Lazy Evaluation and Delimited Control

Ronald Garcia *
Rice University
ronald.garcia@rice.edu

Andrew Lumsdaine Amr Sabry
Indiana University
{lums,sabry}@indiana.edu

Abstract

The call-by-need lambda calculus provides an equational framework for reasoning syntactically about lazy evaluation. This paper examines its operational characteristics.

By a series of reasoning steps, we systematically unpack the standard-order reduction relation of the calculus and discover a novel abstract machine definition which, like the calculus, goes "under lambdas." We prove that machine evaluation is equivalent to standard-order evaluation.

Unlike traditional abstract machines, delimited control plays a significant role in the machine's behavior. In particular, the machine replaces the manipulation of a heap using store-based effects with disciplined management of the evaluation stack using control-based effects. In short, state is replaced with control.

To further articulate this observation, we present a simulation of call-by-need in a call-by-value language using delimited control operations.

Categories and Subject Descriptors D.3.1 [*Software*]: Programming Languages—Formal Definitions and Theory

General Terms Languages, Theory

Keywords call-by-need, reduction semantics, abstract machines, delimited continuations, lambda calculus

1. Introduction

From early on, the connections between lazy evaluation (Friedman and Wise 1976; Henderson and Morris 1976) and control operations seemed strong. One of these seminal papers on lazy evaluation (Henderson and Morris 1976) advocates laziness for its coroutine-like behavior. Specifically, it motivates lazy evaluation with a solution to the *same fringe* problem: how to determine if two trees share the same fringe without first flattening each tree and then comparing the resulting lists. A successful solution to the problem traverses just enough of the two trees to tell that they do not match. The same fringe problem is also addressed in Sussman and Steele's original exposition of the Scheme programming language (Sussman and Steele Jr. 1998). One of their solutions uses a continuation passing-style representation of coroutines.

* Work conducted at Indiana University.

POPL'09, January 18–24, 2009, Savannah, Georgia, USA.

Same fringe is not the only programming problem that can be solved using either lazy evaluation or continuations. For instance, lazy streams and continuations are also used to implement and reason about backtracking (Wand and Vaillancourt 2004; Kiselyov et al. 2005). Strong parallels in the literature have long suggested that lazy evaluation elegantly embodies a stylized use of coroutines. Indeed, we formalize this connection.

Call-by-need evaluation combines the equational reasoning capabilities of call-by-name with a more efficient implementation technology that systematically shares the results of some computations. However, call-by-need's evaluation strategy makes it difficult to reason about the operational behavior and space usage of programs. To facilitate reasoning, semantic models (Launchbury 1993; Sestoft 1997; Friedman et al. 2007), simulations (Okasaki et al. 1994), and tracing tools (Gibbons and Wansbrough 1996) for call-by-need evaluation have been developed. Many of these artifacts use an explicit store or store-based side-effects (Wang 1990) to represent values that are shared between parts of a program. Stores, being amorphous structures, make it difficult to establish program properties or analyze program execution.

The call-by-need lambda calculus was introduced by Ariola et al. (1995) as an alternative to store-based formalizations of lazy evaluation. It is an equational framework for reasoning about call-by-need programs and languages. Following Plotkin (1975), the authors present a calculus and prove a standardization theorem that links the calculus to a complete and deterministic (i.e. standard order) reduction strategy. The calculus can be used to formally justify transformations, particularly compiler optimizations, because any terms it proves equal are also contextually equivalent under call-by-need evaluation.

Call-by-need calculi were investigated by two groups (Maraist et al. 1998; Ariola and Felleisen 1997). The resulting two calculi are quite similar but their subtle differences yield trade-offs that are discussed in the respective papers.

One notable feature of Ariola and Felleisen's calculus is its use of evaluation contexts within the notions of reduction. It has been observed that evaluation contexts correspond to continuations in some presentations of language semantics (Felleisen and Friedman 1986; Biernacka and Danvy 2007). It's not obvious, however, that the evaluation contexts in this calculus have anything to do with control operations: they're used to model variable references and demand-driven evaluation, not first-class continuations.

This paper exposes how Ariola and Felleisen's call-by-need evaluation relates to continuations. By systematically unpacking the standard-order reduction relation of the calculus, we discover a novel abstract machine that models call-by-need style laziness and sharing without using a store. Instead, the machine manipulates its evaluation context in a manner that corresponds to a stylized use of delimited control operations. The machine's behavior reveals a connection between control operations and laziness that was present but hidden in the reduction semantics.

To directly interpret this connection in the terminology of delimited control, we construct a simulation of call-by-need terms in the call-by-value language of Dybvig et al. (2007), which provides a general framework for delimited continuations with first-class generative prompts.

Our concrete specifications of the relationship between call-by-need and delimited control firmly establish how lazy evaluation relates to continuations and other control-oriented language constructs and effects. Implementations of both the machine and the simulation are available at the following url:
`http://osl.iu.edu/~garcia/call-by-need.tgz`.

2. The Call-by-need Lambda Calculus

The remainder of this paper examines Ariola and Felleisen's formalization of call-by-need (Ariola and Felleisen 1997). The terms of the calculus are standard:

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

The calculus highlights two particular subsets of terms. First, as is typical, lambda terms are considered values:

$$v \quad ::= \quad \lambda x.t$$

The calculus also distinguishes a subset of terms called *answers*:

$$a \quad ::= \quad v \mid (\lambda x.a)\ t$$

Answers are a syntactic representation of (partial) closures. Notice that an answer takes the form of a lambda term nested inside some applications. The surrounding applications simulate environment bindings for free variables in the nested lambda term. This representation makes it possible for the calculus to explicitly account for variable binding, in particular to syntactically model how call-by-need evaluation shares lazily computed values.

Following the style of Felleisen and Hieb (1992), the calculus uses evaluation contexts to indicate those locations in an expression that may be subject to reduction:

$$E \quad ::= \quad \Box \mid E\ t \mid (\lambda x.E[x])\ E \mid (\lambda x.E)\ t$$

Call-by-need contexts are unusual in that they are not merely inductively defined. The third context production, $(\lambda x.E[x])\ E$, encodes a side condition and expands as follows: if E is a context and t is a term, then if there exists a context E' that does not capture x and $t \equiv E'[x]$, then $(\lambda x.t)\ E$ is also a context.

The calculus has three notions of reduction:

$$\begin{array}{lll}
(\lambda x.E[x])\ v & \rightarrow_{\text{need}} & (\lambda x.E[v])\ v \\
(\lambda x.a)\ t_1\ t_2 & \rightarrow_{\text{need}} & (\lambda x.a\ t_2)\ t_1 \\
(\lambda x_1.E[x_1])\ ((\lambda x_2.a)\ t_1) & \rightarrow_{\text{need}} & (\lambda x_2.(\lambda x_1.E[x_1])\ a)\ t_1
\end{array}$$

As popularized by Barendregt (1981), each reduction assumes a hygiene convention. When combined with the evaluation contexts, the notions of reduction yield a deterministic standard order reduction relation ($\longmapsto_{sr}$) and its reflexive-transitive closure ($\longmapsto\!\!\!\!\!\rightarrow_{sr}$).

Definition 1. *$t_1 \longmapsto_{sr} t_2$ if and only if $t_1 \equiv E[t_r]$, $t_2 \equiv E[t_c]$ and $t_r \rightarrow_{\text{need}} t_c$.*

Standard order reduction provides us an effective specification of call-by-need evaluation: if t is a program (i.e. closed term), then t call-by-need evaluates to an answer if and only if $t \longmapsto\!\!\!\!\!\rightarrow_{sr} a$ for some answer a.

3. From Reduction Semantics to Machine Semantics

Some reduction semantics have been shown to correspond directly to abstract machine semantics, thereby establishing the equivalence of a reducer and a tail-recursive abstract machine implementation (Felleisen and Friedman 1986; Felleisen and Flatt 2002). In particular, Danvy and Nielsen (2004) derive abstract machine semantics directly from reduction semantics using provably correct transformations. Proceeding from a reduction semantics for call-by-need to an accessible and informative tail-recursive abstract machine semantics is not, however, as straightforward as for call-by-name or call-by-value.

Call-by-need redexes require more computational effort to recognize than either call-by-name or call-by-value. For instance, given a term t, only a fixed number of terminal operations are required to detect whether t is a call-by-name redex: one to check if the term is an application, one to access the operator position, and one to check if the operator is an abstraction.

Contrast this with the call-by-need redex $(\lambda x.E[x])\ ((\lambda y.a)\ t)$. Given a call-by-need term t_x, testing whether it matches this redex form requires an unpredictable number of operations: check if t_x is an application; check if its operator position is a lambda abstraction; check, in an unpredictable number of steps, if the operator's body can be decomposed into $E[x]$, where x is both free in E and bound by the operator; and finally check, in an unpredictable number of steps, if the operand has the inductive structure of an answer.

To make matters worse, some terms can be decomposed into the form $E[x]$ in more than one way. For instance, consider the term $(\lambda x.(\lambda y.y)\ x)$. It can be decomposed as both $(\lambda x.E_1[y])$ and $(\lambda x.E_2[x])$ where $E_1 \equiv (\lambda y.\Box)\ x$ and $E_2 \equiv (\lambda y.y)\ \Box$. As such, a recursive procedure for decomposing a term cannot stop at the first variable it finds: it must be able to backtrack in a way that guarantees it will find the right decomposition $E[x]$—and in turn the right redex—if there is one.

A straightforward call-by-need reducer implementation naïvely decomposes a term into a context and a redex. Any application could be one of three different redexes, each of which is nontrivial to detect, so whenever the decompose function detects an application, it sequentially applies several recursive predicates to the term in hopes of detecting a redex. If the term is a redex, it returns; if not, it recursively decomposes the operator position. In our experience, this implementation strategy sheds no light on how to produce an understandable tail-recursive implementation.

Recall that one of the evaluation contexts has the form $(\lambda x.E)t$. This means that redex evaluation can occur "under binders" (Moggi and Sabry 2004; Kameyama et al. 2008). All three call-by-need notions of reduction shuffle lambda abstractions about in unusual ways. Furthermore, while reducing a recursive routine, a call-by-need evaluator may end up performing reductions under multiple copies of the same lambda abstraction. Call-by-name and call-by-value evaluators can address hygiene concerns by using environments and closures, but a call-by-need evaluator must prevent its evaluation context from incorrectly capturing free variable references. Any evaluator that goes under lambdas must pay particular attention to hygiene (Xi 1997).

3.1 Towards an Abstract Machine

To find call-by-need redexes tail-recursively, we apply an insight from the CK abstract machine (Felleisen and Friedman 1986; Felleisen and Flatt 2002). The CK machine implements an evaluation strategy for call-by-value based on a reduction semantics using the (inside-out) evaluation contexts $\Box$, $E[\Box\ t]$ and $E[(\lambda x.t)\,\Box]$. To find a redex, the machine iteratively examines the outermost constructor of a term and uses the evaluation context to remember what has been discovered.

