Evaluating Call-By-Need on the Control Stack

Stephen Chang,[⋆] David Van Horn,[⋆⋆] and Matthias Felleisen[⋆]

PLT & PRL, Northeastern University, Boston, MA 02115

Abstract. Ariola and Felleisen's call-by-need λ-calculus replaces a variable occurrence with its value at the last possible moment. To support this gradual notion of substitution, function applications—once established—are never discharged. In this paper we show how to translate this notion of reduction into an abstract machine that resolves variable references via the control stack. In particular, the machine uses the *static address* of a variable occurrence to extract its current value from the *dynamic control stack*.

1 Implementing Call-by-need

Following Plotkin [1], Ariola and Felleisen characterize the by-need λ-calculus as a variant of β:

$$(\lambda x.E[x])\ V = (\lambda x.E[V])\ V\ \ ,$$

and prove that a machine is an algorithm that searches for a (generalized) value via the leftmost-outermost application of this new reduction [2].

Philosophically, the by-need λ-calculus has two implications:

1. First, its existence says that imperative assignment isn't truly needed to implement a lazy language. The calculus uses only one-at-a-time substitution and does not require any store-like structure. Instead, the by-need β suggests that a variable dereference is the resumption of a continuation of the function call, an idea that Garcia et al. [3] recently explored in detail by using delimited control operations to derive an abstract machine from the by-need calculus. Unlike traditional machines for lazy functional languages, Garcia et al.'s machine eliminates the need for a store by replacing heap manipulations with control (stack) manipulations.
2. Second, since by-need β does not remove the application, the binding structure of programs—the association of a function parameter with its value—remains the same throughout a program's evaluation. This second connection is the subject of our paper. This binding structure *is* the control stack, and thus we have that in call-by-need, *static* addresses can be resolved in the *dynamic* control stack.

⋆ Partially supported by grants from the National Science Foundation.

⋆⋆ Supported by NSF Grant 0937060 to the CRA for the CIFellow Project.

Our key innovation is the CK+ machine, which refines the abstract machine of Garcia et al. by making the observation that when a variable reference is in focus, the location of the corresponding binding context in the dynamic control stack can be determined by the lexical index of the variable. Whereas Garcia et al.'s machine linearly traverses their control stack to find a specific binding context, our machine employs a different stack organization where indexing can be used instead of searching. Our machine organization also simplifies the hygiene checks used by Garia et al., mostly because it explicitly maintains Garcia et al.'s "well-formedness" condition on machine states, instead of leaving it as a side condition.

The paper starts with a summary of the by-need λ-calculus and the abstract textual machine induced by the standard reduction theorem. We then show how to organize the machine's control stack so that when the control string is a variable reference, the machine is able to use the lexical address to compute the location of the variable's binding site in the control stack.

2 The Call-by-need λ-calculus, the de Bruijn Version

The terms of the by-need λ-calculus are those of the λ-calculus [4], which we present using de Bruijn's notation [5], i.e., lexical addresses replace variables:

$$M ::= n \mid \lambda.M \mid M\ M$$

where $n \in \mathbb{N}$. The set of values is just the set of abstractions:

$$V ::= \lambda.M$$

One of the fundamental ideas of call-by-need is to evaluate the argument in an application only when it is "needed," and when the argument *is* needed, to evaluate that argument only once. Therefore, the by-need calculus cannot use the β notion of reduction because doing so may evaluate the argument when it is not needed, or may cause the argument to be evaluated multiple times. Instead, β is replaced with the *deref* notion of reduction:

$$(\lambda.E[n])\ V\ \mathbf{need}\ (\lambda.E[V])\ V, \quad \lambda \text{ binds } n \qquad \textit{deref}$$

The *deref* notion of reduction requires the argument in an application to be a value and requires the body of the function to have a special shape. This special shape captures the demand-driven substitution of values for variables that is characteristic of call-by-need. In the *deref* notion of reduction, when a variable is replaced with the value V, some renaming may still be necessary to avoid capture of free variables in V, but for now, we assume a variant of Barendregt's hygiene condition for de Bruijn indices and leave all necessary renaming implicit.

Here is the set of evaluation contexts E:

$$E ::= [\,] \mid E\ M \mid (\lambda.E)\ M \mid (\lambda.E'[n])\ E$$

Like all contexts, an evaluation context is an expression with a hole in the place of a subexpression. The first evaluation context is an empty context that is just a hole. The second evaluation context indicates that evaluation of applications proceeds in a leftmost-outermost order. This is similar to how evaluation proceeds in the by-name λ-calculus [1]. Unlike call-by-name, however, call-by-need defers dealing with arguments until absolutely necessary. It therefore demands evaluation within the body of a let-like binding. The third evaluation context captures this notion. This context allows the *deref* notion of reduction to search under applied λs for variables to substitute. The fourth evaluation context explains how the demand for a parameter's value triggers and directs the evaluation of the function's argument. In the fourth evaluation context, the visible λ binds n in $\lambda.E'[n]$. This means that there are n additional λ abstractions in E' between n and its binding λ.

To make this formal, let us define the function $\varDelta : E \rightarrow \mathbb{N}$ as:

$$\begin{aligned} \varDelta([\,]) &= 0 & \varDelta((\lambda.E'[n])\ E) &= \varDelta(E) \\ \varDelta(E\ M) &= \varDelta(E) & \varDelta((\lambda.E)\ M) &= \varDelta(E)+1 \end{aligned}$$

With $\varDelta$, the side condition for the fourth evaluation context is $n = \varDelta(E')$.

