calculus of closures [8]

2.2 Evaluation Strategies

There are three standard evaluation strategies for lambda calculus: call-by-value, call-by-need, and call-by-name. Call-by-value evaluates every argument to a value, whereas call-by-need and call-by-name only evaluate an argument if it is needed. If an argument is needed more than once, call-by-name re-computes the value, whereas call-by-need memoizes the value, so it is computed at most once. Thus, call-by-need attempts to embody the best of both worlds—never repeat work (call-by-value), and never perform unnecessary work (call-by-name). These are intuitively good properties to have, and we illustrate the correctness of such an intuition with the following example, modified from [10]:

$$\overbrace{c_m(c_m(\cdots(c_m}^{m}\ id\ \overbrace{id)\cdots)id)}^{m}\ true\ id\ bottom$$

where $c_n = \lambda s.\lambda z.\overbrace{s\ (s \cdots (s}^{n}\ z)\cdots)$, $true = \lambda t.\lambda f.t$, $id = \lambda x.x$, and $bottom = (\lambda x.x\,x)\lambda x.x\,x$. Call-by-value never terminates, call-by-name takes exponential time, and call-by-need takes only polynomial time [10]. Of course, this is a contrived example, but it illustrates desirable properties of call-by-need.

In practice, however, there are significant issues with call-by-need evaluation. We focus on the following: *Delaying a computation is slower than performing it immediately.* This issue is well known [18, 24], and has become part of the motivation for *strictness analysis* [23, 32], which transforms non-strict evaluation to strict when possible.

2.3 Existing Call-by-Need Machines

Diehl et al. [11] review the call-by-need literature in detail. Here we summarize the most relevant points.

The best known machine for lazy evaluation is the Spineless Tagless G-Machine (STG machine), which underlies the Glasgow Haskell Compiler (GHC). STG uses flat environments that can be allocated on the stack, the heap, or some combination [24].

Two other influential lazy evaluation machines relevant to the $\mathscr{CE}$ machine are the call-by-need Krivine machines [14, 19, 28], and the three-instruction machine (TIM) [13]. Krivine machines started as an approach to call-by-name evaluation, and were later extended to call-by-need [19, 28, 10, 14]. The $\mathscr{CE}$ modifies the lazy Krivine machine to capture the environment sharing given by the cactus environment. The TIM is an implementation of call-by-need and call-by-name [13]. It involves, as the name suggests, three machine instructions, `TAKE`, `PUSH`, and `ENTER`. In Section 6, we follow Sestoft [28] and re-appropriate these instructions for the $\mathscr{CE}$ machine.

There has also been recent interest in *heapless* abstract machines for lazy evaluation. Danvy et al. [9] and Garcia et al. [27] independently derived similar machines from the call-by-need lambda calculus [4]. These are interesting approaches, but it is not yet clear how these machines could be implemented efficiently.

3 Environment Representations

As mentioned in Section 2, environments bind free variables to closures. There is significant flexibility in how they can be represented. In this section we review this design space in the context of existing work, both for call by value and call by need.[2]

There are two common approaches to environment representation: *flat* environments and *shared* environments (also known as linked environments) [2, 29]. A flat environment is one in which each closure has its own record of the terms its free variables are bound to. A shared environment is one in which parts of that record can be shared among multiple closures [2, 29]. For example, consider the following term:

$$(\lambda x.(\lambda y.t)(\lambda z.t_1))t_2$$

Assuming the term t has both x and y as free variables, we must evaluate it in the environment binding both x and y. Similarly, assuming t_1 contains both z and x as free

[2] Some work refers to this space as *closure* representation rather than *environment* representation [29, 2]. Because the term part of the closure is simply a code pointer and the interesting design choices are in the environment, we refer to the topic as environment representation.

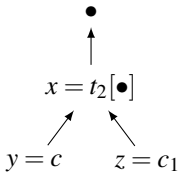
variables, we must evaluate it in an environment containing bindings for both x and z. Thus, we can represent the closures for evaluating t and t_1 as

$$t[x = t_2[\bullet], y = c]$$

and

$$t_1[x = t_2[\bullet], z = c_1]$$

respectively, where $\bullet$ is the empty environment. These are examples of *flat* environments, where each closure comes with its own record of all of its free variables. Because of the nested scope of the given term, x is bound to the same closure in the two environments. Thus, we can also create a shared, linked environment, represented by the following diagram:



Now each of the environments is represented by a linked list, with the binding of x shared between them. This is an example of a *shared* environment [2]. This shared, linked structure dates back to the first machine for evaluating expressions, Landin's SECD machine [20].

The drawbacks and advantages of each approach are well known. With a flat environment, variable lookup can be performed with a simple offset [24, 1]. On the other hand, significant duplication can occur, as we will discuss in Section 3.1. With a shared environment, that duplication is removed, but at the cost of possible link traversal upon dereference.

As with most topics in compilers and abstract machines, the design space is actually more complex. For example, Appel and Jim show a wide range of hybrids [2] between the two, and Appel and Shao [29] show an optimized hybrid that aims to achieve the benefits of both approaches. And as shown in the next section, choice of evaluation strategy further complicates the picture.

3.1 Existing Call-by-Need Environments

Existing call by need machines use flat environments with a heap of closures [24, 13, 18, 7]. These environments may contain some combination of primitive values and pointers into the heap (p below). The pointers and heap implement the memoization of results required for call by need. Returning to the earlier example, $(\lambda x.(\lambda y.t)(\lambda z.t_1))t_2$, we can view a simplified execution state for this approach when entering t as follows:

Closure

$$t[x = p_0, y = p_1]$$

Heap

$$p_0 \mapsto t_2[\bullet]$$
$$p_1 \mapsto \lambda z.t_1[x = p_0]$$

Consider $t_2[\bullet]$, the closure at p_0. If it is not in WHNF (this sort of unevaluated closure is called a *thunk* [17, 25]), then if it is entered in either the evaluation of t or t_1, the resulting value will overwrite the closure at p_0. The result of the computation is then shared with all other instances of x in t and t_1. In the case that terms have a large number of shared variables, environment duplication can be expensive. Compile-time transformation [25] (tupling arguments) helps, but we show that the machine can avoid duplication completely.

Depending on t, either or both of the closures created for its free variables may not be evaluated. Therefore, it is possible that the work of creating the environment for that thunk will be wasted. This waste is well known, and existing approaches address it by avoiding thunks as much as possible [24, 18]. Unfortunately, in cases like the above example, thunks are necessary. We aim to minimize the cost of creating such thunks.