To illustrate this in action, we walk through an example. Consider the program $(\lambda x.x)\ \lambda y.y$. Evaluation begins with configuration $\langle [\,], \underline{(\lambda x.x)\ \lambda y.y}\rangle$. Underlining indicates subterms that the machine knows nothing about; at the beginning of evaluation, it knows nothing about the entire term. On the first step of the reduction, the machine detects that the term is an application $\underline{(\lambda x.x)}\ \underline{\lambda y.y}$. To examine the term further, the machine must move its focus to either the operator or operand of this application. Since the machine is tail recursive, it must also push an evaluation context to store the as-yet uncovered structure of the term. The only context it can reliably push at this point is $[\Box\ \underline{\lambda y.y}]$: it cannot push $[\underline{(\lambda x.x)}\ \Box]$ because it has not yet discovered that the operator position is a value. So the machine pushes the $[\Box\ \underline{\lambda y.y}]$ context, which serves as a reminder that it is focused on the operator of an application.

On the second step of the reduction, the machine detects that the operator is an abstraction $\lambda x.\underline{x}$, and observes that the innermost context is $[\Box\ \underline{\lambda y.y}]$. In response, the machine pops the context, focuses on $\underline{\lambda y.y}$, and pushes the context $[(\lambda x.\underline{x})\ \Box]$, since the operator is now known to be a value. This context serves as a reminder that evaluation is currently focused on the operand of an application that can be reduced once that operand becomes a value.

On the third step, the machine detects the abstraction $(\lambda y.\underline{y})$, and remembers that the innermost context is $[(\lambda x.\underline{x})\ \Box]$. At this point, the machine has deduced enough information to recognize the redex $(\lambda x.\underline{x})\ \lambda y.\underline{y}$. This example illustrates how the CK machine uses a depth-first left-to-right search strategy to detect call-by-value redexes.

Now consider the same term under call-by-need using a similar strategy. As with call-by-value, the top-level application can be detected, the operand can be pushed onto the evaluation context, and the operator can be exposed as the abstraction $\lambda x.\underline{x}$. At this point behavior must diverge from call-by-value because the body of the abstraction is still unknown and call-by-need does not have $[(\lambda x.t)\Box]$ contexts for arbitrary t. However, call-by-need does have contexts of the form $[(\lambda x.\Box)\ \underline{\lambda y.y}]$. Therefore, it is possible to proceed under the first lambda abstraction, push the context, and focus on $\underline{x}$.

The term is exposed as a variable x, which combines with the context $[(\lambda x.\Box)\ \underline{\lambda y.y}]$ to form the term $(\lambda x.E[x])\ \ \underline{\lambda y.y}$ (where $E \equiv \Box$). At this point, enough information has been uncovered to push the context $[(\lambda x.\Box[x])\ \Box]$ and focus on $\underline{\lambda y.y}$. The abstraction $\lambda y.\underline{y}$ is recognized, and with that a call-by-need redex $(\lambda x.\Box[x])\ \lambda y.\underline{y}$ has been found.

Success with this example suggests a promising strategy for implementing call-by-need reduction tail-recursively.

3.2 An Initial Abstract Machine

In this section, we elaborate the above search strategy into a simple but inefficient tail-recursive abstract machine. We present it without proof and then by a series of correct transformations we derive an efficient machine that we prove correct.

This abstract machine uses the same terms, values, and answers as the calculus. However, it introduces two alternate notions. First, the machine uses a more versatile representation of evaluation contexts. As observed in Danvy and Nielsen (2004), evaluation contexts can be mathematically specified in more than one way. For optimal flexibility, we define evaluation contexts as lists of frames, where the empty list $[\ ]$ and single-frame lists $[f]$ are our simple units, and the operator $\circ$ stands for list concatenation.

$$\begin{array}{lcl} f & ::= & \Box\ t \mid (\kappa x.E)\ \Box \mid (\lambda x.\Box)\ t \\ E & ::= & [\,] \mid [f] \circ E \mid E \circ [f] \\ & & \text{where } E \circ [\,] = [\,] \circ E = E \\ & & \text{and } E_1 \circ (E_2 \circ E_3) = (E_1 \circ E_2) \circ E_3 \end{array}$$

When two contexts are composed, the second context is plugged into the hole of the first context: for example $[\Box\ t_2] \circ [\Box\ t_1] = [(\Box\ t_1)\ t_2]$.

We call the frame $[(\lambda x.\Box)\ t]$ a *binder* frame. It represents a variable binding in the context. It can be read as $[\text{let}\, x = t \,\text{in}\, \Box]$, but we use the former notation to emphasize that call-by-need evaluation proceeds under lambdas. This observation motivates our analysis of hygiene in Section 4.5.

We call the frame $[(\kappa x.E)\ \Box]$ a *cont* frame, in reference to continuations. The construction $(\kappa x.E)$ is called a cont and replaces the metalinguistic term notation $(\lambda x.E[x])$ from the calculus. We use a different notation for conts than lambda terms to indicate that in the machine conts are distinct from terms (they are of type `Cont` rather than type `Term` in an implementation). Conts indicate nontrivial structural knowledge that the machine retains as it searches for a redex. This distinction matters when we establish continuation semantics for machine states. As we shall see, a cont frame represents a suspended variable reference.

Finally we call the frame $[\Box\ t]$ an *operand* frame, and it represents a term waiting for an abstraction.

The abstract machine also introduces a notion of *redexes*:

$$r \quad ::= \quad a\ t \mid (\kappa x.E)\ a$$

Redexes are distinguished from terms in the machine, meaning that in an implementation, the type `Redex` is distinct from the type `Term`. This distinction suggests that conts $(\kappa x.E)$ are neither terms nor first-class entities in the call-by-need language: they only appear in evaluation contexts and in redexes.

The transition rules for the machine are staged into four distinct groups: refocus, rebuild, need, and reduce. Each machine configuration can be related to a term in the language of the calculus. The *refocus* rules examine the current term and push as many operand frames $[\Box\ t]$ as possible. A refocus configuration $\langle E, t\rangle_f$ represents the term $E[t]$.

$$\boxed{\langle E, t\rangle_f} \quad \text{(Refocus)}$$

$$\begin{array}{rcl} \langle E, x\rangle_f & \mapsto & \langle E, [\,], x\rangle_n \\ \langle E, \lambda x.t\rangle_f & \mapsto & \langle E, \lambda x.t\rangle_b \\ \langle E, t_1\ t_2\rangle_f & \mapsto & \langle E \circ [\Box\ t_2], t_1\rangle_f \end{array}$$

Upon reaching a variable, refocus transitions to the need rules; upon reaching a lambda abstraction, it transitions to the rebuild rules.

The *rebuild* rules search up into the context surrounding an answer for the next applicable redex. A rebuild configuration $\langle E, a\rangle_b$ represents the term $E[a]$.

$$\boxed{\langle E, a\rangle_b} \quad \text{(Rebuild)}$$

$$\begin{array}{rcl} \langle [\,], a\rangle_b & \mapsto & a \\ \langle E \circ [\Box\ t_1], a\rangle_b & \mapsto & \langle E, a\ t_1\rangle_d \\ \langle E \circ [(\lambda x.\Box)\ t_1], a\rangle_b & \mapsto & \langle E, (\lambda x.a)\ t_1\rangle_b \\ \langle E_1 \circ [(\kappa x.E_2)\ \Box], a\rangle_b & \mapsto & \langle E_1, (\kappa x.E_2)\ a\rangle_d \end{array}$$

These rules examine the current context and proceed to build a maximal answer-shaped term, progressively wrapping binder frames around the current answer. If the entire context is consumed then evaluation has completed and the entire program is an answer. Upon reaching an operand or cont frame, a redex has been found, and rebuild transitions to the reduce rules. These rules resemble the $\mathrm{refocus_{aux}}$ rules of Danvy and Nielsen (2004).

The *need* rules also examine the context, but they search for the binder frame that corresponds to the variable under focus. A need configuration $\langle E_1, E_2, x\rangle_n$ represents the term $E_1[E_2[x]]$.

$$\boxed{\langle E, E, x\rangle_n} \quad \text{(Need)}$$
$$\begin{array}{rcl}
\langle E_1 \circ [(\lambda x.\square)\ t], E_2, x\rangle_n & \mapsto & \langle E_1 \circ [(\kappa x.E_2)\ \square], t\rangle_f \\
\langle E_1 \circ [f], E_2, x\rangle_n & \mapsto & \langle E_1, [f] \circ E_2, x\rangle_n \\
\multicolumn{3}{c}{\text{where, } [f] \not\equiv [(\lambda x.\square)\ t]}
\end{array}$$

Since input programs are closed, the associated binder must be somewhere in the context. Upon finding the right binder frame, a cont frame $[(\kappa x.E)\ \square]$ is pushed onto the context and evaluation proceeds to refocus on the operand from the associated binder frame.

The *reduce* rules simulate the notions of reduction from the calculus. A reduce configuration $\langle E, r\rangle_d$ represents the term $E[r]$ where a cont $\kappa x.E$ represents the term $\lambda x.E[x]$.

$$\boxed{\langle E, r\rangle_d} \quad \text{(Reduce)}$$
$$\begin{array}{rcl}
\langle E_1, (\kappa x.E_2)\ v\rangle_d & \mapsto & \langle E_1, (\lambda x.E_2[v])\ v\rangle_f \\
\langle E_1, (\kappa x_1.E_2)\ ((\lambda x_2.a)\ t)\rangle_d \mapsto & & \\
 & & \langle E_1, (\lambda x_2.(\lambda x_1.E_2[x_1])\ a)\ t\rangle_f \\
\langle E, (\lambda x.a)\ t_1\ t_2\rangle_d & \mapsto & \langle E, (\lambda x.a\ t_2)\ t_1\rangle_f \\
\langle E, (\lambda x.t_1)\ t_2\rangle_d & \mapsto & \langle E \circ [(\lambda x.\square)\ t_2], t_1\rangle_f
\end{array}$$

Each of the first two reduce rules transforms a cont into a lambda abstraction by plugging its context with a term and abstracting its variable. As such, each reduce rule transforms a redex into a pure term of the calculus and transitions to a refocus configuration, which searches for the next redex.