Unlike β, *deref* does not remove the argument from a term when substitution is complete. Instead, a term $(\lambda.M)\ N$ is interpreted as a term M and an environment where the variable (index) bound by λ is associated with N. Since arguments are never removed from a by-need term, reduced terms are not necessarily values. In the by-need λ-calculus, reductions produce "answers" a (this representation of answers is due to Garcia et al. [3]):

$$\begin{aligned} a &::= A[V] && \text{answers} \\ A &::= [\,] \mid (\lambda.A)\ M && \text{answer contexts} \end{aligned}$$

Answer contexts A are a strict subset of evaluation contexts E.

Since both the operator and the operand in an application reduce to answers, two additional notions of reduction are needed:

$$\begin{aligned} (\lambda.A[V])\ M\ N\ &\mathbf{need}\ (\lambda.A[V\ N])\ M && \textit{assoc-L} \\ (\lambda.E[n])\ ((\lambda.A[V])\ M)\ &\mathbf{need}\ (\lambda.A[(\lambda.E[n])\ V])\ M, \ \text{ if } \varDelta(E) = n && \textit{assoc-R} \end{aligned}$$

As mentioned, some adjustments to de Bruijn indices are necessary when performing substitution in λ-calculus terms. For example, in a *deref* reduction, every free variable in the substituted V must be incremented by $\varDelta(E)+1$. Otherwise, the indices representing free variables in V no longer count the number of λs between their occurrence and their respective binding λs. Similar adjustments are needed for the *assoc-L* and *assoc-R* reductions, where subterms are also pulled under λs.

Formally, define a function $\uparrow$ that takes three inputs: a term M, an integer x, and a variable (index) m, and increments all free variables in M by x, where a free variable is defined to be an index n such that $n \geq m$. In this paper, we use the notation $M{\uparrow}^x_m$. Here is the formal definition of $\uparrow$:

$$n{\uparrow}^x_m = n + x, \text{ if } n \geq m \qquad (M\ N){\uparrow}^x_m = ((M{\uparrow}^x_m)\ (N{\uparrow}^x_m))$$
$$n{\uparrow}^x_m = n, \quad \text{ if } n < m \qquad \lambda.M{\uparrow}^x_m = \lambda.(M{\uparrow}^x_{m+1})$$

Using the $\uparrow$ function for index adjustments, the notions of reduction are:

$$(\lambda.E[n])\ V\ \textbf{need}\ (\lambda.E[V{\uparrow}_0^{\Delta(E)+1}])\ V, \text{ if } \Delta(E) = n \qquad \textit{deref}$$
$$(\lambda.A[V])\ M\ N\ \textbf{need}\ (\lambda.A[V\ (N{\uparrow}_0^{\Delta(A)+1})])\ M \qquad \textit{assoc-L}$$
$$(\lambda.E[n])\ ((\lambda.A[V])\ M)\ \textbf{need}\ (\lambda.A[((\lambda.E[n]){\uparrow}_0^{\Delta(A)+1})\ V])\ M, \text{ if } \Delta(E) = n \qquad \textit{assoc-R}$$

It is acceptable to apply the Δ function to A because A is a subset of E.

3 Standard Reduction Machine

In order to derive an abstract machine from the by-need λ-calculus, Ariola and Felleisen prove a Curry-Feys-style Standardization Theorem. Roughly, the theorem states that a term M reduces to a term N in a canonical manner if M reduces to N in the by-need calculus.

The theorem thus determines a state machine for reducing programs to answers. The initial state of the machine is the program, the collection of states is all possible programs, and the final states are answers. Transitions in the state machine are equivalent to reductions in the calculus:

$$E[M] \longmapsto_{\textbf{need}} E[M'], \text{ if } M \textbf{ need } M'$$

where E represents the same evaluation contexts that are used to define the demand-driven substitution of variables in the *deref* notion of reduction.

The machine is deterministic because all programs M satisfy the unique decomposition property. This means that M is either an answer or can be uniquely decomposed into an evaluation context and a redex. Hence, we can use the state machine transitions to define an evaluator function:

$$\texttt{eval}_{\textbf{need}}(M) = \begin{cases} a, & \text{if } M \longmapsto\!\!\!\!\to_{\textbf{need}} a \\ \bot, & \text{if for all } M \longmapsto\!\!\!\!\to_{\textbf{need}} N,\ N \longmapsto_{\textbf{need}} L \end{cases}$$

Lemma 1. *$\texttt{eval}_{\textbf{need}}$ is a total function.*

Proof. The lemma follows from the standard reduction theorem [2]. □

4 The CK+ Machine

A standard reduction machine specifies evaluation steps at a high-level of abstraction. Specifically, at each evaluation step in the machine, the entire program is partitioned into an evaluation context and a redex. This repeated partitioning is inefficient because the evaluation context at any given evaluation step

tends to share a large common prefix with the evaluation context in the previous step. To eliminate this inefficiency, Felleisen and Friedman propose the CK machine [6, Chapter 6], an implementation for a standard reduction machine of a call-by-value language. Consider the following call-by-value evaluation:

$$\begin{aligned}
&((\lambda w.w)\ \underline{((\lambda x.(x\ ((\lambda y.y)\ \lambda z.z)))\ \lambda x.x)}) \\
\longmapsto_v\ &((\lambda w.w)\ ((\lambda x.x)\ \underline{((\lambda y.y)\ \lambda z.z)})) \\
\longmapsto_v\ &((\lambda w.w)\ \underline{((\lambda x.x)\ \lambda z.z)}) \\
\longmapsto_v\ &\underline{((\lambda w.w)\ \lambda z.z)} \\
\longmapsto_v\ &\lambda z.z
\end{aligned}$$

In each step, the β_v redex is underlined. The evaluation contexts for the first and third term are the same, $((\lambda w.w)\ [\,])$, and it is contained in the evaluation context for the second term, $((\lambda w.w)\ ((\lambda x.x)\ [\,]))$. Although the evaluation contexts in the first three terms have repeated parts, a standard reduction machine for the call-by-value calculus must re-partition the program at each evaluation step.