Thunks are special in another way. Recall that one advantage of flat environments is quick variable lookups. In a lazy language, this advantage is reduced because *a thunk can only be entered once*. After it is entered, it is overwritten with a value, so the next time that heap location is entered it is entered with a value and a different environment. Thus, the work to ensure that the variable lookup is fast is used only once. This is in contrast to a call by value language, in which every closure is constructed for a value, and can be entered an arbitrary number of times.

A more subtle drawback of the flat environment representation is that environments can vary in size, and thus a value in WHNF can be too large to fit in the space allocated for the thunk it is replacing. This problem is discussed in [24], where the proposed solution is to put the value closure in a fresh location in the heap where there is sufficient room. The original thunk location is then replaced with an indirection to the value at the freshly allocated location. These indirections are removed during garbage collection, but do impose some cost, both in runtime efficiency and implementation complexity [24].

We have thus far ignored a number of details with regard to current implementations. For example, the STG machine can split the flat environment, so that part is allocated on the stack and part on the heap. The TIM allocates its flat environments separately from its closures so that each closure is a code pointer, environment pointer pair [13] while the STG machine keeps environment and code co-located [24]. Still, the basic design principle holds: a flat environment for each closure allows quick variable indexing, but with an initial overhead.

To summarize, the flat environment representation in a call by need language implies that whenever a term might be needed, the necessary environment is constructed from the current environment. This operation can be expensive, and it is wasted if the variable is never entered. In this work, we aim to minimize this potentially unnecessary overhead.

Figure 2 depicts the design space relevant to this paper. There are existing call by value machines with both flat and shared environments, and call by need machines with

flat environments. As far as we are aware, we are the first to use a shared environment to implement lazy evaluation.

It is worth noting that there has been work on lazy machines that effectively use linked environments, which could potentially be implemented as a shared environment, e.g., Sestoft's work on Krivine machines [28], but none make the realization that the shared environment can be used to implement sharing of results, which is the primary contribution of this paper.

	Flat Environment	Shared Environment
Call by need	STG [24], TIM [13], GRIN [7]	$\mathscr{CE}$ Machine (this paper)
Call by value	ZAM [21], SML/NJ [3]	ZAM, SECD [20], SML/NJ

Fig. 2. Evaluation strategy and environment structure design space. Each acronym refers to an existing implementation. Some implementations use multiple environment representations.

4 Cactus Environment Calculus

This section shows how the shared environment approach can be applied to call-by-need evaluation. We start with a calculus that abstracts away environment representation, Curien's calculus of closures, and we show how it can be modified to force sharing. See Curien's call-by-name calculus of closures in Figure 1. [3]

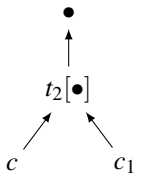
The LEval rule pushes a closure onto the environment, and the LVar rule indexes into the environment, entering the corresponding closure. We show in this section that by removing ambiguity about how the environments are represented, and forcing them to be represented in a *cactus stack* [30], we can define our novel call-by-need calculus.

To start, consider again the example from Section 3, this time with deBruijn indices: $(\lambda(\lambda t)\ (\lambda t_1))t_2$. The terms t and t_1, when evaluated in the closure calculus, would have the following environments, respectively:

$$c \cdot t_2[\bullet] \cdot \bullet$$
$$c_1 \cdot t_2[\bullet] \cdot \bullet$$

Again, the second closure is identical in each environment. And again, we can represent these environments with a shared environment, this time keeping call-by-need evaluation in mind:

[3] Curien calls it a "lazy" evaluator, and there is some ambiguity with the term lazy, but we use the term only to mean call-by-need. We also remove the condition checking that $i < m$ because we are only concerned with evaluation of closed terms.



This inverted tree structure seen earlier with the leaves pointing toward the root is called a *cactus stack* (sometimes called a spaghetti stack or saguaro stack) [15, 16]. In this particular cactus stack, every node defines an environment as the sequence of closures in the path to the root. If $t_2[\bullet]$ is a thunk, and is updated in place with the value after its first reference, then both environments would contain the resulting value. This is exactly the kind of sharing that is required by call-by-need, and thus we can use this structure to build a call-by-need evaluator. This is the essence of the cactus environment calculus and the cactus environment ($\mathscr{CE}$) machine.

Curien's calculus of closures does not differentiate between flat and shared environment representations, and indeed, no calculus that we are aware of has had the need to. Therefore, we must derive a calculus of closures, forcing the environment to be shared. Because we can hold the closure directly in the environment, we choose to replace the standard heap of closures with a *heap of environments*. To enforce sharing, we extend Curien's calculus of closures to explicity include the heap of environments, which we refer to as a *cactus environment*.

See Figure 3 for the syntax and semantics of the cactus calculus. Recall that we are only concerned with evaluation of closed terms. The initial closed term t is placed in a $(t[0], \varepsilon[0 \mapsto \bullet])$ tuple, and evaluation terminates on a value. We use some shorthand to make heap notation more palatable, for both the big-step semantics presented here and the small step semantics presented in the next section. $\mu(l,i) = l' \mapsto c \cdot l''$ denotes that looking up the i'th element in the linked environment structure starting at l results in location l', where closure c and continuing environment l'' reside. $\mu(l) = c \cdot l'$ is the statement that $l \mapsto c \cdot l' \in \mu$, and $\mu(u \mapsto c \cdot l')$ is μ with location u updated to map to $c \cdot e$. We define two different semantics, one for call-by-name and one for call-by-need, which makes the connection to Curien's call-by-name calculus more straightfoward. The rule for application (MEval and NEval) is identical for both semantics: each evaluates the left hand side to a function, then binds the variable in the cactus environment, extending the current environment.

The only difference between this semantics and Curien's is that if we need to extend an environment multiple times, the semantics *requires* sharing it among the extensions. This makes no real difference for call-by-name, but it is needed for the sharing of results in the NVar rule. The explicit environment sharing ensures that the closure that is overwritten with a value is shared correctly.

4.1 Correctness

Ariola et al. define the standard call-by-need semantics in [4]. To show correctness, we show that there is a strong bisimulation between $\rightarrow_N$ and their operational semantics, $\Downarrow$ (Figure 4).