The reduce rules also handle terms of the form $(\lambda x.t_1)\ t_2$, even though such terms are not call-by-need redexes. Including this rule gives the set of redexes greater uniformity: all terms of the form $a\ t$ are redexes, just like the terms of the form $(\kappa x.E)\ a$. This symmetry is not exhibited in the call-by-need calculus. However, Ariola and Felleisen (1997) defines and uses an auxiliary *let calculus* that adds the reduction

$$(\lambda x.t_1)\ t_2 \to_{\mathrm{need}} \mathrm{let}\ x = t_2\ \mathrm{in}\ t_1$$

to the calculus and defines the other reductions in terms of the let expressions. The fourth reduce rule corresponds to this reduction rule. However, our presentation shows that an auxiliary let term, though compatible with this model, is not needed to specify call-by-need: the syntax of pure calculus terms suffices. Furthermore, the reduce rules are improved in the next section so that all reduce rules change their representative terms nontrivially.

Machine evaluation of a program t begins with the refocus configuration $\langle [\,], t\rangle_f$ and terminates if it arrives at an answer a. Its behavior in between can be summarized as follows: search downwards until a value or variable reference is reached. If a variable reference is reached, store a cont in the context to remember the variable reference and proceed to evaluate its binding. If an abstraction is reached, accumulate an answer up to the innermost redex, or the top of the evaluation context if none is found. In short, the machine performs a depth-first, left-to-right traversal in search of a call-by-need redex. Along the way it uses the evaluation context to store and retrieve information about program structure, particularly the location of variable bindings (using binder frames) and variable references (using cont frames). The refocus, rebuild, and need rules leave the term representation of their configurations unchanged (e.g. if $\langle E_1, t_1\rangle_f \mapsto \langle E_2, t_2\rangle_f$ then $E_1[t_1] \equiv E_2[t_2]$), and the reduce rules embody the notions of reduction from the calculus.

Our strategy for producing this machine generalizes the strategy of Danvy and Nielsen (2004), which does not account for terms like the call-by-need variable references, which are neither redexes nor values, yet cannot be decomposed.

The following partial trace demonstrates how this machine discovers the first redex for our running example $(\lambda x.x)\ \lambda y.y$:

$$\begin{gathered}
\left\langle [\,], \underline{(\lambda x.x)\ \lambda y.y}\right\rangle_f \mapsto \left\langle [\square\ \underline{\lambda y.y}], \underline{\lambda x.x}\right\rangle_f \mapsto \\
\left\langle [\square\ \underline{\lambda y.y}], \lambda x.\underline{x}\right\rangle_b \mapsto \left\langle [\,], (\lambda x.\underline{x})\ \underline{\lambda y.y}\right\rangle_d \mapsto \\
\left\langle [(\lambda x.\square)\ \underline{\lambda y.y}], \underline{x}\right\rangle_f \mapsto \left\langle [(\lambda x.\square)\ \underline{\lambda y.y}], [\,], x\right\rangle_n \mapsto \\
\left\langle [(\kappa x.[\,])\ \square], \underline{\lambda y.y}\right\rangle_f \mapsto \left\langle [(\kappa x.[\,])\ \square], \lambda y.\underline{y}\right\rangle_b \mapsto \\
\left\langle [\,], (\kappa x.[\,])\ \lambda y.\underline{y}\right\rangle_d
\end{gathered}$$

4. Refining the Machine

In this section we study the behavior of the abstract machine and make some improvements based on our observations. These changes lead us from the initial machine above to our final machine specification.

4.1 Grabbing and Pushing Conts

The need rules linearly search the nearest context for a binder frame that matches the variable under question. This process can be specified as one step:

$$\begin{gathered}
\langle E_1 \circ [(\lambda x.\square)\ t] \circ E_2, x\rangle_n \mapsto \langle E_1 \circ [(\kappa x.E_2)\ \square], t\rangle_f \\
\text{where } [(\lambda x.\square)\ t] \notin E_2
\end{gathered}$$

This evaluation step accumulates a segment of the current evaluation context and stores it. In general, abstract machines that model control operators represent control capture in a similar manner. In this particular case, only part of the evaluation context is captured, and the amount of context captured depends on the dynamic location in the context of a certain frame. As such, the need rules seem to perform some kind of *delimited* control capture. This analogy becomes stronger upon further analysis of the first reduce rule from Section 3.2. The machine uses its structural knowledge of $\kappa x.E$ to construct the abstraction $\lambda x.E[v]$. However, the resulting machine configuration no longer retains any of the structure that had previously been discovered. Recall our example execution trace from Section 3.2. The machine reduces the redex found at the end of that trace as follows:

$$\left\langle [\,], (\kappa x.[\,])\ \lambda y.\underline{y}\right\rangle_d \mapsto \langle [\,], \underline{(\lambda x.\lambda y.y)\ \lambda y.y}\rangle_f$$

By returning to refocus following the reduction, the machine loses all structural knowledge of the term. To continue execution, it must examine the structure of the contractum from scratch. Fortunately, the evaluator can be safely improved so that it retains knowledge of the contractum's structure:

Proposition 1.

$$\langle E_1, (\lambda x.E_2[v])\ v\rangle_f \longmapsto\!\!\!\!\!\rightarrow \langle E_1 \circ [(\lambda x.\square)\ v] \circ E_2, v\rangle_b$$

Proof. Corollary of $\langle E_1, E_2[v]\rangle_f \longmapsto\!\!\!\!\!\rightarrow \langle E_1 \circ E_2, v\rangle_b$, which is proven by induction on E_2. □

This proposition justifies replacing the first reduce rule with one that pushes the evaluation context embedded in the cont and proceeds to rebuild an answer:

$$\langle E_1, (\kappa x.E_2)\ v\rangle_d \mapsto \langle E_1 \circ [(\lambda x.\square)\ v] \circ E_2, v\rangle_b$$

This short-circuit rule extends the current evaluation context with a binder frame and the context E_2 that was inside the cont. The rule is suggestive of delimited control because machine models of control operators generally represent the reinstatement of delimited continuations by extending the current context with a piece of captured evaluation context. Of more immediate interest, though, is how reduction of our example now proceeds:

$$\left\langle [\,], (\kappa x.[\,])\ \lambda y.\underline{y} \right\rangle_d \mapsto \left\langle [(\lambda x.\Box)\ (\lambda y.\underline{y})], \lambda y.\underline{y} \right\rangle_b$$

All knowledge of the contractum's structure is retained, though much of it is now stored in the evaluation context.

4.2 Shifting Binder Frames

The second and third reduce rules from Section 3.2 also discard structural information. Specifically, they both transition to the forgetful refocus rule. However their information can be preserved.

Proposition 2.

$$\langle E, (\lambda x.a\ t_2)\ t_1 \rangle_f \longmapsto\!\!\!\!\rightarrow \langle E \circ [(\lambda x.\Box)\ t_1], a\ t_2 \rangle_d\,.$$

Proof. Corollary of $\langle E_1, a \rangle_f \longmapsto\!\!\!\!\rightarrow \langle E_1, a \rangle_b$, which is proven by induction on a. ☐

Proposition 3. *If E_2 does not capture x_1 (Section 4.5), then*

$$\langle E_1, (\lambda x_2.(\lambda x_1.E_2[x_1])\ a)\ t \rangle_f \longmapsto\!\!\!\!\rightarrow \langle E_1 \circ [(\lambda x_2.\Box)\ t], (\kappa x_1.E_2)\ a \rangle_d\,.$$

Proof. Corollary of $\langle E_1, a \rangle_f \longmapsto\!\!\!\!\rightarrow \langle E_1, a \rangle_b$ and $\langle E_1, (\lambda x_1.E_2[x_1])\ t \rangle_f \longmapsto\!\!\!\!\rightarrow \langle E_1 \circ [(\kappa x_1.E_2)\ \Box], t \rangle_f$, which is proven by case analysis and induction on E_2. ☐

These propositions justify short-circuiting the respective evaluation rules. The new rules improve the behavior of the abstract machine.

$$\langle E_1, (\kappa x_1.E_2)\ ((\lambda x_2.a)\ t) \rangle_d \mapsto \langle E_1 \circ [(\lambda x_2.\Box)\ t], (\kappa x_1.E_2)\ a \rangle_d$$

$$\langle E, (\lambda x.a)\ t_1\ t_2 \rangle_d \mapsto \langle E \circ [(\lambda x.\Box)\ t_1], a\ t_2 \rangle_d$$

By fast-forwarding to reduce, the rules retain the discovered term structure and thereby avoid retracing the same terms again.

4.3 Answer = Binders × Value

The transition rules repeatedly create binder frames out of terms and reabsorb those frames into answers. In this section we simplify this protocol. We distinguish answers from terms by providing them a separate representation:

$$\boxed{a \quad ::= \quad \llbracket E, v \rrbracket, \quad \text{where } E = \overline{[(\lambda x_i.\Box)\ t_i]}}$$

An answer is now represented as a tuple containing the nested lambda abstraction and the binder frames that are wrapped around them in the old presentation. This presentation bears strong similarity to calculi with explicit substitutions (Abadi et al. 1991), where each binder frame $[(\lambda x.\Box)\ t]$ corresponds to a substitution $[t/x]$. An answer can be seen as a lambda term nested inside a sequence of explicit substitutions, $v\overline{[t_i/x_i]}$.

The rebuild rules could be reformulated as a three place configuration, $\langle E, E, v \rangle_b$, but instead we immediately apply the same improvement that we applied to the need rules in Section 4.1. For instance, the new transition rule for rebuilding to a cont frame is:

$$\langle E_1 \circ [(\kappa x.E_2)\ \Box] \circ E_3, v \rangle_b \mapsto \langle E_1, (\kappa x.E_2)\ \llbracket E_3, v \rrbracket \rangle_d$$
$$\text{where } E_3 = \overline{[(\lambda x_i.\Box)\ t_i]}$$

Returning to our running example, reduction from its most recent state (at the end of Section 4.1) transitions to a final answer, signaling the end of execution:

$$\left\langle [(\lambda x.\Box)\ \lambda y.\underline{y}], \lambda y.\underline{y} \right\rangle_b \mapsto \left\langle \llbracket [(\lambda x.\Box)\ \lambda y.\underline{y}], \lambda y.\underline{y} \rrbracket \right\rangle$$

4.4 Aggregate Reduction

Now that answers explicitly carry their binders in aggregate, the reduce rules can be substantially consolidated. Currently, the second and third reduce rules iteratively remove the top binder frame from an answer and push it onto the evaluation context. This process repeats until the answer is just a lambda abstraction. At that point, the second and third reduce rules defer to the first and fourth reduce rules respectively. This corresponds exactly with standard-order reduction (cf. Definition 1):

Proposition 4.

$$E[((\lambda x_n. \ldots ((\lambda x_1.((\lambda x_0.v)\ t_0))\ t_1)\ \ldots)\ t_n)\ t] \longmapsto\!\!\!\!\rightarrow_{sr}$$
$$E[((\lambda x_n. \ldots ((\lambda x_1.((\lambda x_0.v\ t)\ t_0))\ t_1)\ \ldots)\ t_n)].$$

$$E[(\lambda x.E[x])\ ((\lambda x_n. \ldots ((\lambda x_1.((\lambda x_0.v)\ t_0))\ t_1)\ \ldots)\ t_n)] \longmapsto\!\!\!\!\rightarrow_{sr}$$
$$E[((\lambda x_n. \ldots ((\lambda x_1.((\lambda x_0.(\lambda x.E[x])\ v)\ t_0))\ t_1)\ \ldots)\ t_n)].$$

Proof. By induction on the structure of the answer term, using the unique decomposition lemma of Ariola and Felleisen (1997). ☐

Using the new answer representation, each pair of associated reduce rules can be merged into one omnibus rule that moves all the binder frames at once and simultaneously performs a reduction using the underlying value.

$$\langle E_1, (\kappa x.E_2)\ \llbracket E_3, v \rrbracket \rangle_d \mapsto \langle E_1 \circ E_3 \circ [(\lambda x.\Box)\ v] \circ E_2, v \rangle_b$$
$$\langle E_1, \llbracket E_2, (\lambda x.t_1) \rrbracket\ t_2 \rangle_d \mapsto \langle E_1 \circ E_2 \circ [(\lambda x.\Box)\ t_2], t_1 \rangle_f$$

As a result of these transformations, both conts and answers contain evaluation contexts. Furthermore, conts and answers are not terms of the calculus, and the machine never reverts a cont or answer to a term. The rules that create them, rebuild for answers and need for conts, capture part of the evaluation context, and the rules that consume them, the reduce rules, reinstate the captured contexts.