The CK machine improves upon the standard reduction machine for the by-value λ-calculus by eliminating redundant search steps. While the standard reduction machine uses whole programs as machine states, a state in the CK machine is divided into separate subterm (C) and evaluation context (K) registers. More precisely, the C in the CK machine represents a control string, i.e., the subterm to be evaluated, and the K is a continuation, which is a data structure that represents an evaluation context in an "inside-out" manner. The original program can be reconstructed from a CK machine state by "plugging" the expression in the C subterm register into the context represented by K. When the control string is a redex, the CK machine can perform a reduction, just like the standard reduction machine. Unlike the standard reduction machine though, the CK machine still remembers the previous evaluation context in the context register and can therefore resume the search for the next redex from the contractum in C and the evaluation context in K.

4.1 CK+ Machine States

We introduce the CK+ machine, a variant of the CK machine, for the by-need λ-calculus. The CK+ machine is also a modification of the abstract machine of Garcia et al. [3]. The machine states for the CK+ machine are specified in figure 1. The core CK+ machine has three main registers, a control string (C), a "renaming" environment (R), and a continuation stack ($\bar{K}$).

In figure 1, the $\ldots$ notation means "zero or more of the preceeding element" and in the stack $\|k,\ K,\ \ldots\|$, the partial stack frame k is the top of the stack. The initial CK+ machine state is $\langle M, (\ ), \|\mathtt{mt}\|\rangle$, where M is the given program, $(\ )$ is an empty renaming environment, and $\|\mathtt{mt}\|$ is a stack with just one element, an empty frame.

$S, T ::= \langle C, R, \bar{K} \rangle$	machine states
$C ::= M$	control strings
$R ::= (i,\ \ldots)$	renaming environments
$i \in \mathbb{N}$	offsets
$\bar{K} ::= \lVert k,\ K,\ \ldots \rVert$	continuation stacks
$K ::= (\texttt{bind}\ M\ R\ k)$	complete stack frames
$k ::= \texttt{mt} \mid (\texttt{arg}\ M\ R\ k) \mid (\texttt{op}\ \bar{K}\ k)$	partial stack frames

Fig. 1. CK+ machine states.

4.2 Renaming Environment

As mentioned in section 2, substitution requires some form of renaming, which manifests itself as lexical address adjustments when using a de Bruijn representation of terms. Instead of adjusting addresses directly, the CK+ machine delays the adjustment by keeping track of offsets for all free variables in the control string in a separate *renaming environment*. The delayed renaming is forced when a variable occurrence is evaluated, at which point the offset is added to the variable before it is used to retrieve its value from the control stack.

Here we use lists for renaming environments and the offset corresponding to variable n, denoted $R(n)$, is the n-th element in R (0-based). The : function is cons, and the function $M \Leftarrow R$ applies a renaming environment R to a term M, yielding a term like M except with appropriately adjusted lexical addresses:

$$
\begin{aligned}
M \Leftarrow (\ ) &= M \\
n \Leftarrow R &= n + R(n) \\
(\lambda.M) \Leftarrow R &= \lambda.(M \Leftarrow (0 : R)) \\
(M\ N) \Leftarrow R &= ((M \Leftarrow R)\ (N \Leftarrow R))
\end{aligned}
$$

Because the CK+ machine uses renaming environments, the $\uparrow$ function from section 2 is replaced with an operation on R. When the machine needs to increment all free variables in a term, it uses the $\oplus$ function to increment all offsets in the renaming environment that accompanies the term. The notation $R \oplus x$ means that all offsets in renaming environment R are incremented by x. Thus, the use of indices in place of variables enables hygiene maintenance through simple incrementing and decrementing of the indices. As a result, we have eliminated the need to keep track of the "active variables" that are present in Garcia et al.'s machine [3, Section 4.5].

4.3 Continuations and the Continuation Stack

Like the CK machine, the CK+ machine represents evaluation contexts as continuations. The [ ] context is represented by the $\texttt{mt}$ continuation. An evaluation

context $E[([\,]\ N)]$ is represented by a continuation (`arg` $M\ R\ k$) where k represents E and $(M{\Leftarrow}R) = N$. An evaluation context $E[(\lambda.[\,])\ N]$ is represented by a continuation (`bind` $M\ R\ k$) where k represents E and $(M{\Leftarrow}R) = N$. Finally, the $E[(\lambda.E'[n])\ [\,]]$ context is represented by an (`op` $\bar{K}\ k$) continuation. The E' under the λ in the evaluation context is represented by the nested $\bar{K}$ stack in the continuation and the E surrounding the evaluation context corresponds to the k in the continuation. The `op` continuation does not need to remember the n variable in the evaluation context because the variable can be derived from the length of $\bar{K}$.