Syntax

$$t ::= i \mid \lambda t \mid t\ t \qquad \text{(Term)}$$
$$i \in \mathbb{N} \qquad \text{(Variable)}$$
$$c ::= t[l] \qquad \text{(Closure)}$$
$$v ::= \lambda t[l] \qquad \text{(Value)}$$
$$\mu ::= \varepsilon \mid \mu[l \mapsto \rho] \qquad \text{(Heap)}$$
$$\rho ::= \bullet \mid c \cdot l \qquad \text{(Environment)}$$
$$l, f \in \mathbb{N} \qquad \text{(Location)}$$
$$s ::= (c, \mu) \qquad \text{(State)}$$

Call-by-Name Semantics

$$\frac{(t[l],\mu) \xrightarrow{*}_M (\lambda t_2[l'],\mu') \quad f \notin \mathrm{dom}(\mu')}{(t\ t_3[l],\mu) \to_M (t_2[f],\mu'[f \mapsto t_3[l] \cdot l'])} \qquad \text{(MEval)}$$

$$\frac{\mu(l,i) = l' \mapsto c \cdot l''}{(i[l],\mu) \to_M (c,\mu)} \qquad \text{(MVar)}$$

Call-by-Need Semantics

$$\frac{(t[l],\mu) \xrightarrow{*}_N (\lambda t_2[l'],\mu') \quad f \notin \mathrm{dom}(\mu')}{(t\ t_3[l],\mu) \to_N (t_2[f],\mu'[f \mapsto t_3[l] \cdot l'])} \qquad \text{(NEval)}$$

$$\frac{\mu(l,i) = l' \mapsto c \cdot l'' \quad (c,\mu) \xrightarrow{*}_N (v,\mu')}{(i[l],\mu) \to_N (v,\mu'(l' \mapsto v \cdot l''))} \qquad \text{(NVar)}$$

Fig. 3. Cactus calculus syntax and semantics.

$$\Phi, \Psi, \Upsilon ::= x_1 \mapsto t_1, \ldots, x_n \mapsto t_n \qquad \text{(Heap)}$$

$$\frac{\langle\Phi\rangle t \Downarrow \langle\Psi\rangle \lambda x.t'}{\langle\Phi, x \mapsto t, \Upsilon\rangle x \Downarrow \langle\Psi, x \mapsto \lambda x.t', \Upsilon\rangle \lambda x.t'} \qquad \text{(Id)}$$

$$\frac{}{\langle\Phi\rangle \lambda x.t \Downarrow \langle\Phi\rangle \lambda x.t} \qquad \text{(Abs)}$$

$$\frac{\langle\Phi\rangle t_l \Downarrow \langle\Psi\rangle \lambda x.t_n \quad \langle\Psi, x' \mapsto t_m\rangle [x'/x] t_n \Downarrow \langle\Upsilon\rangle \lambda y.t'}{\langle\Phi\rangle t_l\ t_m \Downarrow \langle\Upsilon\rangle \lambda y.t'} \qquad \text{(App)}$$

Fig. 4. Ariola et. al's Operational Semantics

Theorem 1. (Strong Bisimulation)

$$\rightarrow_N \sim \Downarrow$$

We start with a *flattening* relation between a configuration for $\Downarrow$ and a configuration for $\rightarrow_N$. The deBruijn indexed terms and the standard terms are both converted to terms that use deBruijn indices for local variables and direct heap locations for free variables. The flattening relation holds only when both terms are closed under their corresponding heaps. It holds trivially for the special case of initializing each configuration with a standard term and its corresponding deBruijn-indexed term, respectively. The proof proceeds by induction on the step relation for each direction of the bisimulation.

5 $\mathscr{CE}$ Machine

Using the calculus of cactus environments defined in the previous section, we derive an abstract machine: the $\mathscr{CE}$ machine. The syntax and semantics are defined in Figure 5.

Syntax

$$\begin{aligned}
s &::= \langle c, \sigma, \mu \rangle && \text{(State)}\\
t &::= i \mid \lambda t \mid t\ t && \text{(Term)}\\
i &\in \mathbb{N} && \text{(Variable)}\\
c &::= t[l] && \text{(Closure)}\\
v &::= \lambda t[l] && \text{(Value)}\\
\mu &::= \varepsilon \mid \mu[l \mapsto \rho] && \text{(Heap)}\\
\rho &::= \bullet \mid c \cdot l && \text{(Environment)}\\
\sigma &::= \square \mid \sigma\ c \mid \sigma\ u && \text{(Context)}\\
l, u, f &\in \mathbb{N} && \text{(Location)}
\end{aligned}$$

Semantics

$$\begin{aligned}
\langle v, \sigma\ u, \mu \rangle &\rightarrow_{\mathscr{CE}} \langle v, \sigma, \mu(u \mapsto v \cdot l) \rangle \text{ where } c \cdot l = \mu(u) && \text{(Upd)}\\
\langle \lambda t[l], \sigma\ c, \mu \rangle &\rightarrow_{\mathscr{CE}} \langle t[f], \sigma, \mu[f \mapsto c \cdot l] \rangle f \notin \text{dom}(\mu) && \text{(Lam)}\\
\langle t\ t'[l], \sigma, \mu \rangle &\rightarrow_{\mathscr{CE}} \langle t[l], \sigma\ t'[l], \mu \rangle && \text{(App)}\\
\langle 0[l], \sigma, \mu \rangle &\rightarrow_{\mathscr{CE}} \langle c, \sigma\ l, \mu \rangle \text{ where } c \cdot l' = \mu(l) && \text{(Var1)}\\
\langle i[l], \sigma, \mu \rangle &\rightarrow_{\mathscr{CE}} \langle (i-1)[l'], \sigma, \mu \rangle \text{ where } c \cdot l' = \mu(l) && \text{(Var2)}
\end{aligned}$$

Fig. 5. Syntax and semantics of the $\mathscr{CE}$ machine.

The machine operates as a small-step implementation of the calculus, extended only with a context to implement the updates from the NVar subderivation ($\sigma\ u$) and the operands from the NEval subderivation ($\sigma\ c$). Much like the calculus, a term t is inserted into an initial state $\langle t[0], \sigma, \varepsilon[0 \mapsto \bullet]$. On the update rule, the current closure is a value,

and there is an update marker as the outermost context. This implies that a variable was entered and that the current closure represents the corresponding value for that variable. Thus, we update the location u that the variable entered, replacing whatever term was entered with the current closure. The Lam rule takes an argument off the context and binds it to a variable, allocating a fresh heap location for the bound variable. This ensures that every instance of the variable will point to this location, and thus the bound term will be evaluated at most once. The App rule simply pushes an argument term in the current environment. The Var1 rule enters the closure pointed to by the environment location, while the Var2 rule traverses the cactus environment to locate the correct closure.