4.5 Variable Hygiene

Presentations of calculi often invoke a hygiene convention and from then on pay little attention to bound or free variables. In this manner, calculi do not commit to any of the numerous ways that hygiene can be enforced. Many abstract machines, however, use environments or explicit sources of fresh names to guarantee hygiene and thereby provide a closer correspondence to concrete implementations. In this section, we augment the call-by-need machine with simple measures to enforce hygiene.

Our primary hygiene concerns are that evaluation occurs under binders and binders are shifted about in unusual ways. In order to ensure that binding structure is preserved throughout evaluation, we need to be able to reason locally, within each machine configuration, about bound variables. To make this possible, we make one last change to the machine. We add a list of names to each machine configuration.

$$\boxed{X \quad ::= \quad \overline{x_i}}$$

Most of the machine rules simply pass the list of names along to the next transition. One of the reduce rules manipulates the list of names.

(D.2) $$\langle X \mid E_1, \llbracket E_2, \lambda x.t_1 \rrbracket\ t_2 \rangle_d \longmapsto_{nam}$$
$$\left\langle X, x' \mid E_1 \circ E_2 \circ [(\lambda x'.\Box)\ t_2], t_1[x'/x] \right\rangle_f \quad x' \notin X$$

When this rule goes under a lambda, it adds the name of its bound variable to the list X of variables. The notation X, x expresses adding a new name x to X. If the bound variable x on the left hand side of the rule is already a member of X, then the variable is renamed as part of the transition. As such, X can be considered a set.

Now each machine configuration has one of five forms:

$$\begin{array}{lcl} \langle X \mid E, ? \rangle & ::= & \langle X \mid \llbracket E, v \rrbracket \rangle \mid \langle X \mid E, r \rangle_d \mid \langle X \mid E, t \rangle_f \\ & \mid & \langle X \mid E, v \rangle_b \mid \langle X \mid E, x \rangle_n \end{array}$$

We use the notation $\langle X \mid E, ? \rangle$ below to uniformly discuss all configuration types, where X refers to the list of names, E refers to the context, and ? refers to the term or redex. For a final configuration $\langle X \mid \llbracket E, v \rrbracket \rangle$, ? refers to the answer's underlying value v, and E corresponds to the answer's binder frames E. We use the metavariable C to range over configurations when the internal structure does not matter.

The call-by-need abstract machine uses the set X of names to keep track of *active variables*: any variable x whose binding instance has been pushed into a binder frame $[(\lambda x.\square)\ t]$:

$$\begin{array}{lcl} AV([\,]) & = & \emptyset \\ AV(E \circ [(\lambda x.\square)\ t]) & = & AV(E) \cup \{\, x \,\} \\ AV(E \circ [\square\ t]) & = & AV(E) \\ AV(E \circ [(\kappa x.E_1)\ \square]) & = & AV(E) \cup \{\, x \,\} \cup AV(E_1) \end{array}$$

Cont-bound variables are counted among the active variables because machine evaluation must have gone under a binding to construct the cont frame.

The renaming condition on the $(D.2)$ reduce rule ensures that active variables are mutually distinguishable. This guarantees that the machine's need rule can never capture the wrong evaluation context and thus execute the wrong bound expression.

Renaming is not obviously sufficient to ensure bound variable hygiene because of how the machine manipulates evaluation contexts. For instance, even though the need rule is guaranteed to only match a variable with the right binder frame, we have no guarantee that the right binder frame could never be trapped inside a cont frame and hidden from view while a need transition searches for it. Were this to happen, the machine would get stuck. Furthermore, the reduction rules flip and shift evaluation contexts that might contain binder frames. If a binder frame were to end up below another context frame that contains references to its bound variable, then those references would no longer be bound in the context; the need rule would exhaust the evaluation context if it attempted to resolve any of these references.

To establish that binder frames remain properly positioned, we define a notion of well-formed evaluation contexts:

$$\frac{}{\emptyset \mid [\,]\ \mathbf{wf}} \qquad \frac{X \mid E\ \mathbf{wf} \quad FV(t) \subseteq CV(E)}{X \mid (E \circ [\square\ t])\ \mathbf{wf}}$$

$$\frac{X \mid E\ \mathbf{wf} \quad FV(t) \subseteq CV(E) \quad x \notin X}{X, x \mid (E \circ [(\lambda x.\square)\ t])\ \mathbf{wf}}$$

$$\frac{X \mid (E_1 \circ [(\lambda x.\square)\ (\lambda x.x)] \circ E_2)\ \mathbf{wf}}{X \mid (E_1 \circ [(\kappa x.E_2)\ \square])\ \mathbf{wf}}$$

which is used to define a notion of well-formed machine configurations:

$$\frac{X \mid E\ \mathbf{wf} \quad FV(v) \subseteq CV(E)}{\langle X \mid \llbracket E, v \rrbracket \rangle\ \mathbf{wf}}$$

$$\frac{X \mid (E_1 \circ E_3 \circ [(\kappa x.E_2)\ \square])\ \mathbf{wf} \quad FV(v) \subseteq CV(E_1 \circ E_3)}{\langle X \mid E_1, (\kappa x.E_2)\ \llbracket E_3, v \rrbracket \rangle_d\ \mathbf{wf}}$$

$$\frac{X \mid (E_1 \circ E_2)\ \mathbf{wf} \quad FV(v) \subseteq CV(E_1 \circ E_2) \quad FV(t_2) \subseteq CV(E_1)}{\langle X \mid E_1, \llbracket E_2, v \rrbracket\ t_2 \rangle_d\ \mathbf{wf}}$$

$$\frac{X \mid E\ \mathbf{wf} \quad FV(t) \subseteq CV(E)}{\langle X \mid E, t \rangle_f\ \mathbf{wf}}$$

$$\frac{X \mid E\ \mathbf{wf} \quad FV(v) \subseteq CV(E)}{\langle X \mid E, v \rangle_b\ \mathbf{wf}}$$

$$\frac{X \mid E\ \mathbf{wf} \quad \{\, x \,\} \subseteq CV(E)}{\langle X \mid E, x \rangle_n\ \mathbf{wf}}$$

Both well-formedness relations rely on straightforward notions of captured and free context variables:

$$\begin{array}{lcl} CV([\,]) & = & \emptyset \\ CV([E \circ (\lambda x.\square)\ t]) & = & \{\, x \,\} \cup CV(E) \\ CV([E \circ \square\ t]) & = & CV(E) \\ CV([E \circ (\kappa x.E_1)\ \square]) & = & CV(E) \\ \\ FV([\,]) & = & \emptyset \\ FV([(\lambda x.\square)\ t] \circ E) & = & FV(t) \cup (FV(E) - \{\, x \,\}) \\ FV([\square\ t] \circ E) & = & FV(E) \cup FV(t) \\ FV([(\kappa x.E_1)\ \square] \circ E) & = & FV(E) \cup (FV(E_1) - \{\, x \,\}) \end{array}$$

Well-formedness of an evaluation context guarantees that no binder frames interfere with each other:

Lemma 1. *If $X \mid (E_1 \circ E_2)$ **wf** then $AV(E_1) \cap AV(E_2) = \emptyset$.*

Proof. By lexicographical induction on the measure (k, n) of E_2, where k is its total number of cont frames and n is its linear length. □

Lemma 2. *$CV(E) \subseteq AV(E)$.*

Proof. By induction on the length of E. □

Well-formedness of configurations combined with rule $D.2$'s name management ensures that machine evaluation respects variable binding structure.

Theorem 3. *If t is a closed term of the calculus, then $\langle \emptyset \mid [\,], t \rangle_f$ **wf**.*

Proof. $\emptyset \mid [\,]$ **wf** and $FV(t) \subseteq CV([\,]) = \emptyset$. □

Theorem 4. *Let C_1 and C_2 be configurations. If C_1 **wf** and $C_1 \longmapsto_{nam} C_2$ then C_2 **wf**.*

Proof. By cases on $\longmapsto_{nam}$. The cases are immediate except for two rebuild rules that transition to reduce configurations: each ensures by induction that the subsequent reduction step preserves hygiene. □

In short, well-formedness of the reduce configurations ensures that the reduce rules can be safely performed without any implicit renaming. Since the machine preserves well-formedness, this property persists throughout evaluation. The rest of this paper only considers well-formed configurations.

4.6 An Abstract Machine for Call-by-need

Putting together our observations from the previous section, we now present the specification of the abstract machine. Figure 1 presents its transitions rules. We have derived a heap-less abstract machine for call-by-need evaluation. It replaces the traditional manipulation of a heap using store-based effects with disciplined management of the evaluation stack using control-based effects. In short, state is replaced with control.

Machine evaluation of a program t begins with $\langle \emptyset \mid [\,], t\rangle_f$ and terminates at $\langle X \mid \llbracket E, v \rrbracket\rangle$.

5. Correctness of the Machine

The previous section proves that the machine manipulates terms in a manner that preserves variable binding. In this section, we prove that those manipulations correspond to standard-order call-by-need evaluation.