The contents of the $\bar{K}$ register represent the control stack of the program and we refer to an element of this stack as a frame. The key difference between the CK+ machine and Garcia et al.'s machine is in the organization of the frames of the stack. Instead of a flat list of frames like in Garcia et al.'s machine, our control stack frames are groups of nested continuations of a special shape. Thus we also call our control stack a "continuation stack." We use two kinds of frames, partial and complete. The first frame in the continuation stack is always a partial one, while all others are complete. The outermost continuation of a complete frame is a `bind` and all other nested pieces of a complete frame are `op`, `arg`, or `mt`. Thus, not counting the first partial frame, there is exactly one frame in the control stack for every `bind` continuation in the program. As a result, the machine can use a variable (lexical address) n to find the `bind` corresponding to that variable in the control stack.

4.4 Maintaining the Continuation Stack

Each frame of the control stack, with the exception of the top frame, has the shape (`bind` $M\ R\ k$), where k is a partial frame that contains no additional `bind` frames. In order for the continuation stack to maintain this invariant, CK+ machine transitions must adhere to two conditions:

1. When a machine transition is executed, only the top partial frame of the stack is updated unless the instruction descends under a λ.
2. If a machine transition descends under a λ, the partial frame on top of the stack is completed and a new `mt` partial frame is pushed onto the stack.

Essentially, the top frame in the stack "accumulates context" until a λ is encountered, at which time the top partial frame becomes a complete frame. Maintaining evaluation contexts for the program in this way implies a major consequence for the CK+ machine:

> when the control string is a variable n, then the binding for n is $(n + R(n) + 1)$ stack frames away.

4.5 Relating Machine States to Terms

Figure 2 defines the φ function, which converts machine states to λ-terms. It uses the $M{\Leftarrow}R$ function to apply the renaming environment to the control string and

$$\varphi(\langle M, R, \bar{K}\rangle) = \bar{K}[M \Leftarrow R]$$

$$\begin{aligned} \|k,\ K,\ \ldots\|\,[M] &= \ldots[K[k[M]]] \\ \mathtt{mt}[M] &= M \\ (\mathtt{arg}\ N\ R\ k)[M] &= k[(M\ (N \Leftarrow R))] \\ (\mathtt{op}\ \bar{K}\ k)[M] &= k[(\lambda.\bar{K}[\mathtt{len}(\bar{K}) - 1])\ M] \\ (\mathtt{bind}\ N\ R\ k)[M] &= k[(\lambda.M)\ (N \Leftarrow R)] \end{aligned}$$

Fig. 2. φ converts CK+ machine states to λ-calculus terms.

then uses a family of "plug" functions, dubbed $\cdot[\cdot]$, to plug the renamed control string into the hole of the context represented by the continuation component of the state. Figure 2 also defines these plug functions, where $K[M]$ yields the term obtained by plugging M into the context represented by K, and $\bar{K}[M]$ yields the term when M is plugged into the context represented by the continuation stack $\bar{K}$.

4.6 CK+ Machine State Transitions

Figure 3 shows the first four state transitions for the CK+ machine. The $+\!\!+$ notation indicates an "append" operation for the continuation stack. Since the purpose of the CK+ machine is to remember intermediate states in the search for a redex, three of the first four rules are search rules. They shift pieces of the control string to the $\bar{K}$ register. For example, the [shift-arg] transition shifts the argument of an application to the $\bar{K}$ register.

$\longmapsto_{ck+}$

	[shift-arg]
$\langle (M\ N), R, \|k,\ K,\ \ldots\|\rangle$	$\langle M, R, \|(\mathtt{arg}\ N\ R\ k),\ K,\ \ldots\|\rangle$
	[descend-λ]
$\langle \lambda.M, R, \|(\mathtt{arg}\ N\ R'\ k),\ K,\ \ldots\|\rangle$	$\langle M, 0\!:\!R, \|\mathtt{mt},\ (\mathtt{bind}\ N\ R'\ k),\ K,\ \ldots\|\rangle$
	[lookup-arg]
$\langle n, R, \bar{K}+\!\!+\|(\mathtt{bind}\ N\ R'\ k),\ K,\ \ldots\|\rangle$ where $\mathtt{len}(\bar{K}) = n + R(n) + 1$	$\langle N, R', \|(\mathtt{op}\ \bar{K}\ k),\ K,\ \ldots\|\rangle$
	[resume]
$\langle V, R, \|(\mathtt{op}\ \bar{K}\ k),\ K,\ \ldots\|\rangle$	$\langle V, R', \bar{K}+\!\!+\|(\mathtt{bind}\ V\ R\ k),\ K,\ \ldots\|\rangle$ where $R' = R \oplus \mathtt{len}(\bar{K})$

Fig. 3. State transitions for the CK+ machine.

The [descend-λ] transition shifts a λ binding to the $\bar{K}$ register. When the control string in the CK+ machine is a λ abstraction, and that λ is the operator in an application term—indicated by an `arg` frame on top of the stack—the body of the λ becomes the control string; the top frame in the stack is updated to be a complete `bind` frame; and a new partial `mt` frame is pushed onto the stack.

The [descend-λ] instruction also updates the renaming environment which, as mentioned, is a list of numbers. There is one offset in the renaming environment for each `bind` continuation in the control stack and the offsets in the renaming environment appear in the same order as their corresponding `bind` continuations. When the machine descends into a λ expression, a new `bind` continuation is added to the top of the control stack so a new corresponding offset is also added to the front of the renaming environment. Since offsets are only added to the renaming environment when the machine goes under a λ, whenever a variable n (a lexical address) becomes the control string, its renaming offset is located at the n-th position in the renaming environment. A renaming offset keeps track of the relative position of a `bind` continuation since it was added to the control stack so a [descend-λ] instruction adds a 0 offset to the renaming environment.