To get some intuition for the $\mathscr{CE}$ machine and how it works, consider Figure 6, evaluation of the term $(\lambda a.(\lambda b.b\,a)\lambda c.c\,a)\,((\lambda i.i)\lambda j.j)$, which is $(\lambda(\lambda 0\,1)\,\lambda 0\,1)\,((\lambda 0)\,\lambda 0)$ with deBruijn indices.

5.1 Correctness

We prove that the reflexive transitive closure of the $\mathscr{CE}$ machine step relation evaluates to a value and heap and empty context iff $\rightarrow_N$ evaluates to the same value and heap.

Theorem 2. (Equivalence)

$$(c,\mu) \rightarrow_N (v,\mu') \;\leftrightarrow\; \langle c,\square,\mu\rangle \xrightarrow{*}_{\mathscr{CE}} \langle v,\square,\mu'\rangle$$

By induction on the $\rightarrow_N$ step relation for one direction, and induction on the reflexive transitive closure of the $\rightarrow_{\mathscr{CE}}$ step relation for the other.

6 Implementation

This section describes how the $\mathscr{CE}$ machine can be mapped directly to x64 insructions. Specifically, we re-define the three instructions given by the TIM [13]: `TAKE`, `ENTER`, and `PUSH`, and implement them with x64 assembly. We also describe several design decisions, as well as some optimizations. All implementation and benchmark code is available at `http://cs.unm.edu/~stelleg/cem-tfp2016.tar.gz`.

Each closure is represented as a ⟨code pointer, environment pointer¿ tuple. The Context is implemented as a stack, with updates represented as a ¡null pointer, environment pointer¿ tuple to differentiate them from closure arguments. The Heap, or cactus environment, is implemented as a heap containing ¡closure, environment pointer¿ structs. As a result, each cell in the heap takes 3 machine words.

6.1 Compilation

The three instructions are given below, with descriptions of their behavior.

- `TAKE`: Pops a context item off the stack. If the item is an update u, the instruction updates the location u with the current closure. If it is an argument c, the instruction binds the closure c to the fresh location in the cactus environment.

$\langle (\lambda(\lambda 0\ 1)\ \lambda 0\ 1)\ ((\lambda 0)\ \lambda 0)[0], \Box, \varepsilon[0 \mapsto \bullet] \rangle$

$\to_{\mathscr{CE}} \langle \lambda(\lambda 0\ 1)\ \lambda 0\ 1[0], \Box(\lambda 0)\ \lambda 0[0], \varepsilon[0 \mapsto \bullet] \rangle$

$\to_{\mathscr{CE}} \langle (\lambda 0\ 1)\ \lambda 0\ 1[1], \Box, \varepsilon[0 \mapsto \bullet][1 \mapsto (\lambda 0)\ \lambda 0[0] \cdot 0] \rangle$

$\to_{\mathscr{CE}} \langle \lambda 0\ 1[1], \Box\lambda 0\ 1[1], \varepsilon[0 \mapsto \bullet][1 \mapsto (\lambda 0)\ \lambda 0[0] \cdot 0] \rangle$

$\to_{\mathscr{CE}} \langle 0\ 1[2], \Box, \varepsilon[0 \mapsto \bullet][1 \mapsto (\lambda 0)\ \lambda 0[0] \cdot 0][2 \mapsto \lambda 0\ 1[1] \cdot 1] \rangle$

$\to_{\mathscr{CE}} \langle 0[2], \Box 1[2], \varepsilon[0 \mapsto \bullet][1 \mapsto (\lambda 0)\ \lambda 0[0] \cdot 0][2 \mapsto \lambda 0\ 1[1] \cdot 1] \rangle$

$\to_{\mathscr{CE}} \langle \lambda 0\ 1[1], \Box 1[2]2, \varepsilon[0 \mapsto \bullet][1 \mapsto (\lambda 0)\ \lambda 0[0] \cdot 0][2 \mapsto \lambda 0\ 1[1] \cdot 1] \rangle$

$\to_{\mathscr{CE}} \langle \lambda 0\ 1[1], \Box 1[2], \varepsilon[0 \mapsto \bullet][1 \mapsto (\lambda 0)\ \lambda 0[0] \cdot 0][2 \mapsto \lambda 0\ 1[1] \cdot 1] \rangle$

$\to_{\mathscr{CE}} \langle 0\ 1[3], \Box, \varepsilon[0 \mapsto \bullet][1 \mapsto (\lambda 0)\ \lambda 0[0] \cdot 0][2 \mapsto \lambda 0\ 1[1] \cdot 1][3 \mapsto 1[2] \cdot 1] \rangle$

$\to_{\mathscr{CE}} \langle 0[3], \Box 1[3], \varepsilon[0 \mapsto \bullet][1 \mapsto (\lambda 0)\ \lambda 0[0] \cdot 0][2 \mapsto \lambda 0\ 1[1] \cdot 1][3 \mapsto 1[2] \cdot 1] \rangle$

$\to_{\mathscr{CE}} \langle 1[2], \Box 1[3]3, \varepsilon[0 \mapsto \bullet][1 \mapsto (\lambda 0)\ \lambda 0[0] \cdot 0][2 \mapsto \lambda 0\ 1[1] \cdot 1][3 \mapsto 1[2] \cdot 1] \rangle$

$\to_{\mathscr{CE}} \langle 0[1], \Box 1[3]3, \varepsilon[0 \mapsto \bullet][1 \mapsto (\lambda 0)\ \lambda 0[0] \cdot 0][2 \mapsto \lambda 0\ 1[1] \cdot 1][3 \mapsto 1[2] \cdot 1] \rangle$

$\to_{\mathscr{CE}} \langle (\lambda 0)\ \lambda 0[0], \Box 1[3]31, \varepsilon[0 \mapsto \bullet][1 \mapsto (\lambda 0)\ \lambda 0[0] \cdot 0][2 \mapsto \lambda 0\ 1[1] \cdot 1][3 \mapsto 1[2] \cdot 1] \rangle$