To proceed, we first establish correspondences between abstract machine configurations and call-by-need terms. As we have alluded to previously, abstract machine contexts correspond directly to calculus contexts:

$$\begin{array}{lcl} \mathcal{C}\llbracket[\,]\rrbracket & = & \Box \\ \mathcal{C}\llbracket[\Box\ t] \circ E\rrbracket & = & \mathcal{C}\llbracket E\rrbracket\ t \\ \mathcal{C}\llbracket[(\kappa x.E_1)\ \Box] \circ E_2\rrbracket & = & (\lambda x.\mathcal{C}\llbracket E_1\rrbracket[x])\ \mathcal{C}\llbracket E_2\rrbracket \\ \mathcal{C}\llbracket[(\lambda x.\Box)\ t] \circ E\rrbracket & = & (\lambda x.\mathcal{C}\llbracket E\rrbracket)\ t \end{array}$$

Redexes also map to call-by-need terms:

$$\begin{array}{lcl} \mathcal{C}\llbracket \llbracket E, v\rrbracket\ t\rrbracket & = & (\mathcal{C}\llbracket E\rrbracket[v])\ t \\ \mathcal{C}\llbracket(\kappa x.E_1)\ \llbracket E_2, v\rrbracket\rrbracket & = & (\lambda x.\mathcal{C}\llbracket E_1\rrbracket[x])\ (\mathcal{C}\llbracket E_2\rrbracket[v]) \end{array}$$

Given that terms map identically to terms, configuration mapping is defined uniformly:

$$\mathcal{C}\llbracket\langle X \mid E, ?\rangle\rrbracket \quad = \quad \mathcal{C}\llbracket E\rrbracket[\ \mathcal{C}\llbracket ?\rrbracket\ ]$$

Since the calculus is defined over alpha equivalence classes, we reason up to alpha equivalence when relating terms to machine configurations.

We now state our fundamental correctness theorems. First we guarantee soundness, the property that every step of the abstract machine respects standard-order reduction.

Theorem 5. *If $t_1 = \mathcal{C}\llbracket C_1\rrbracket$ and $C_1 \longmapsto_{nam} C_2$, then $t_1 \longmapsto\!\!\!\!\rightarrow_{sr} t_2$, for some $t_2 = \mathcal{C}\llbracket C_2\rrbracket$.*

Proof. By cases on $\longmapsto_{nam}$. Only rules $D.1$ and $D.2$ are not immediate. The other rules preserve equality under $\mathcal{C}\llbracket C\rrbracket$. ☐

Corollary 6 (Soundness)**.**
If $t = \mathcal{C}\llbracket C\rrbracket$ and $C \longmapsto\!\!\!\!\rightarrow_{nam} \langle X \mid \llbracket E, v\rrbracket\rangle$, then $t \longmapsto\!\!\!\!\rightarrow_{sr} a$, for some $a = \mathcal{C}\llbracket\langle X \mid \llbracket E, v\rrbracket\rangle\rrbracket$.

Proof. By induction on the length of the $\longmapsto\!\!\!\!\rightarrow_{nam}$ sequence. ☐

We also prove completeness, namely that abstract machine reduction subsumes standard order reduction.

Theorem 7 (Completeness)**.**
If $t = \mathcal{C}\llbracket C\rrbracket$ and $t \longmapsto\!\!\!\!\rightarrow_{sr} a$, then $C \longmapsto\!\!\!\!\rightarrow_{nam} \langle X \mid \llbracket E, v\rrbracket\rangle$, with $a = \mathcal{C}\llbracket\langle X \mid \llbracket E, v\rrbracket\rangle\rrbracket$.

Proof. This proof proceeds by induction on the length of $\longmapsto\!\!\!\!\rightarrow_{sr}$ sequences. It utilizes Proposition 4 to accelerate the $\longmapsto_{sr}$ rules in accordance with $\longmapsto_{nam}$. It also relies on a number of lemmas to establish that $\longmapsto_{nam}$ will find the unique redex of a term from any decomposition of a term into a context E and a subterm t. ☐

Theorem 8 (Correctness)**.** *If $t = \mathcal{C}\llbracket C\rrbracket$, then $t \longmapsto\!\!\!\!\rightarrow_{sr} a$, if and only if $C \longmapsto\!\!\!\!\rightarrow_{nam} \langle X \mid \llbracket E, v\rrbracket\rangle$, with $a = \mathcal{C}\llbracket\langle X \mid \llbracket E, v\rrbracket\rangle\rrbracket$.*

5.1 Discussion

This abstract machine has nice inductive properties. The refocus rules always dispatch on the outermost term constructor. The rebuild and need rules dispatch on a prefix of the context, though each has different criteria for bounding the prefix.

The abstract machine's evaluation steps should not be seen as merely a desperate search for a redex. Rather, the machine exposes the fine-grain structure of call-by-need evaluation, just as the CK machine and the Krivine machine (Krivine 2007) model evaluation for call-by-value and call-by-name respectively. Answers are the partial results of computations, and the rebuild rules represent the process of reconstructing and returning a result to a reduction site. Furthermore, the need rules can be viewed as a novel form of variable-lookup combined with lazy evaluation. The evaluation context captures the rest of computation, but not in order: variable references cause evaluation to skip around in a manner that is difficult to predict.

The way that variables behave in these semantics reveals a connection to coroutines. The reduction rule $D.2$ binds a variable to a delayed computation; referencing that variable suspends the current computation and jumps to its associated delayed computation. Upon completion of that computation, any newly delayed computations (i.e. binder frames) are added to the evaluation context and the original computation is resumed.

The standard-order reduction relation of the call-by-need lambda calculus defines an evaluator concisely but abstractly. Surely unique decomposition, standardization, and hygiene ensure the existence of a deterministic evaluator, but these properties do not spell out the details or implications. Based on a reasoned inspection of standard-order reduction, we expose its computational behavior and capture it in a novel abstract machine that has no store. The improvements to the initial machine produce a variant that effectively assimilates computational information, explicitly accounts for variable hygiene and thereby reveals the coarse-grained operational structure of call-by-need standard-order evaluation.

6. Simulating Call-by-need Using Control

As we allude to above, call-by-need machine evaluation is highly suggestive of delimited control operations, but the connection is indirect and mixed with the other details of lazy evaluation. In this section, we directly interpret this connection in the terminology of delimited control.

Based on the operational behavior of the abstract machine, we derive a simulation of call-by-need execution under call-by-value augmented with delimited control operators. In particular, we translate call-by-need terms into the framework of Dybvig et al. (2007). First we overview the language of delimited control operations. Then we describe how the abstract machine performs delimited control operations. Next we present the simulation of call-by-need using delimited control. Finally we show its correctness.

6.1 Delimited Control Operators

Dybvig et al. (2007) defines a language with delimited control operators. We explain these operators using a machine semantics.

$$
\boxed{\langle X \mid E, r\rangle_d} \quad \text{(Reduce)}
$$

$$
\begin{array}{llll}
(D.1) & \langle X \mid E_1, (\kappa x.E_2)\ \llbracket E_3, v\rrbracket\rangle_d & \longmapsto_{nam} & \langle X \mid E_1 \circ E_3 \circ [(\lambda x.\Box)\ v] \circ E_2, v\rangle_b \\
(D.2) & \langle X \mid E_1, \llbracket E_2, \lambda x.t_1\rrbracket\ t_2\rangle_d & \longmapsto_{nam} & \langle X, x' \mid E_1 \circ E_2 \circ [(\lambda x'.\Box)\ t_2], t_1[x'/x]\rangle_f \quad x' \notin X
\end{array}
$$

$$
\boxed{\langle X \mid E, t\rangle_f} \quad \text{(Refocus)}
$$

$$
\begin{array}{llll}
(F.1) & \langle X \mid E, x\rangle_f & \longmapsto_{nam} & \langle X \mid E, x\rangle_n \\
(F.2) & \langle X \mid E, \lambda x.t\rangle_f & \longmapsto_{nam} & \langle X \mid E, \lambda x.t\rangle_b \\
(F.3) & \langle X \mid E, t_1\ t_2\rangle_f & \longmapsto_{nam} & \langle X \mid E \circ [\Box\ t_2], t_1\rangle_f
\end{array}
$$

$$
\boxed{\langle X \mid E, v\rangle_b} \quad \text{(Rebuild)}
$$

$$
\begin{array}{llll}
(B.1) & \langle X \mid E_b, v\rangle_b & \longmapsto_{nam} & \langle X \mid \llbracket E_b, v\rrbracket\rangle \\
(B.2) & \langle X \mid E_1 \circ [\Box\ t] \circ E_b, v\rangle_b & \longmapsto_{nam} & \langle X \mid E_1, \llbracket E_b, v\rrbracket\ t\rangle_d \\
(B.3) & \langle X \mid E_1 \circ [(\kappa x.E_2)\ \Box] \circ E_b, v\rangle_b & \longmapsto_{nam} & \langle X \mid E_1, (\kappa x.E_2)\ \llbracket E_b, v\rrbracket\rangle_d
\end{array}
$$

where $E_b = \overline{[(\lambda x_i.\Box)\ t_i]}$

$$
\boxed{\langle X \mid E, x\rangle_n} \quad \text{(Need)}
$$

$$
(N.1) \quad \langle X \mid E_1 \circ [(\lambda x.\Box)\ t] \circ E_2, x\rangle_n \longmapsto_{nam} \langle X \mid E_1 \circ [(\kappa x.E_2)\ \Box], t\rangle_f
$$

where $[(\lambda x.\Box)\ t] \notin E_2$

Figure 1. Call-by-need Machine

$$
\begin{array}{lcl}
t & ::= & x \mid v \mid t\ t \mid \mathit{newPrompt} \mid \mathit{pushPrompt}\ t\ t \\
 & \mid & \mathit{withSubCont}\ t\ t \mid \mathit{pushSubCont}\ t\ t \\
v & ::= & \lambda x.t \mid p \mid \langle M\rangle \\
E & ::= & \Box \mid E[\Box\ t] \mid E[(\lambda x.t)\ \Box] \mid E[\mathit{pushPrompt}\ \Box\ t] \\
 & \mid & E[\mathit{withSubCont}\ \Box\ t] \mid E[\mathit{withSubCont}\ p\ \Box] \\
 & \mid & E[\mathit{pushSubCont}\ \Box\ t] \\
M & ::= & [\,] \mid E : M \mid p : M \\
p & ::= & \mathbb{N}
\end{array}
$$

The language extends the call-by-value untyped lambda calculus with the four operators *newPrompt*, *pushPrompt*, *withSubCont*, and *pushSubCont* as well as two new values: first-class *prompts* p, and first-class delimited continuations $\langle M\rangle$. Its control structure is defined using evaluation contexts E, and metacontexts M, which are lists that interleave prompts and contexts. Metacontexts use Haskell list notation. Prompts are modeled using natural numbers.