When the control string is a variable n, the binding for n is accessed from the continuation stack by accessing the $(n+R(n)+1)$-th frame in the stack. The [lookup-arg] instruction moves the argument that is bound to the variable into the control string register. The `op` frame on top of the stack is updated to store all the frames inside the binding λ, in the same order that they appear in the stack. Using this strategy, the machine can "jump" back to this context after it is done evaluating the argument. For a term $(\lambda.E[n])\ M$, this is equivalent to evaluating M while saving E and then returning to the location of n after the argument M has been evaluated. Note that the [lookup-arg] transition does not perform substitution. The argument has been copied into the control string register, but it has also been removed from the continuation stack register.

When the frame on top of the stack is an `op`, it means the current control string is an argument in an application term. When that argument is a value, then a redex has been found and the value should be substituted for the variable that represents it. The [resume] rule is the only rule in figure 3 that performs a reduction in the sense of the by-need calculus. It is the implementation of the *deref* notion of reduction from the calculus. Specifically, the [resume] rule realizes this substitution by restoring the frames in the `op` frame back into the continuation stack as well as copying the value into a new `bind` frame. The result is nearly equivalent to the left hand side of the [lookup-arg] rule except that the argument has been evaluated and has been substituted for the variable.

Since the [resume] rule performs substitution, it must also update the renaming environment. Hence, the distance between V and its binding frame is added to every offset in the renaming environment R, as indicated by $R \oplus \texttt{len}(\bar{K})$. In other words, each offset in the environment is being incremented by the number of `bind` continuations that are added to the control stack.

In summary, the four rules of figure 3 represent intermediate partitions of the program into a subterm and an evaluation context before a partitioning of the

program into an evaluation context and a *deref* redex is found. As a result, the CK+ machine does not need to repartition the entire program on every machine step and is therefore more efficient than standard reduction. To complete the machine now, we must make it deal with answers.

4.7 Dealing with Answers

The CK+ machine described so far has no mechanism to identify whether a control string represents an answer. The by-need calculus, however, assumes that it is possible to distinguish answers from terms on several occasions, one of which is the completion of evaluation. To efficiently identify answers, the CK+ machine uses a fourth "answer" register. The CK+ machine identifies answers by searching the continuation stack for frames that are answer contexts. To distinguish answer contexts from evaluation contexts, we characterize answer contexts in figure 4. A final machine state has the form $\langle V, R, \| \, \|, \bar{A} \rangle$.

$S, T ::= \langle C, R, \bar{K} \rangle \mid \langle V, R, \|F,\ \ldots,\ K,\ \ldots\|, \bar{A} \rangle$	machine states
$F ::= (\texttt{bind}\ M\ R\ \texttt{mt})$	answer (complete) frame
$\bar{A} ::= \|\texttt{mt},\ F,\ \ldots\|$	answer stacks

Fig. 4. CK+ machine answer states.

When the control string is a value V and `mt` is the topmost stack frame, then some subterm in the program is an answer. In this situation, the `mt` frame in the stack is followed by an arbitrary number of F frames. The machine searches for the answer by shifting `mt` and F frames from the continuation stack register to the answer register. The machine continues searching until either a K frame is seen or the end of the continuation stack is reached. If the end of the continuation stack is reached, the entire term is an answer and evaluation is complete.

The presence of a K frame means an *assoc-L* or an *assoc-R* redex has been found. In order to implement these shifts, the CK+ machine requires four additional rules for handling answers, as shown in figure 5. The [ans-search1] rule shifts the `mt` frame to the answer register. The [ans-search2] rule shifts F frames to the answer register. The [assoc-L] rule and the [assoc-R] rule roughly correspond to the *assoc-L* and *assoc-R* notions of reduction in the calculus, respectively. The rules are optimized versions of corresponding notions of reduction in the calculus because the transition after the reduction is always known. The [assoc-L] machine rule performs the equivalent of an *assoc-L* reduction in the calculus, followed by a [descend-λ] machine transition. The [assoc-R] machine rule performs the equivalent of an *assoc-R* reduction in the calculus, followed by a [resume] machine transition.

$$\longmapsto_{ck+}$$

[ans-search1]

$$\langle V, R, \|\texttt{mt},\ K,\ \ldots\| \rangle \qquad \langle V, R, \|K,\ \ldots\|, \|\texttt{mt}\| \rangle$$

[ans-search2]

$$\langle V, R, \|F',\ K,\ \ldots\|, \|\texttt{mt},\ F,\ \ldots\| \rangle \qquad \langle V, R, \|K,\ \ldots\|, \|\texttt{mt},\ F,\ \ldots,\ F'\| \rangle$$

[assoc-L]

$$\langle \lambda.M', R, \|(\texttt{bind}\ M\ R'\ (\texttt{arg}\ N\ R''\ k)),\ K,\ \ldots\|, \|\texttt{mt},\ F,\ \ldots\| \rangle$$

$$\langle M', 0\!:\!R, \|\texttt{mt},\ (\texttt{bind}\ N\ R'''\ \texttt{mt}),\ F,\ \ldots,\ (\texttt{bind}\ M\ R'\ k),\ K,\ \ldots\| \rangle$$
$$\text{where } R''' = R'' \oplus \texttt{len}(\|F,\ \ldots\|) + 1$$