$\to_{\mathscr{CE}} \langle \lambda 0[0], \Box 1[3]31\lambda 0[0], \varepsilon[0 \mapsto \bullet][1 \mapsto (\lambda 0)\ \lambda 0[0] \cdot 0][2 \mapsto \lambda 0\ 1[1] \cdot 1][3 \mapsto 1[2] \cdot 1] \rangle$

$\to_{\mathscr{CE}} \langle 0[4], \Box 1[3]31, \varepsilon[0 \mapsto \bullet][1 \mapsto (\lambda 0)\ \lambda 0[0] \cdot 0][2 \mapsto \lambda 0\ 1[1] \cdot 1][3 \mapsto 1[2] \cdot 1][4 \mapsto \lambda 0[0] \cdot 0] \rangle$

$\to_{\mathscr{CE}} \langle \lambda 0[0], \Box 1[3]314, \varepsilon[0 \mapsto \bullet][1 \mapsto (\lambda 0)\ \lambda 0[0] \cdot 0][2 \mapsto \lambda 0\ 1[1] \cdot 1][3 \mapsto 1[2] \cdot 1][4 \mapsto \lambda 0[0] \cdot 0] \rangle$

$\to_{\mathscr{CE}} \langle \lambda 0[0], \Box 1[3]31, \varepsilon[0 \mapsto \bullet][1 \mapsto (\lambda 0)\ \lambda 0[0] \cdot 0][2 \mapsto \lambda 0\ 1[1] \cdot 1][3 \mapsto 1[2] \cdot 1][4 \mapsto \lambda 0[0] \cdot 0] \rangle$

$\to_{\mathscr{CE}} \langle \lambda 0[0], \Box 1[3]3, \varepsilon[0 \mapsto \bullet][1 \mapsto \lambda 0[0] \cdot 0][2 \mapsto \lambda 0\ 1[1] \cdot 1][3 \mapsto 1[2] \cdot 1][4 \mapsto \lambda 0[0] \cdot 0] \rangle$

$\to_{\mathscr{CE}} \langle \lambda 0[0], \Box 1[3], \varepsilon[0 \mapsto \bullet][1 \mapsto \lambda 0[0] \cdot 0][2 \mapsto \lambda 0\ 1[1] \cdot 1][3 \mapsto \lambda 0[0] \cdot 1][4 \mapsto \lambda 0[0] \cdot 0] \rangle$

$\to_{\mathscr{CE}} \langle 0[5], \Box, \varepsilon[0 \mapsto \bullet][1 \mapsto \lambda 0[0] \cdot 0][2 \mapsto \lambda 0\ 1[1] \cdot 1][3 \mapsto \lambda 0[0] \cdot 1][4 \mapsto \lambda 0[0] \cdot 0][5 \mapsto 1[3] \cdot 0] \rangle$

$\to_{\mathscr{CE}} \langle 1[3], \Box 5, \varepsilon[0 \mapsto \bullet][1 \mapsto \lambda 0[0] \cdot 0][2 \mapsto \lambda 0\ 1[1] \cdot 1][3 \mapsto \lambda 0[0] \cdot 1][4 \mapsto \lambda 0[0] \cdot 0][5 \mapsto 1[3] \cdot 0] \rangle$

$\to_{\mathscr{CE}} \langle 0[1], \Box 5, \varepsilon[0 \mapsto \bullet][1 \mapsto \lambda 0[0] \cdot 0][2 \mapsto \lambda 0\ 1[1] \cdot 1][3 \mapsto \lambda 0[0] \cdot 1][4 \mapsto \lambda 0[0] \cdot 0][5 \mapsto 1[3] \cdot 0] \rangle$

$\to_{\mathscr{CE}} \langle \lambda 0[0], \Box 51, \varepsilon[0 \mapsto \bullet][1 \mapsto \lambda 0[0] \cdot 0][2 \mapsto \lambda 0\ 1[1] \cdot 1][3 \mapsto \lambda 0[0] \cdot 1][4 \mapsto \lambda 0[0] \cdot 0][5 \mapsto 1[3] \cdot 0] \rangle$

$\to_{\mathscr{CE}} \langle \lambda 0[0], \Box 5, \varepsilon[0 \mapsto \bullet][1 \mapsto \lambda 0[0] \cdot 0][2 \mapsto \lambda 0\ 1[1] \cdot 1][3 \mapsto \lambda 0[0] \cdot 1][4 \mapsto \lambda 0[0] \cdot 0][5 \mapsto 1[3] \cdot 0] \rangle$

$\to_{\mathscr{CE}} \langle \lambda 0[0], \Box, \varepsilon[0 \mapsto \bullet][1 \mapsto \lambda 0[0] \cdot 0][2 \mapsto \lambda 0\ 1[1] \cdot 1][3 \mapsto \lambda 0[0] \cdot 1][4 \mapsto \lambda 0[0] \cdot 0][5 \mapsto \lambda 0[0] \cdot 0] \rangle$

Fig. 6. $\mathscr{CE}$ machine example. Evaluation of $(\lambda(\lambda 0\ 1)\ \lambda 0\ 1)\ ((\lambda 0)\ \lambda 0)$

- `ENTER i`: Enters the closure defined by variable index `i`, the current environment pointer, and the current cactus environment.
- `PUSH m`: Pushes the code location `m` along with the current environment pointer.

Each of these instructions corresponds directly to a corresponding lambda term: abstraction compiles to `TAKE`, application to `PUSH`, and variables to `ENTER`. Each is compiled using a direct implementation of the transition functions of the $\mathscr{CE}$ machine. The mapping from lambda terms can be seen in Figure 7, which defines the compiler. Unlike the TIM, our version of `TAKE` doesn't have an arity; we compile a sequence of lambdas as a sequence of `TAKE` instructions. While we have not compared performance of the two approaches directly, we suspect that `n` inlined `TAKE` instructions should be roughly as fast as a `TAKE n` instruction. Similarly, the `ENTER i` instruction can be implemented either as a loop or unrolled, depending on `i`, and more performance comparisons are needed to determine the trade-off between code size and speed.

$$
\begin{aligned}
C[\![t\ t']\!] &= \texttt{PUSH}\ label_{C[\![t']\!]} : C[\![t]\!] +\!\!+ C[\![t']\!] \\
C[\![\lambda t]\!] &= \texttt{TAKE} : C[\![t]\!] \\
C[\![i]\!] &= \texttt{ENTER}\ i
\end{aligned}
$$

Fig. 7. $\mathscr{CE}$ machine compilation scheme. C compiles a sequence of instructions from a term. The *label* represents a code label: each instruction is given a unique label. The : operator denotes prepending an item to a sequence and $+\!\!+$ denotes concatenating two sequences.