A program state comprises an expression t, continuation E, metacontinuation M, and fresh prompt source p. The initial state for a program t is $\Box[t], [\,], 0$.

$$
\begin{array}{lcl}
E[(\lambda x.t)\ v], M, p & \mapsto & E[t[v/x]], M, p \\
E[\mathit{newPrompt}], M, p & \mapsto & E[p], M, p+1 \\
E[\mathit{pushPrompt}\ p_1\ t], M, p_2 & \mapsto & \Box[t], p_1 : E : M, p_2 \\
\multicolumn{3}{l}{E[\mathit{withSubCont}\ p_1\ \lambda x.t], M_1 \mathbin{+\!\!+} (p_1 : M_2), p_2 \mapsto} \\
 & & \Box[t[\langle E : M_1\rangle /x]], M_2, p_2 \\
\multicolumn{3}{c}{p_1 \notin M_1} \\
\multicolumn{3}{l}{E[\mathit{pushSubCont}\ \langle M_1\rangle\ \ t], M_2, p \mapsto} \\
 & & \Box[t], M_1 \mathbin{+\!\!+} (E : M_2), p \\
\Box[v], E : M, p & \mapsto & E[v], M, p \\
\Box[v], p_1 : M, p_2 & \mapsto & \Box[v], M, p_2
\end{array}
$$

The four operators manipulate delimited continuations, or *subcontinuations*, which are part of an execution context. The *withSubCont* operator takes a prompt and a function; it captures the smallest subcontinuation that is delimited by the prompt and passes it to the function. The non-captured part of the continuation becomes the new continuation. The prompt instance that delimited the captured subcontinuation is discarded: it appears in neither the captured subcontinuation nor the current continuation. This operator generalizes $\mathcal{F}$ (Felleisen 1988) and `shift` (Danvy and Filinski 1990).

The *pushSubCont* operator takes a subcontinuation and an expression; it composes the subcontinuation with the current continuation and proceeds to evaluate its second argument in the newly extended continuation.

The *pushPrompt* operator takes a prompt and an expression; it extends the current continuation with the prompt and evaluates the expression in the newly extended continuation. The *newPrompt* operator returns a distinguished fresh prompt each time it is called. These two operators generalize the delimiting operators # (Felleisen 1988) and `reset` (Danvy and Filinski 1990), which extend a continuation with a single common delimiter.

To illustrate these operators in action, we consider a program that uses arithmetic and conditionals:

$$
\begin{array}{l}
\mathit{let}\ p = \mathit{newPrompt} \\
\mathit{in}\ 2 + \mathit{pushPrompt}\ p \\
\qquad \mathit{if}\ (\mathit{withSubCont}\ p \\
\qquad\quad (\lambda k.(\mathit{pushSubCont}\ k\ \mathit{False}) + \\
\qquad\qquad (\mathit{pushSubCont}\ k\ \mathit{True}))) \\
\qquad \mathit{then}\ 3 \\
\qquad \mathit{else}\ 4
\end{array}
$$

A fresh prompt is bound to p and pushed onto the continuation just prior to evaluation of the *if* expression. *withSubCont* captures the subcontinuation $[\mathit{if}\ \Box\ \mathit{then}\ 3\ \mathit{else}\ 4]$, which was delimited by p, and binds it to k. The subcontinuation k is pushed twice, given the value False the first time and True the second. The result of evaluation is the expression $2 + 4 + 3$ which yields 9.

6.2 Delimited Control Naïvely Simulates the Machine

The call-by-need abstract machine performs two different kinds of partial control capture. To review, the rebuild and need rules of the abstract machine both capture some portion of the evaluation context. In particular, the rebuild rules capture binder frames. If only binder frames remain, then execution is complete. When either of the other frames is found, then a reduction is performed. On the

Let s be a distinguished identifier:

$\mathcal{P}[\![t]\!] = runCC\ (let\ s = newPrompt\ in\ pushPrompt\ s\ \mathcal{K}[\![t]\!])$

$$\begin{aligned}
\mathcal{K}[\![x]\!] = {} & withSubCont\ x\ \lambda k. \\
& \quad \lambda f_{th}.\mathsf{do}\ v_a \Leftarrow \mathsf{force}\ f_{th} \\
& \qquad \mathsf{in\ delay}\ (\mathsf{return}\ v_a)\ \mathsf{as}\ x \\
& \qquad\quad \mathsf{in}\ pushSubCont\ k\ (\mathsf{return}\ v_a)
\end{aligned}$$

$$\begin{aligned}
\mathcal{K}[\![t_1\ t_2]\!] = {} & \mathsf{do}\ v_a \Leftarrow \mathcal{K}[\![t_1]\!] \\
& \mathsf{in}\ let\ x_p = newPrompt \\
& \quad in\ \mathsf{delay}\ \mathcal{K}[\![t_2]\!]\ \mathsf{as}\ x_p\ \mathsf{in}\ (v_a\ x_p)
\end{aligned}$$

$\mathcal{K}[\![\lambda x.t]\!] = \mathsf{return}\ \lambda x.\mathcal{K}[\![t]\!]$

$\mathsf{return}\ v_a \equiv withSubCont\ s\ \lambda k_a.\ \langle k_a, v_a\rangle$

$$\begin{aligned}
\mathsf{do}\ x \Leftarrow t_1\ \mathsf{in}\ t_2 \equiv {} & let\ \langle k_a, x\rangle = pushPrompt\ s\ t_1 \\
& in\ pushSubCont\ k_a\ t_2
\end{aligned}$$

$$\begin{aligned}
\mathsf{delay}\ t_1\ \mathsf{as}\ x\ \mathsf{in}\ t_2 \equiv {} & let\ f_k = pushPrompt\ x\ t_2 \\
& in\ f_k\ \lambda().t_1
\end{aligned}$$

$\mathsf{force}\ f \equiv f\ ()$

Figure 2. Translating CBN to CBV+Control

other hand, the need rule captures the evaluation context up to the binder that matches the variable whose value is needed.

These actions of the abstract machine can be recast in the language of delimited control capture. First, the need rule uses the identity of its variable, which must be an active variable, to delimit the context it captures. The well-formedness conditions from Section 4.5 guarantee that each binder frame binds a unique variable, so each active variable acts as a unique delimiter. Second, the rebuild rule uses the nearest non-binder frame to delimit the context it captures. This means that rebuild operates as though the operand frames, the cont frames, and the top of the evaluation context share a common delimiter. This guarantees that only binder frames are captured (as is stipulated in the rules).

In short, call-by-need evaluation captures partial evaluation contexts. These partial evaluation contexts correspond to delimited continuations, and there are two different kinds of delimitation, redex-based (for rebuild) and binder-based (for need).

It is useful to also consider how the machine manipulates these delimited continuations. Each reduce rule in Figure 1 immediately pushes the context associated with an answer onto the current evaluation context. In this manner, binders are consistently moved above the point of evaluation. The reduce rule then operates on the value part of the answer and the associated cont (for $D.1$) or term (for $D.2$).

Although each reduce rule pushes binders onto the evaluation context, only the $D.2$ rule creates new binders. The variable bound by the answer's underlying lambda abstraction may already be a member of the set X, in which case it must be alpha-converted to a fresh name with respect to the set X. Also note that if $\lambda x.t$ is alpha converted to $\lambda x'.t[x'/x]$, the body under call-by-value satisfies the equation $t[x'/x] = (\lambda x.t)\ x'$. Since we are using the identifiers x' as delimiters, and we never turn the binder frame $[(\lambda x'.\square)\ t]$ back into a term, we can replace fresh variables x' with fresh *prompts*.

From these observations, we construct the simulation in Figure 2. The simulation can be understood as a direct encoding of the abstract machine semantics for call-by-need. To execute a program, $\mathcal{P}[\![t]\!]$, the transformation uses *runCC* to initiate a control-based computation, acquires a fresh prompt, and binds it to a distinguished variable s. This prompt is the *redex prompt*, which is used to delimit every continuation that denotes a redex.

To expose the conceptual structure of the simulation, we define four syntactic macros, do, return, delay, and force. We accord no formal properties to them: they merely simplify the presentation. The return macro captures the nearest subcontinuation that is delimited by the redex prompt s. Since the s delimiter appears before every reduction, the captured continuation is guaranteed to contain only code equivalent to binder frames. The translation returns a tuple containing the subcontinuation and the argument to return, which must be a value; the tuple represents an answer. So the translation rule for lambda abstractions, $\mathcal{K}[\![\lambda x.t]\!]$, literally simulates the rebuild rules.

The do macro executes a term t_1 under the current continuation extended with the redex prompt. If the term returns an answer $\langle k_a, x\rangle$ it immediately pushes the subcontinuation part and continues execution, binding the value part to the variable x. As such, the translation rule for applications, $\mathcal{K}[\![t_1\ t_2]\!]$, executes $[\![t_1]\!]$ and binds the resulting operator to v_a. The answer binders are pushed by the do macro, which starts the simulation of the $D.2$ rule.

The remainder of the $D.2$ rule is subtle. In the abstract machine, binder frame variables delimit the need rules. Since the delimited continuation framework relies on prompts to delimit continuations, fresh prompts literally substitute for variables (Kiselyov et al. 2006). The translation uses *newPrompt* to acquire a fresh prompt x_p and then uses the delay macro to simulate pushing a binder frame: the context $\mathsf{delay}\ t\ \mathsf{as}\ x\ \mathsf{in}\ \square$ is analogous to the binder frame $[(\lambda x.\square)\ t]$. The delay macro anticipates that its body returns a function f_k that expects the delayed argument, so it applies f_k to a suspension of t. As we see below, the function f_k is a cont $(\kappa x.E)$.

In the context of delay, the simulation executes $v_a\ x_p$. Since alpha conversion of $\lambda x.t$ can be written $(\lambda x_p.t[x_p/x])$, the term $v_a x_p$ is analogous to $(\lambda x.t)\ x_p = t[x_p/x]$: it substitutes a fresh prompt for a fresh variable.

The translation rule for variables, $\mathcal{K}[\![x]\!]$, captures the continuation delimited by x (which had better be a prompt!) and returns a function $\lambda f_{th}.\ldots$ that closes over both x and the captured continuation k. This function is the cont $\kappa x.E$, with x modeling the bound variable of the same name, and continuation k modeling E. The function expects the binder frame $[(\lambda x.\square)\ t]$, which is now at the top of the current continuation, to pass it the suspension $\lambda().[\![t]\!]$. The simulation forces the suspension, and the do macro pushes the resulting answer binders and binds v_a to the underlying value. Pushing the answer binders begins the simulation of the $D.1$ rule.