[assoc-R]

$$\left\langle V, R, \left\|(\texttt{bind}\ M\ R'\ (\texttt{op}\ \bar{K}\ k)),\ K, \ldots\right\|, \|\texttt{mt},\ F,\ \ldots\| \right\rangle$$

$$\left\langle V, R'', \bar{K}' +\!\!+ \|(\texttt{bind}\ V\ R\ \texttt{mt}),\ F, \ldots\ (\texttt{bind}\ M\ R'\ k),\ K,\ \ldots\| \right\rangle$$
$$\text{where } \bar{K}' = \bar{K} \oplus \texttt{len}(\|F,\ \ldots\|) + 1, \text{ and } R'' = R \oplus \texttt{len}(\bar{K}')$$

Fig. 5. Transitions of the CK+ machine that handle answer terms.

In figure 5, the function $\oplus$ has been extended to a family of functions defined over renaming environments, continuation stacks, and stack frames: $R \oplus x$ increments every offset in the renaming environment R by x and the function $\bar{K} \oplus x$ increments every offset in every renaming environment in every frame in $\bar{K}$ by x. The function $\texttt{len}(\|F,\ \ldots\|)$ returns the number of frames in $\|F,\ \ldots\|$. Maintaining the offsets in this manner is equivalent to obeying Garcia et al.'s "well-formedness" condition on machine states.

4.8 Correctness

Correctness means that the standard reduction machine and the CK+ machine define the same evaluator functions. Let us start with an appropriate definition for the CK+ machine:

$$\texttt{eval}_{ck+}(M) = \begin{cases} a, & \text{if } \langle M, (\ ), \|\texttt{mt}\| \rangle \longmapsto\!\!\!\!\to_{ck+} \left\langle V, R, \|\ \|, \bar{A} \right\rangle, \\ & \text{where } a = \varphi(\left\langle V, R, \|\ \|, \bar{A} \right\rangle) \\ \bot, & \text{if for all } \langle M, (\ ), \|\texttt{mt}\| \rangle \longmapsto\!\!\!\!\to_{ck+} S, S \longmapsto_{ck+} T \end{cases}$$

Recall that the function φ converts CK+ machine states to λ-calculus terms (figure 2). Here, φ has been extended to handle "answer" machine states:

$$\varphi(\left\langle M, R, \bar{K}, \bar{A} \right\rangle) = \bar{K}[\bar{A}[M \Leftarrow R]]$$

The desired theorem says that the two `eval` functions are equal.

Theorem 1. $\mathtt{eval}_{\mathit{need}} = \mathtt{eval}_{ck+}$.

To prove the theorem, we first establish some auxiliary lemmas on the totality of $\mathtt{eval}_{\mathbf{ck+}}$ and the relation between CK+ transitions and standard reduction transitions.

Lemma 2. *$\mathtt{eval}_{ck+}$ is a total function.*

Proof. The lemma is proved via a subject reduction argument. □

The central lemma uses φ to relate CK+ machine transitions to reductions.

Lemma 3. *For all CK+ machine states S and T, if $S \longmapsto_{ck+} T$, then either $\varphi(S) \longmapsto_{\mathit{need}} \varphi(T)$ or $\varphi(S) = \varphi(T)$.*

Proof. We proceed by case analysis on each machine transition, starting with [resume]. Assume

$$\begin{aligned}&\left\langle V, R, \left\|(\mathtt{op}\ \bar{K}\ k),\ K,\ \ldots\right\|\right\rangle \longmapsto_{ck+} \\ &\left\langle V, R \oplus \mathtt{len}(\bar{K}), \bar{K}{+\!+}\|(\mathtt{bind}\ V\ R\ k),\ K,\ \ldots\|\right\rangle\ ,\end{aligned}$$

then let

$$\begin{aligned}M_1 &= \varphi(\left\langle V, R, \left\|(\mathtt{op}\ \bar{K}\ k),\ K,\ \ldots\right\|\right\rangle) \\ &= \|K,\ \ldots\|\ [k[(\lambda.\bar{K}[\mathtt{len}(\bar{K}) - 1])\ (V{\Leftarrow}R)]] \\ M_2 &= \varphi(\left\langle V, R \oplus \mathtt{len}(\bar{K}), \bar{K}{+\!+}\|(\mathtt{bind}\ V\ R\ k),\ K,\ \ldots\|\right\rangle) \\ &= \|K,\ \ldots\|\ [k[(\lambda.\bar{K}[V{\Leftarrow}(R \oplus \mathtt{len}(\bar{K}))])\ (V{\Leftarrow}R)]]\ .\end{aligned}$$

Since M_1 is a standard *deref* redex, we have:

$$\begin{aligned}&\|K,\ \ldots\|\ [k[(\lambda.\bar{K}[\mathtt{len}(\bar{K}) - 1])\ (V{\Leftarrow}R)]] \longmapsto_{\mathbf{need}} \\ &\|K,\ \ldots\|\ [k[(\lambda.\bar{K}[(V{\Leftarrow}R)\uparrow_0^{\mathtt{len}(\bar{K})}])\ (V{\Leftarrow}R)]]\end{aligned}$$

To conclude that $M_1 \longmapsto_{\mathbf{need}} M_2$ by the *deref* notion of reduction, we need to show:

$$(V{\Leftarrow}R)\uparrow_0^{\mathtt{len}(\bar{K})} = V{\Leftarrow}(R \oplus \mathtt{len}(\bar{K}))$$

Lemma 4 proves the general case for this requirement. Therefore, we can conclude that $M_1 \longmapsto_{\mathbf{need}} M_2$. The proofs for [assoc-L] and [assoc-R] are similar.