We compile to x64 assembly. Each of the three instructions is mapped onto x64 instructions with a macro. The `PUSH` instruction is particularly simple, consisting of only two x64 instructions (two `push`es, one for the code pointer and one for the environment pointer). This is actually an important point: *thunk creation is only two hardware instructions, regardless of environment size*.

6.2 Machine Literals and Primitive Operations

Following Sestoft [28], we extend the $\mathscr{CE}$ machine to include machine literals and primitive operations. Figure 8 shows the parts of syntax and semantics that are new or modified.

6.3 Omitted Extensions

Our implementation omits a few other standard extensions. Here we address some of these omissions.

Data types are a common extension that we omit [24, 7]. We take the approach of Sestoft [28] that these can be efficiently implemented with pure lambda terms. For example, consider a list data type (in Haskell syntax): `data List a = Cons a (List`

Syntax

$$t ::= i \mid \lambda t \mid t\ t \mid n \mid op \quad \text{(Term)}$$
$$n \in \mathbb{I} \quad \text{(Integer)}$$
$$op ::= + \mid - \mid * \mid / \mid = \mid > \mid < \quad \text{(PrimOp)}$$
$$v ::= \lambda t[l] \mid n[l] \quad \text{(Value)}$$

Integer and Primop Semantics

$$\langle n[l], \sigma\ c, \mu, k\rangle \rightarrow_{\mathscr{CE}} \langle c, \sigma\ n[l], \mu, k\rangle \quad \text{(Int)}$$
$$\langle op[l], \sigma\ n'\ n, \mu, k\rangle \rightarrow_{\mathscr{CE}} \langle op(n', n)[l], \sigma, \mu, k\rangle \quad \text{(Op)}$$

Fig. 8. Extensions to the syntax and semantics of the $\mathscr{CE}$ machine.

`a) | Nil`. This can be represented in pure lambda terms with $Cons = \lambda h.\lambda t.\lambda c.\lambda n.c\ h\ t$ and $Nil = \lambda c.\lambda n.n$.

Let bindings are another term commonly included in functional language compilers, even in the internal representation [7, 24]. Non-recursive let is syntactic sugar for a lambda binding and application, and we treat it as such. This approach helps ensure that we can of compile arbitrary lambda terms, while some approaches require pre-processing [28, 13].

Recursive let bindings are a third omission. Here we follow Rozas [26]: If it can be represented in pure lambda terms, it should be. Thus, we implement recursion using the standard Y combinator. In the case of mutual recursion, we use the Y combinator in conjunction with a church tuple of the mutually recursive functions. Without the appropriate optimizations [26], this approach has high overhead, as we discuss in Section 7.1.

6.4 Optimizations

The $\mathscr{CE}$ implementation described in the previous section is completely unoptimized. For example, no effort is expended to discover global functions to avoid costly jumps to pointers in the heap [24]. Indeed, every variable reference will look up the code pointer in the shared environment and jump to it. There is also no implementation of control flow analysis as used by Rozas to optimize away the Y combinator. Thus, every recursive call exhibits the large overhead involved in re-calculating the fixed point of the function.

We do, however, implement two basic optimizations, primarily to reduce the load on the heap:

- `POP`: A `TAKE` instruction can be converted to a `POP` instruction that throws away the operand on the top of the stack if there are no variables bound to the λ term in question. For example, the function $\lambda x.\lambda y.x$ can be implemented with `TAKE`, `POP`, `ENTER 0`.

– `ENTERVAL`: An `ENTER` instruction, when entering a closure that is already a value, should not push an update marker onto the stack. This shortcut prevents unnecessary writes to the stack and heap [24, 14, 28].

6.5 Garbage Collection

We have implemented a simple mark and sweep garbage collector with the property that it does not require two spaces because constant-sized closures in the heap allow a linked-list representation for the free cells. Indeed, while the abstract machine from Section 5 increments a free heap counter, the actual implementation uses the next free cell in the linked list.

Because the focus of this paper is not on the performance of garbage collection, we ensure the benchmarks in Section 7 are not dominated by GC time.

7 Performance Evaluation

This section reports experiments that assess the strengths and weaknesses of the $\mathscr{CE}$ machine. We evaluate using benchmarks from the `nofib` benchmark suite. Because we have implemented only machine integers, and must translate the examples by hand, we use a subset of the `nofib` suite that excludes floating point values and arrays. A list of the benchmarks used and a brief description is given in Figure 9.

– **exp3:** A Peano arithmetic benchmark. Computes 3^8 and prints the result.
– **queens:** Computes the number of solutions to the nqueens problem for an n by n board.
– **primes:** A simple primes sieve that computes the nth prime.
– **digits-of-e1:** A calculation of the first *n* digits of *e* using continued fractions.
– **digits-of-e2:** Another calculation of the first *n* digits of *e* using an infinite series.
– **fib:** Naively computes the nth Fibonacci number.
– **fannkuch:** Counts the number of reverses of a subset of a list.
– **tak:** A synthetic benchmark involving basic recursion.

Fig. 9. Description of Benchmarks

We compare the $\mathscr{CE}$ machine with two existing implementations:

– GHC: The Glasgow Haskell compiler. A high performance, optimizing compiler based on the STG machine [24].
– UHC: The Utrecht Haskell compiler. An optimizing compiler based on the GRIN machine [7, 12].

We use GHC version 7.10.3 and UHC version 1.1.9.3. We compile with -O0 and -O3, and show the results for both. Where possible, we pre-allocate a heap of 1GB to avoid measuring the performance of GC implementations. The tests were run on an Intel(R) Xeon(R) CPU E5-4650L at 2.60GHz, running Linux version 3.16.

7.1 Results

Figure 10 gives the benchmark results. In general, we are outperformed by GHC, sometimes significantly, and we outperform UHC. We spend the remainder of the section analyzing these performance differences.