The simulation of $D.1$ delays a computation that immediately returns the result v_a of evaluating the term t, pushes the continuation k associated with the cont, and returns v_a to the extended continuation. Now any subsequent evaluation of x immediately returns the memoized value v_a instead of recomputing t. This yields an answer $\langle k_a, v_a\rangle$ where k_a is an empty subcontinuation. The value v_a is delayed exactly as before and is also returned from the properly extended continuation. This part of the translation bears close resemblance to the paradoxical Y combinator (Curry and Feys 1958), suggesting that the simulation requires recursive types (Shan 2007).

7. Correctness of the Simulation

We prove correctness of the simulation relative to the machine semantics. Since we already proved correctness of the machine semantics relative to standard-order reduction, the result is a proof that our simulation provides a continuation semantics for call-by-need.

The previous discussion provides an informal justification for the structure of the call-by-need simulation. To prove the correctness of the simulation, we appeal to the continuation semantics

$$\begin{array}{l}
newPrompt_c = \lambda\kappa.\lambda\gamma.\lambda q.\kappa\ q\ \gamma\ (q+1) \\
withSubCont_c = \lambda p.\lambda f.\lambda\kappa.\lambda\gamma.f\ (\kappa : \gamma^p_\uparrow)\ \kappa_0\ \gamma^p_\downarrow \\
pushPrompt_c = \lambda p.\lambda t.\lambda\kappa.\lambda\gamma.t\ \kappa_0\ (p : \kappa : \gamma) \\
pushSubCont_c = \lambda\gamma'.\lambda t.\lambda\kappa.t\ \kappa_0\ (\gamma' \mathbin{++} (\kappa : \gamma)) \\
\kappa_0 = \lambda v.\lambda\gamma\lambda q.\mathcal{K}(v, \gamma, q) \\
\\
\mathcal{K}(v, [\,], q) = v \\
\mathcal{K}(v, p : \gamma, q) = \mathcal{K}(v, \gamma, q) \\
\mathcal{K}(v, \kappa : \gamma, q) = \kappa\ \gamma\ q
\end{array}$$

Figure 3. Delimited Control Combinators

for delimited control (Dybvig et al. 2007). This semantics is completely standard for the terms of the lambda calculus. Figure 3 presents the interesting parts of the semantics. All CPS terms take a standard continuation κ, but the control combinators also take a *metacontinuation* γ, which is a list of continuations and prompts, and a global prompt counter q. The base continuation κ_0 delimits each proper continuation and directs evaluation up the metacontinuation, discarding any intervening prompts. Given a CPS program t, the expression $t\ \kappa_0\ [\,]\ 0$ runs it.

To prove correctness, we compose $\mathcal{K}[\![\cdot]\!]$ with the delimited continuation semantics to produce a translation $[\![\cdot]\!]_{\lambda_{\beta\eta}}$ to the $\lambda_{\beta\eta}$ calculus. We also give each abstract machine configuration and its constituents a denotation (see Figures 4 through 7).

$$\begin{array}{l}
[\![t]\!]_X = [\![t]\!]_{\lambda_{\beta\eta}} \overline{[\iota(x_i, X)/x_i]} \\
\\
[\![t]\!]^P_{\lambda_{\beta\eta}} = [\![t]\!]_{\lambda_{\beta\eta}}\ \kappa_0\ (0 : [\,])\ 1 \\
\\
[\![x]\!]_{\lambda_{\beta\eta}} = \\
withSubCont_c\ x \\
\quad \lambda k_x.\lambda k_1.k_1 \\
\quad\quad \lambda f_{th}.\lambda k_2. \\
\quad\quad\quad pushPrompt_c\ 0\ (f_{th}\ ()) \\
\quad\quad\quad\quad (\lambda\,\langle k_a, v_a\rangle. \\
\quad\quad\quad\quad\quad pushSubCont_c\ k_a \\
\quad\quad\quad\quad\quad\quad (\lambda k_3. \\
\quad\quad\quad\quad\quad\quad\quad pushPrompt_c\ x \\
\quad\quad\quad\quad\quad\quad\quad\quad (pushSubCont_c\ k_x \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad (withSubCont_c\ 0\ \lambda k_a.\lambda k.k\ \langle k_a, v_a\rangle)) \\
\quad\quad\quad\quad\quad\quad\quad\quad (\lambda f_k.f_k\ (\lambda().withSubCont_c\ 0 \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \lambda k_a.\lambda k.k\ \langle k_a, v_a\rangle) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad k_3)) \\
\quad\quad\quad\quad\quad\quad k_2) \\
\\
[\![t_1\ t_2]\!]_{\lambda_{\beta\eta}} = \lambda k_1.pushPrompt_c\ 0\ [\![t_1]\!]_{\lambda_{\beta\eta}} \\
\quad\quad\quad\quad\quad\quad\quad (\lambda\,\langle k_a, v_a\rangle. \\
\quad\quad\quad\quad\quad\quad\quad\quad pushSubCont_c\ k_a \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad (\lambda k_2. \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad newPrompt_c \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \lambda x_p.pushPrompt_c\ x_p\ (v_a\ x_p) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (\lambda f_k.f_k\ (\lambda().[\![t_2]\!]_{\lambda_{\beta\eta}})\ k_2)) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad k_1) \\
\\
[\![\lambda x.t]\!]_{\lambda_{\beta\eta}} = withSubCont_c\ 0\ \lambda k_a.\lambda k.k\ \langle k_a, \lambda x.[\![t]\!]_{\lambda_{\beta\eta}}\rangle
\end{array}$$

Figure 4. Denotations for Terms

Denotations of machine configurations are constructed from their components: the configuration's focus ?, context E, and list of names X. A machine configuration denotes the translation of its focus applied to three arguments: the base continuation κ_0 as its starting continuation, the denotation of its context, bounded by the redex delimiter 0, as the metacontinuation, and the size $|X|$ of X plus 1 as its initial prompt. The redex delimiter attached to the metacontinuation handles the case when an answer subsumes the entire context by returning the answer as the result.

$$\begin{array}{l}
\iota(x_i, X) = \iota(x_i, [x_1, x_2, \ldots, x_i, \ldots, x_n]) = i \\
\\
|X| = |[x_1, x_2, \ldots, x_i, \ldots, x_n]| = n \\
\\
[\![\langle X \mid E, ?\rangle]\!]_{\lambda_{\beta\eta}} = [\![?]\!]_X\ \kappa_0\ ([\![E]\!]_X \mathbin{++} (0 : [\,]))\ (|X| + 1)
\end{array}$$

Figure 5. Denotations for Names and Configurations

Our semantic translation takes advantage of X being a proper list of unique names. Free active variables denote prompts in our translation, and since 0 is the redex delimiter, we assign to each variable its 1-based index in X. We use $|X| + 1$ as the global prompt counter to ensure that no future prompts conflict with the current active variable denotations, thereby guaranteeing hygiene (see Section 4.5).

Each evaluation context frame denotes a two-element metacontinuation consisting of a prompt and a proper continuation. The prompt for a binder frame is the prompt translation $\iota(x, X)$ of the bound variable x. The cont and operand frames have redex prompts 0. These prompts guarantee that answer building operations will arrive at the innermost redex. Each continuation function specializes a subexpression of the CPS translation for terms $[\![\cdot]\!]_X$ with the denotations of the context frame's parts. Compare, for instance, the denotation of an application, $t_1\ t_2$, to that of an operand frame, $[\Box\ t_2]$. The application term pushes the global prompt, and executes t_1 in the context of a continuation that receives an answer $\langle k_a, v_a\rangle$. The denotation of the operand frame is a metacontinuation containing the same prompt and continuation.

$$\begin{array}{l}
[\![E \circ [f]]\!]_X = [\![[f]]\!]_X \mathbin{++} [\![E]\!]_X \\
\\
[\![[\,]]\!]_X = [\,] \\
\\
[\![[\#]]\!]_X = \kappa_0 : [\,] \\
\\
[\![[\Box\ t_2]]\!]_X = 0 : k' : [\,] \\
\text{where } k' = \lambda\,\langle k_a, v_a\rangle. \\
\quad\quad\quad\quad\quad pushSubCont_c\ k_a \\
\quad\quad\quad\quad\quad\quad (\lambda k_2. \\
\quad\quad\quad\quad\quad\quad\quad newPrompt_c \\
\quad\quad\quad\quad\quad\quad\quad\quad \lambda x_p.pushPrompt_c\ x_p\ (v_a\ x_p) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (\lambda f_k.f_k\ (\lambda().[\![t_2]\!]_X)\ k_2)) \\
\quad\quad\quad\quad\quad\quad \kappa_0 \\
\\
[\![[(\lambda x.\Box)\ t_2]]\!]_X = \iota(x, X) : k' : [\,] \\
\text{where } k' = \lambda f_k.f_k\ (\lambda().[\![t_2]\!]_X)\ \kappa_0 \\
\\
[\![[(\kappa x.E)\ \Box]]\!]_X = 0 : k' : [\,] \\
\text{where } k' = \\
\lambda\,\langle k_a, v_a\rangle. \\
\quad pushSubCont_c\ k_a \\
\quad\quad (\lambda k_3. \\
\quad\quad\quad pushPrompt_c\ \iota(x, X) \\
\quad\quad\quad\quad (pushSubCont_c\ [\![E]\!]_X \\
\quad\quad\quad\quad\quad (withSubCont_c\ 0\ \lambda k_a.\lambda k.k\ \langle k_a, v_a\rangle)) \\
\quad\quad\quad\quad (\lambda f_k.f_k\ (\lambda().withSubCont_c\ 0\ \lambda k_a.\lambda k.k\ \langle k_a, v_a\rangle) \\
\quad\quad\quad\quad\quad\quad k_3)) \\
\quad\quad \kappa_0
\end{array}$$

Figure 6. Denotations for Evaluation Contexts

A redex denotes a CPS'ed term that closes over the denotations of its constituents and implements the corresponding reduction step.