As for the remaining instructions, they only shift subterms/contexts back and forth between registers, so the proof is a straightforward calculation. □

Lemma 4. *$\forall R, R_1, R_2$, where $R = R_1{+\!+}R_2$ and $m = \mathtt{len}(R_1)$:*

$$(M{\Leftarrow}R)\uparrow_m^x = M{\Leftarrow}(R_1{+\!+}(R_2 \oplus x))$$

Proof. By structural induction on M. □

Using lemma 3, the argument to prove our main theorem is straightforward.

Proof (of Theorem 1). We show $\texttt{eval}_{ck+}(M) = a \iff \texttt{eval}_{\mathbf{need}}(M) = a$.

The left-to-right direction follows from the observation that for all CK+ machine starting states S and final machine states S_{final}, if $S \longmapsto\!\!\!\!\rightarrow_{ck+} S_{\mathit{final}}$, then $M \longmapsto\!\!\!\!\rightarrow_{\mathbf{need}} a$, where $\varphi(S_{\mathit{final}}) = a$. This is proved using lemma 3 and induction on the length of the $\longmapsto\!\!\!\!\rightarrow_{ck+}$ sequence.

The other direction is proved by contradiction. Assume $\texttt{eval}_{\mathbf{need}}(M) = a \neq \bot$ and $\texttt{eval}_{ck+}(M) \neq a$. Since $\texttt{eval}_{ck+}$ is a total function, either:

1. $\langle M, (\,), \|\texttt{mt}\| \rangle \longmapsto\!\!\!\!\rightarrow_{ck+} S_{\mathit{final}}$, where $\varphi(S_{\mathit{final}}) \neq a$, or
2. the reduction of $\langle M, (\,), \|\texttt{mt}\| \rangle$ diverges.

It follows from the left-to-right direction of the theorem that, in the first case, $\texttt{eval}_{\mathbf{need}}(M) = \varphi(S_{\mathit{final}}) \neq a$, and in the second case, $\texttt{eval}_{\mathbf{need}}(M) = \bot$. However, $\texttt{eval}_{\mathbf{need}}(M) = a$ was assumed and $\texttt{eval}_{\mathbf{need}}$ is a total function, so a contradiction has been reached in both cases. Since none of the cases are possible, we conclude that if $\texttt{eval}_{\mathbf{need}}(M) = a$, then $\texttt{eval}_{ck+}(M) = a$. □

5 Stack Compacting

Because the by-need λ-calculus does not substitute the argument of a function call for all occurrences of the parameter at once, applications are never removed. In the CK+ machine, arguments accumulate on the stack and remain there forever. For a finite machine, an ever-growing stack is a problem. In this section, we explain how to compact the stack.

To implement a stack compaction algorithm in the CK+ machine, we introduce a separate SC machine which removes all unused stack bindings from a CK+ machine state. Based on the SC machine, the CK+ machine can be equipped with a non-deterministic [sc] transition:

$$\left\langle M, R, \bar{K} \right\rangle \longmapsto_{ck+} \left\langle M, R', \bar{K}' \right\rangle \quad \text{[sc]}$$
$$\text{where } \left\langle (\texttt{FV}\ M\ R\ 0), (M, R), \bar{K}, \|\ \| \right\rangle \longmapsto\!\!\!\!\rightarrow_{sc} \left\langle \mathcal{F}, (M, R'), \|\ \|, \bar{K}' \right\rangle$$

Figure 6 presents the SC machine. In this figure, FV refers to a family of functions that extracts the set of free variables from terms, stack frames, and continuation stacks. The function FV takes a term M, a renaming environment R and a variable m, and extracts free variables from M, where a free variable is defined to be all n such that $n + R(n) \geq m$. The function FV is similarly defined for stack frames and continuation stacks. In addition, $\mathcal{F}{-}{-}$ denotes the set obtained by decrementing every element in $\mathcal{F}$ by one. Finally, $\bar{K}@k$ represents a frame merged appropriately into a continuation stack. For example, $\|k',\ K,\ \ldots, (\texttt{bind}\ M\ R\ k'')\|\,@k = \|k',\ K,\ \ldots,\ (\texttt{bind}\ M\ R\ k'')@k\|$, where $(\texttt{bind}\ M\ R\ k'')@k = (\texttt{bind}\ M\ R\ k''@k)$, and so on, until finally $\texttt{mt}@k = k$.