	$\mathscr{CE}$	GHC -O0	UHC -O0	GHC -O3	UHC -O3
`exp3 8`	1.530	1.176	3.318	1.038	2.286
`tak 16 8 0`	0.366	0.146	1.510	0.006	1.416
`primes 1500`	0.256	0.272	1.518	0.230	1.532
`queens 9`	0.206	0.050	0.600	0.012	0.598
`fib 35`	2.234	0.872	10.000	0.110	8.342
`digits-of-e1 1000`	3.576	1.274	21.938	0.118	22.010
`digits-of-e2 1000`	0.404	0.792	3.430	0.372	3.278
`fannkuch 8`	0.560	0.084	2.184	0.048	2.196

Fig. 10. Machine Literals Benchmark Results. Measurement is wall clock time, units are seconds. Times averaged over 5 runs.

There are many optimizations built into the abstract machine underlying GHC, but profiling indicates that three in particular lead to much of the performance disparity:

- **Register allocation:** The $\mathscr{CE}$ machine has no register allocator. In contrast, by passing arguments to functions in registers, GHC avoids much heap thrashing.
- **Unpacked literals:** This allows GHC to keep machine literals without tags in registers for tight loops. In contrast, the $\mathscr{CE}$ machine operates entirely on the stack, and has a code pointer associated with every machine literal.
- **Y combinator:** Because recursion in the $\mathscr{CE}$ machine is implemented with a Y combinator, it performs poorly. This could be alleviated with CFA-based techniques, similar to those used in [26].

Lack of register allocation is the primary current limitation of the $\mathscr{CE}$ machine. The STG machine pulls the free variables into registers, allowing tight loops with everything kept in registers. However, it is less clear how to effectively allocate registers in a fully shared environment setting. That said, we believe being lazier about register allocation, e.g., not loading values into registers that may not be used, could have some performance benefits.

To isolate the effect of register allocation and unpacked machine literals, we replace machine integers with Church numerals in a compatible subset of the evaluation programs. Figure 11 shows the performance results with this modification, which are much improved, with the $\mathscr{CE}$ machine occasionally even outperforming optimized GHC.

Next, we consider the disparity due to the Y-combinator, by running a simple exponentiation example with Church numerals, calculating $3^8 - 3^8 = 0$. In this case, the $\mathscr{CE}$ machine significantly outperform both GHC and UHC, as seen in Figure 12 .

These results give us confidence that by adding the optimizations mentioned above, among others, the $\mathscr{CE}$ machine has the potential to be the basis of a real-world compiler.

	$\mathscr{CE}$	GHC -O0	UHC -O0	GHC -O3	UHC -O3
`tak 14 7 0`	1.610	2.428	7.936	1.016	7.782
`primes 32`	0.846	1.494	4.778	0.666	5.290
`queens 8`	0.242	0.374	1.510	0.154	1.508
`fib 23`	0.626	0.940	5.026	0.468	5.336
`digits-of-e2 6`	0.138	1.478	5.056	0.670	5.534
`fannkuch 7`	0.142	0.124	0.796	0.040	0.808

Fig. 11. Church Numeral Benchmark Results. Measurement is wall clock time, units are seconds. Times averaged over 5 runs.

	$\mathscr{CE}$	GHC -O0	UHC -O0	GHC -O3	UHC -O3
`pow 3 8`	0.564	1.994	4.912	0.906	4.932

Fig. 12. Church Numeral Exponentiation Benchmark Results. Measurement is wall clock time, units are seconds. Times averaged over 5 runs.

We discuss how some of these optimizations can be applied to the $\mathscr{CE}$ machine in Section 8.

7.2 The Cost of the Cactus

Recall that variable lookup is linear in the index of the variable, following pointers until the index is zero. As one might guess, the lookup cost is high. For example, for the queens benchmark without any optimizations, variable lookup took roughly $80-90\%$ of the $\mathscr{CE}$ machine runtime, as measured by profiling. Much of that cost was for lookups of known combinators, however, so for the benchmarks above we added the inlining mentioned in the previous section. Still, even with this simple optimization, variable lookup takes roughly 50% of execution time. There is some variation across benchmarks, but this is a rough approximation for the average cost. We discuss how this cost could be addressed in future work in Section 8.

8 Discussion and Related Work

Some related work was discussed earlier to provide background and context. Here, we briefly and explicitly compare our approach with earlier work. We also discuss areas for future work.

8.1 Closure Representation

Appel and Shao [29] and Appel and Jim [2] both cover the design space for closure representation, and develop an approach called *safely linked closures*. It uses flat closures when there is no duplication, and links in a way that preserves liveness, to prevent violation of the *safe for space complexity* (SSC) rule [1]. While we do not address SSC or garbage collection in general, understanding the relationship between SSC and shared

environment call-by-need is an interesting area for future work. In particular, hot environments with no sharing could benefit greatly from replacing shared structure with flat.

8.2 Eval/Apply vs. Push/Enter

Marlow and Peyton Jones describe two approaches to the implementation of function application: eval/apply, where the function is evaluated and then passed the necessary arguments, and push/enter, where the arguments are pushed onto the stack and the function code is entered [22]. They conclude that despite push/enter being a standard approach to lazy machines, eval/apply performs better. While our current approach uses push/enter, investigating whether eval/apply could be usefully implemented for a shared environment machine like the $\mathscr{CE}$ machine is an interesting avenue for future work.

8.3 Collapsed Markers

Friedman et al. show how a machine can be designed to prevent multiple adjacent update markers being pushed onto the stack [14]. This property is desirable because multiple adjacent update markers are always updated with the same value. They give examples showing that in some cases, these redundant update markers can cause an otherwise constant-space stack to grow unbounded. They implement an optimization that collapses update markers by adding a layer of indirection between heap locations and closures. We propose a similar approach, but without the performance hit caused by an extra layer of indirection. Upon a variable dereference the $\mathscr{CE}$ machine checks if the top of the stack is an update. If it is, instead of pushing a redundant update marker onto the stack, it replaces the closure in the heap at the desired location with an update marker. Then, the variable dereference rule checks for an update marker upon dereference, and will update accordingly. We have begun to implement this optimization, but leave the full implementation and description for future work.

8.4 Register Allocation

One advantage of flat environments is that register allocation is straightforward [1, 24, 31]. It is less obvious how to do register allocation with the $\mathscr{CE}$ machine. We speculate that it should be possible to do a sufficient job, particularly in the cases that matter most, e.g. unboxed machine literals, though certainly not easy.

One possible approach that could work well with our shared environment approach would be to only load strict free variables into registers. That is to say, some environment variables may not be used, and only the ones we know will be used should be loaded into registers, while the rest should be loaded on demand.