To facilitate our proof of correctness, we make a slight change to the machine semantics. In the machine, composing an empty

$$
\begin{array}{l}
[\![\kappa x_1.E_1 \; \lfloor\!\lfloor E_2, \lambda x_2.t \rfloor\!\rfloor]\!]_X = \\
pushSubCont_c \; [\![E_2]\!]_X \\
\quad (\lambda k_3. \\
\quad\quad pushPrompt_c \; \iota(x_1, X) \\
\quad\quad\quad (pushSubCont_c \; [\![E_1]\!]_X \\
\quad\quad\quad\quad (withSubCont_c \; 0 \; \lambda k_a.\lambda k.k \; \langle k_a, \lambda x_2.[\![t]\!]_X\rangle)) \\
\quad\quad\quad (\lambda f_k.f_k \; (\lambda().withSubCont_c \; 0 \; \lambda k_a.\lambda k.k \; \langle k_a, \lambda x_2.[\![t]\!]_X\rangle)) \\
\quad\quad\quad\quad\quad\quad k_3)) \\
\\
[\![\lfloor\!\lfloor E, \lambda x.t_1 \rfloor\!\rfloor \; t_2]\!]_X = \\
pushSubCont_c \; [\![E]\!]_X \\
\quad (\lambda k_2. \\
\quad\quad newPrompt_c \\
\quad\quad\quad \lambda x_p.pushPrompt_c \; x_p \; ((\lambda x.[\![t_1]\!]_X) \; x_p) \\
\quad\quad\quad\quad\quad (\lambda f_k.f_k \; (\lambda().[\![t_2]\!]_X) \; k_2))
\end{array}
$$

Figure 7. Denotations for Redexes

context with the current evaluation is an identity operation. The continuation semantics do not share this property. During execution, an empty continuation is denoted by the base continuation κ_0. If a continuation is captured or pushed in the context of an empty continuation, then the empty continuation will be captured as part of the metacontinuation or pushed onto the current metacontinuation before reinstating the pushed continuation. In short, the call-by-need machine semantics guarantees that $E \circ [\;] = E$, but the continuation semantics do not prove that $\kappa_0 : \gamma = \gamma$. Dybvig et al. discuss the notion of proper tail recursion for delimited continuations. Their operational characterization of proper tail recursion corresponds to the latter equation.

To remove this mismatch, we add a *ghost* frame $[\#]$ to our definition of evaluation contexts. The ghost frame denotes the metacontinuation $\kappa_0 : [\;]$. We also extend the unplug operation on evaluation contexts such that it discards ghost frames: $\mathcal{C}[\![E \circ [\#]]\!] = \mathcal{C}[\![E]\!]$. Finally, we alter the right hand side of transition rules that grab and push continuations to pair ghost frames with composed evaluation contexts in a manner consistent with the continuation semantics. For instance, the updated $D.2$ rule is as follows:

$$
\text{(D.2)} \quad \langle X \mid E_1, (\kappa x.E_2) \; \lfloor\!\lfloor E_3, v \rfloor\!\rfloor\rangle_d \longmapsto_{nam} \\
\langle X \mid E_1 \circ [\#] \circ E_3 \circ [(\lambda x.\Box) \; v] \circ [\#] \circ E_2, v\rangle_b
$$

These modifications do not alter the observable behavior of the machine while modeling the property that pushing empty frames has meaning in the continuation semantics.

Given these denotations, it is straightforward to prove correctness of the simulation relative to the abstract machine.

Theorem 9. *If t is a closed term, then* $[\![t]\!]^P_{\lambda_{\beta\eta}} = [\![\langle\emptyset \mid [\;], t\rangle_f]\!]_{\lambda_{\beta\eta}}$.

Proof. $[\![t]\!]^P_{\lambda_{\beta\eta}} = [\![t]\!]_\emptyset \; \kappa_0 \; (0 : [\;]) \; 1 = [\![\langle\emptyset \mid [\;], t\rangle_f]\!]_{\lambda_{\beta\eta}}$. □

Theorem 10. *If $C_1 \longmapsto_{nam} C_2$ then* $[\![C_1]\!]_{\lambda_{\beta\eta}} = [\![C_2]\!]_{\lambda_{\beta\eta}}$.

Proof. By cases on $\longmapsto_{nam}$. The proof utilizes only beta and eta equivalences to establish correspondences. □

8. Conclusions

In this paper, we expose and examine the operational structure of lazy evaluation as embodied in call-by-need semantics. We present this understanding in two ways: as an abstract machine whose operational behavior involves control capture, and as a simulation of call-by-need under call-by-value plus delimited control operations. Delimited control can be used to simulate a global heap, but our particular simulation uses delimited control operations to manage laziness locally, just like the calculus reduction rules.

The artifacts of this investigation provide new tools for increasing our understanding of lazy evaluation and its connections to control. The abstract machine could be used to establish connections to heap-based implementations of call-by-need, and possibly modern graph-reduction based formulations (Peyton Jones and Salkild 1989). In fact it seems that the calculus and abstract machine may point out new structural and dynamic invariants that are inherent to call-by-need evaluation but are hidden in the unstructured representations of heaps.

The abstract machine and simulation might also provide new opportunities for reasoning about the correctness of transformations applied to call-by-need programs. Surely the calculus provides the same equational reasoning powers as the abstract machine. However the machine may enable researchers to better intuit transformations and justifications that are not as easy to recognize in the reduction semantics. Our simulation might be connected to that of Okasaki et al. (1994). The simulation might suggest new mechanisms by which to embed call-by-need evaluation within call-by-value programs.

One significant difference between the two formulations of call-by-need lambda calculi (Maraist et al. 1998; Ariola and Felleisen 1997) is the status of variables. Maraist et al. consider variables to be values, whereas Ariola and Felleisen do not. This paper sheds no light on the inclusion of variables among the values, however it demonstrates in stark detail the consequences of the latter design. In the abstract machine, the transition rules for lambda terms, namely the rebuild rules, differ significantly from the transition rules for variables, the need rules. A similar distinction can be seen simply by observing the complexity of their respective translations. In short, our semantics interpret variables as memoized computations rather than values.

Our results reveal that a proliferation of semantic frameworks is a boon and not a crisis. The reduction semantics of call-by-need elegantly and mysteriously encode a rich semantics whose broad implications can be seen in equivalent machine semantics and continuation semantics. As such, our work provides new perspectives from which to reason about call-by-need, delimited control, and their respective expressive powers.

9. Acknowledgements

We thank Daniel P. Friedman, Roshan James, William Byrd, Michael Adams, and the rest of the Indiana University Programming Languages Group, as well as Jeremy Siek, Zena Ariola, Phil Wadler, and anonymous referees for helpful discussions and feedback on this work.

References

Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

Zena M. Ariola and Matthias Felleisen. The call-by-need lambda calculus. *Journal of Functional Programming*, 7(3):265–301, May 1997.

Zena M. Ariola, John Maraist, Martin Odersky, Matthias Felleisen, and Philip Wadler. A call-by-need lambda calculus. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 233–246, New York, NY, USA, 1995. ACM Press. ISBN 0-89791-692-1.

Henk P. Barendregt. *The Lambda Calculus, its Syntax and Semantics*. North-Holland, Amsterdam, NL, 1981. Studies in Logic and the Foundations of Mathematics.

Malgorzata Biernacka and Olivier Danvy. A concrete framework for environment machines. *ACM Transactions on Computational Logic*, 9(1):6, 2007. ISSN 1529-3785.

Haskell Brookes Curry and Robert Feys. *Combinatory Logic, Volume I*. Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1958. Second printing 1968.

Olivier Danvy and Andrzej Filinski. Abstracting control. In *LFP '90: Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 151–160, New York, NY, USA, 1990. ACM. ISBN 0-89791-368-X.

Olivier Danvy and Lasse R. Nielsen. Refocusing in reduction semantics. Technical Report RS-04-26, BRICS, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, November 2004.

R. Kent Dybvig, Simon Peyton Jones, and Amr Sabry. A monadic framework for delimited continuations. *Journal of Functional Programming*, 17(6):687–730, 2007. ISSN 0956-7968.

Matthias Felleisen. The theory and practice of first-class prompts. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 180–190, New York, NY, USA, 1988. ACM. ISBN 0-89791-252-7.

Matthias Felleisen and Matthew Flatt. Programming languages and lambda calculi. Available from http://www.cs.utah.edu/plt/publications/pllc.pdf, January 2002.

Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD-machine, and the λ-calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts*, pages 193–217. North-Holland, 1986.

Matthias Felleisen and Robert Hieb. A revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

Daniel P. Friedman and David S. Wise. CONS should not evaluate its arguments. In S. Michaelson and Robin Milner, editors, *Automata, Languages and Programming*, pages 257–284, Edinburgh, Scotland, 1976. Edinburgh University Press.

Daniel P. Friedman, Abdulaziz Ghuloum, Jeremy G. Siek, and Onnie Lynn Winebarger. Improving the lazy Krivine machine. *Higher-Order and Symbolic Computation*, 20(3):271–293, 2007.

Jeremy Gibbons and Keith Wansbrough. Tracing lazy functional languages. In Michael E. Houle and Peter Eades, editors, *Proceedings of Conference on Computing: The Australian Theory Symposium*, pages 11–20, Townsville, January 29–30 1996. Australian Computer Science Communications. ISBN ISSN 0157-3055.

Peter Henderson and James H. Morris, Jr. A lazy evaluator. In *POPL '76: Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 95–103, New York, NY, USA, 1976. ACM.

Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. Closing the stage: From staged code to typed closures. In *PEPM '08: Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 147–157, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-977-7.

Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In *ICFP '05: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 192–203, New York, NY, USA, 2005. ACM. ISBN 1-59593-064-7.

Oleg Kiselyov, Chung-chieh Shan, and Amr Sabry. Delimited dynamic binding. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 26–37, New York, NY, USA, 2006. ACM. ISBN 1-59593-309-3.

Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20(3):199–207, 2007.

John Launchbury. A natural semantics for lazy evaluation. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 144–154, New York, NY, USA, 1993. ACM. ISBN 0-89791-560-7.

John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. *Journal of Functional Programming*, 8 (3):275–317, May 1998.

Eugenio Moggi and Amr Sabry. An abstract monadic semantics for value recursion. *Theoretical Informatics and Applications*, 38(4):375–400, 2004.

Chris Okasaki, Peter Lee, and David Tarditi. Call-by-need and continuation-passing style. *Lisp and Symbolic Computation*, 7 (1):57–81, January 1994.

Simon L. Peyton Jones and Jon Salkild. The spineless tagless G-machine. In *FPCA '89: Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 184–201, New York, NY, USA, 1989. ACM. ISBN 0-89791-328-0.

Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, December 1975.

Peter Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, 1997.

Chung-chieh Shan. A static simulation of dynamic delimited control. *Higher-Order and Symbolic Computation*, 20(4):371–401, 2007.

Gerald Jay Sussman and Guy L. Steele Jr. Scheme: An interpreter for extended lambda calculus. *Higher Order Symbolic Computation*, 11(4):405–439, 1998. ISSN 1388-3690.

Mitchell Wand and Dale Vaillancourt. Relating models of backtracking. In *ICFP '04: Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*, pages 54–65, New York, NY, USA, 2004. ACM. ISBN 1-58113-905-5.

Ching-lin Wang. Obtaining lazy evaluation with continuations in Scheme. *Information Processing Letters*, 35(2):93–97, 1990. ISSN 0020-0190.

Hongwei Xi. Evaluation under lambda abstraction. In *PLILP '97: Proceedings of the Ninth International Symposium on Programming Languages: Implementations, Logics, and Programs*, pages 259–273, London, UK, 1997. Springer-Verlag. ISBN 3-540-63398-7.