Also in figure 6, $\uparrow\uparrow$ denotes a family of functions that adjusts the offsets in renaming environments to account for the fact that a λ has been removed

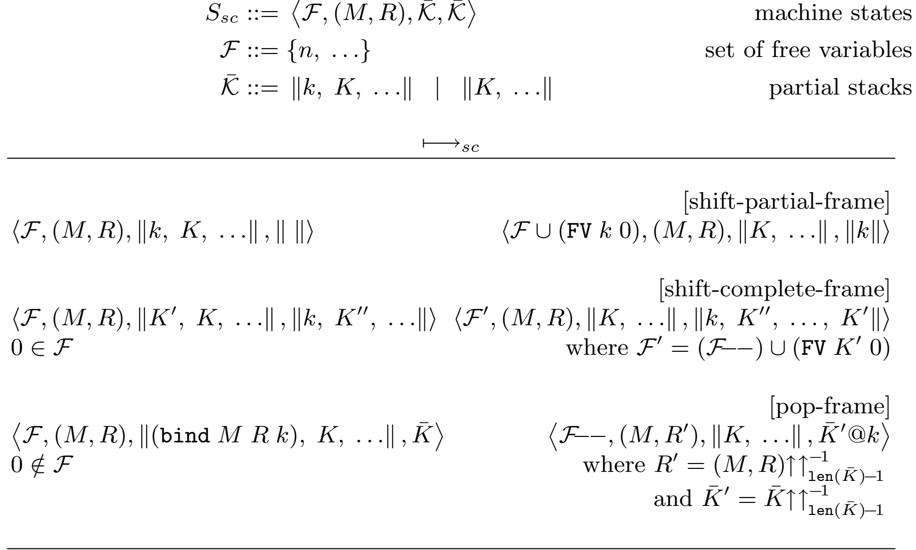

Fig. 6. The SC machine.

from the term. If a variable n refers to a `bind` stack frame that is deeper in the stack than the frame that is removed, then the offset for that variable needs to be decremented by one. A variable n refers to a `bind` that is deeper than the removed frame if $n + R(n)$ is greater than the depth of the removed frame. The $\uparrow\uparrow$ function can be applied to renaming environments directly or to continuation stacks or stack frames that contain renaming environments. We use the notation $(M, R)\uparrow\uparrow^x_\ell$ to mean that the offsets in R are incremented by x for all variables n in M where $n+R(n) > \ell$. The result of $(M, R)\uparrow\uparrow^x_\ell$ is a new renaming environment with the adjusted offsets. The notation $\bar{K}\uparrow\uparrow^x_\ell$ means that the offsets for all M and R pairs in the continuation stack $\bar{K}$ are adjusted. $\bar{K}\uparrow\uparrow^x_\ell$ evaluates to a new continuation stack that contains the adjusted renaming environments.

6 Related Work and Conclusion

The call-by-need calculus is due to Ariola et al. [2, 7, 8]. Garcia et al. [3] derive an abstract machine for Ariola and Felleisen's calculus and, in the process, uncover a correspondence between the by-need calculus and delimited control operations. Danvy et al. [9] derive a machine similar to Garcia et al. by applying "off-the-shelf" transformations to the by-need calculus. Danvy and Zerny's def-use chains also share similarities with our control stack structure [10].

Our paper has focused on the binding structure of call-by-need programs implied by Ariola and Felleisen's calculus. We have presented the CK+ ma-

chine, which restructures the control stack of Garcia et al.'s machine, and we have shown that lexical addresses can be used to directly access binding sites for variables in this dynamic control stack, a first in the history of programming languages. The use of lexical addresses has also simplified hygiene maintenance by eliminating the need for the set of "active variables" that is present in Garcia et al.'s machine states. In addition, we show how using indices in place of variables allows for simple maintenance of Garcia et al.'s "well-formed" machine states. Finally, we have presented a stack compaction algorithm, which is used in the CK+ machine to prevent stack overflow. The compaction algorithm used in this paper is a restriction of the more general garbage collection notion of reduction of Felleisen and Hieb [11] and is also reminiscent of Kelsey's work [12].

Acknowledgments. Thanks to the anonymous reviewers for their feedback and to Daniel Brown for inspiring discussions.

References

1. Plotkin, G.D.: Call-by-name, call-by-value and the λ-calculus. Theoretical Computer Science **1** (1975) 125–159
2. Ariola, Z.M., Felleisen, M.: The call-by-need lambda calculus. Journal of Functional Programming **7** (1997) 265–301
3. Garcia, R., Lumsdaine, A., Sabry, A.: Lazy evaluation and delimited control. In: Proceedings of the 36th Annual Symposium on Principles of Programming Languages, ACM (2009) 153–164
4. Barendregt, H.P.: The Lambda Calculus: Its Syntax and Semantics. North Holland (1981)
5. De Bruijn, N.G.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. Indagationes Mathematicae (1972) 381–392
6. Felleisen, M., Findler, R.B., Flatt, M.: Semantics Engineering with PLT Redex. MIT Press (2009)
7. Ariola, Z.M., Felleisen, M., Maraist, J., Odersky, M., Wadler, P.: The call-by-need lambda calculus. In: Proceedings of the 22nd Annual Symposium on Principles on Programming Languages. (1995) 233–246
8. Maraist, J., Odersky, M., Wadler, P.: The call-by-need lambda calculus. Journal of Functional Programming **8** (1998) 275–317
9. Danvy, O., Millikin, K., Munk, J., Zerny, I.: Defunctionalized interpreters for call-by-need evaluation. In Blume, M., Vidal, G., eds.: 10th International Symposium on Functional and Logic Programming. Lecture Notes in Computer Science, Springer (2010)
10. Danvy, O., Zerny, I.: Three syntactic theories for combinatory graph reduction. In Alpuente, M., ed.: 20th International Symposium on Logic-Based Program Synthesis and Transformation. (2010) Invited talk.
11. Felleisen, M., Hieb, R.: The revised report on the syntactic theories of sequential control and state. Theoretical Computer Science **103** (1992) 235–271
12. Kelsey, R.: Tail-recursive stack disciplines for an interpreter. Technical Report NU-CCS-93-03, Northeastern University (1993)