8.5 Verification

A signal property of our implementation is its simplicity, which makes it an attractive target for a verified compiler. Because it avoids complexities required for flat environment implementations, e.g., black hole updates, basing a verified compiler on this machine is another exciting area for future work.

9 Conclusion

Lazy evaluation has long suffered from high overhead when delaying computations. While strictness analysis helps to alleviate this issue, there are and there always will be cases it cannot catch. Existing implementations choose to pay an up-front cost of constructing a flat environment to ensure efficient variable lookup if a delayed computation is used. When a delayed computation is never used, this overhead is wasted. In this paper we have presented a novel approach that minimizes this overhead. We have achieved this overhead minimization by taking an old idea, shared environments, and using them in a novel way. By leveraging the structure inherent in a shared environment to share results of computation, we have avoided some of the overheads involved in delaying a computation.

We conclude by summarizing the key points of this paper. First, a shared environment, explicitly represented as a cactus stack, is a natural way to share the results of computation as required by lazy evaluation. Second, this approach is in a sense *lazier* about lazy evaluation than existing implementations because it avoids some unnecessary packaging. Third, this approach can be formalized in both big-step and small-step semantics. Lastly, the abstract machine can be implemented as a compiler in a straightforward way, yielding performance comparable to existing implementations.

10 Acknowledgments

This material is based upon work supported by the National Science Foundation under grant CCF-1422840, NSF (1518878,1444871), DARPA (FA8750-15-C-0118), AFRL (FA8750-15-2-0075), the Sandia National Laboratories Academic Alliance, and the Santa Fe Institute. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energys National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

1. Appel, A.W.: Compiling with continuations. Cambridge University Press (1992)
2. Appel, A.W., Jim, T.: Optimizing closure environment representations. Tech. rep. (1988)
3. Appel, A.W., MacQueen, D.B.: Standard ML of New Jersey. In: Programming Language Implementation and Logic Programming. pp. 1–13 (1991)
4. Ariola, Z.M., Maraist, J., Odersky, M., Felleisen, M., Wadler, P.: A call-by-need lambda calculus. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 233–246 (1995)
5. Barendregt, H.P.: The Lambda Calculus. North-Holland Amsterdam (1984)
6. Biernacka, M., Danvy, O.: A concrete framework for environment machines. ACM Transactions on Computational Logic 9(1), 6 (2007)
7. Boquist, U., Johnsson, T.: The GRIN project: A highly optimising back end for lazy functional languages. In: Implementation of Functional Languages. pp. 58–84. Springer (1997)
8. Curien, P.L.: An abstract framework for environment machines. Theoretical Computer Science 82(2), 389–402 (1991)

9. Danvy, O., Millikin, K., Munk, J., Zerny, I.: On inter-deriving small-step and big-step semantics: A case study for storeless call-by-need evaluation. Theoretical Computer Science 435, 21–42 (2012)
10. Danvy, O., Zerny, I.: A synthetic operational account of call-by-need evaluation. In: Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming. pp. 97–108 (2013)
11. Diehl, S., Hartel, P., Sestoft, P.: Abstract machines for programming language implementation. Future Generation Computer Systems 16(7), 739–751 (2000)
12. Dijkstra, A., Fokker, J., Swierstra, S.D.: The architecture of the Utrecht Haskell compiler. In: Proceedings of the 2nd ACM SIGPLAN symposium on Haskell. pp. 93–104 (2009)
13. Fairbairn, J., Wray, S.: TIM: A simple, lazy abstract machine to execute supercombinators. In: Functional Programming Languages and Computer Architecture. pp. 34–45 (1987)
14. Friedman, D., Ghuloum, A., Siek, J., Winebarger, O.: Improving the lazy Krivine machine. Higher-Order and Symbolic Computation 20, 271–293 (2007)
15. Hauck, E., Dent, B.A.: Burroughs' B6500/B7500 stack mechanism. In: Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference. pp. 245–251 (1968)
16. Ichbiah, J.: Rationale for the design of the Ada programming language. Cambridge University Press (1991)
17. Ingerman, P.Z.: A way of compiling procedure statements with some comments on procedure declarations. Commun. ACM 4(1), 55–58 (1961)
18. Johnsson, T.: Efficient compilation of lazy evaluation. In: Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction. pp. 58–69 (1984)
19. Krivine, J.: A call-by-name lambda-calculus machine. Higher-Order and Symbolic Computation 20(3), 199–207 (2007)
20. Landin, P.J.: The mechanical evaluation of expressions. The Computer Journal 6(4), 308–320 (1964)
21. Leroy, X.: The ZINC experiment: an economical implementation of the ML language. Technical report 117, INRIA (1990)
22. Marlow, S., Peyton Jones, S.: Making a fast curry: push/enter vs. eval/apply for higher-order languages. Journal of Functional Programming 16(4-5), 415–449 (2006)
23. Mycroft, A.: Abstract interpretation and optimising transformations for applicative programs. Ph.D. thesis (1982)
24. Peyton Jones, S.L.: Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. Journal of Functional Programming 2(2), 127–202 (1992)
25. Peyton Jones, S.L., Lester, D.R.: Implementing Functional Languages. Prentice-Hall, Inc. (1992)
26. Rozas, G.J.: Taming the Y operator. ACM SIGPLAN Lisp Pointers (1), 226–234 (1992)
27. Sabry, A., Lumsdaine, A., Garcia, R.: Lazy evaluation and delimited control. Logical Methods in Computer Science 6 (2010)
28. Sestoft, P.: Deriving a lazy abstract machine. Journal of Functional Programming 7(3), 231–264 (1997)
29. Shao, Z., Appel, A.W.: Space-efficient closure representations. In: Proceedings of the 1994 ACM Conference on Lisp and Functioanl Programming (1994)
30. Stenstrom, P.: VLSI support for a cactus stack oriented memory organization. In: System Sciences, 1988. Vol. I. Architecture Track, Proceedings of the Twenty-First Annual Hawaii International Conference on. vol. 1, pp. 211–220 (1988)
31. Terei, D.A., Chakravarty, M.M.: An LLVM backend for GHC. In: Proceedings of the Third ACM Haskell Symposium on Haskell. pp. 109–120. Haskell '10 (2010)
32. Wadler, P., Hughes, R.J.M.: Projections for strictness analysis. In: Functional Programming Languages and Computer Architecture. pp. 385–407 (1987)