Classical call-by-need sequent calculi: The unity of semantic artifacts

Zena M. Ariola[1] & Paul Downen[1] & Hugo Herbelin[2] & Keiko Nakata[3] & Alexis Saurin[2]

[1] University of Oregon, ariola@cs.uoregon.edu pdownen@cs.uoregon.edu
[2] CNRS, PPS UMR 7126, Univ Paris Diderot, Sorbonne Paris Cité, PiR2, INRIA Paris Rocquencourt, F-75205 Paris, France {herbelin,saurin}@pps.jussieu.fr
[3] Institute of Cybernetics, Tallinn University keiko@cs.ioc.ee

Abstract. We systematically derive a classical call-by-need sequent calculus, which does not require an unbounded search for the standard redex, by using the unity of semantic artifacts proposed by Danvy *et al.* The calculus serves as an intermediate step toward the generation of an environment-based abstract machine. The resulting abstract machine is context-free, so that each step is parametric in all but one component. The context-free machine elegantly leads to an environment-based CPS transformation. This transformation is observationally different from a natural classical extension of the transformation of Okasaki *et al.*, due to duplication of un-evaluated bindings.

Keywords: call-by-need, lazy evaluation, duality of computation, sequent calculus, $\lambda\mu$-calculus, classical logic, control

1 Introduction

Lazy languages such as Haskell use the call-by-need evaluation model. It has been formalized by Ariola *et al.* [1] and Maraist *et al.* [7], and while their equational theory differs, the calculi are observationally equivalent. Both Garcia *et al.* [6] and Danvy *et al.* [5] present abstract machines that implement the standard call-by-need reduction. The two machines are observationally the same — however, they differ substantially in their construction. Danvy *et al.* derive an abstract machine systematically from the standard reduction using correctness-preserving program transformations, and thus the resulting abstract machine is correct by construction.

Classical call-by-need, an extension of call-by-need with control operators, was introduced by Ariola *et al.* [3]. Unlike minimal call-by-need (without control operators), the classical extension does not preserve observational equivalence with call-by-name. Consider the evaluation of Example 1 below.

Example 1.

```
let a = callcc(fn k ⇒ (true, fn x ⇒ throw k x))
    x = fst a
    q = snd a
in if x then q (false, fn x ⇒ 0) else 99
```

According to the call-by-name evaluation strategy, the terms bound to a, x, and q are immediately substituted, giving the term

```
if fst (callcc(fn k ⇒ (true, fn x ⇒ throw k x)))
  then (snd (callcc(fn k ⇒ (true, fn x ⇒ throw k x)))) (false, fn x ⇒ 0)
  else 99
```

which leads to the result 0. However, when using the call-by-need strategy, we cannot substitute the let-bound terms since that would duplicate unevaluated terms, violating the call-by-need principle of sharing results. According to the classical call-by-need calculus of Ariola *et al.*, Example 1 returns 99.

In a language with control, determining which computations must be *shared* becomes an interesting question. In a term of the form (where I stands for the identity function)

```
let a = I I in let b = callcc(fn k ⇒ e) in e'
```

even though continuation k might be called more than once, it seems reasonable to share the computation of (I I) across each call to k. In general, it makes sense to share bound computations captured *outside* a control effect among each invocation of the continuation. Now consider a term in which bound computations are captured *inside* a control effect.

Example 2.

```
let a = callcc(fn k ⇒ (I, fn x ⇒ throw k x))
    f = fst a
    q = snd a
in f q (I, I)
```

Are the results of f and q shared among different invocations of continuation k? It turns out that the answer is not so simple. And even more surprisingly, the most natural way to answer the question depends on the starting point.

One way is to start with the sequent calculus, a calculus in which control is already explicit. By defining a call-by-need standardization in the most natural way, all bindings inside of a control effect are not shared between separate invocations. So according to the call-by-need sequent calculus of Ariola *et al.* [3], f and q are recomputed each time k is called. Therefore, the above program produces (I, I) as a result.

Another approach is to begin with a continuation-passing style (CPS) transformation of minimal call-by-need, like the one of Okasaki *et al.* [8] which translates minimal call-by-need into a call-by-value calculus with assignment. Since a program written in CPS form has its control flow explicitly reified as a function, it is easy to extend the translation with control operators. By extending the

CPS transformation of Okasaki *et al.* with control operators, sharing inside of a control effect is much more complicated. When a control effect is forced, only the chain of forced bindings leading to that effect are not shared. According to this definition, the computations bound to a and f will not be shared, whereas the computation bound to q is shared across every call to k. Using this semantics, the above program instead loops forever.

The goal of this paper is to clearly illustrate these two different semantics by deriving an abstract machine and CPS transform for classical call-by-need using the unity of semantic artifacts, as first described by Danvy [4]. For a given notion of semantics, there is a calculus, abstract machine, and CPS transform that correspond exactly with one another. Therefore, from any one of the three semantic artifacts, the others may be systematically derived. We begin with a small variant of Ariola *et al.*'s [3] call-by-need sequent calculus (Section 2) and derive the machine that it corresponds to (Section 3). This machine can be reinterpreted as a calculus that is interesting in its own right—the *multicut* calculus. The multicut calculus does for call-by-need what the sequent calculus does for call-by-value and call-by-name: the standard redex is always at the top of the program so that there is no unbounded search for the next reduction to be performed. However, the machine generated in Section 3 is not satisfying for two reasons. First, the machine inefficiently uses a substitution operation during evaluation, which must traverse the entire sub-program in a single step. Second, the evaluation strategy is not context-free. Given a term and its context, both must be analyzed together in order for the machine to take a step. In other words, the meaning of a term depends on its context, and vice versa. The solution to both of these problems is the same: store *all* terms and contexts in the environment instead of using an early substitution strategy. By applying this change to the multicut calculus we get a context-free, environment-based abstract machine (Section 4). Using the context-free machine in Section 4, we generate an environment-based CPS transform for classical call-by-need (Section 5). In order to see the impact of the approach in defining a semantics for classical call-by-need, we extend Okasaki *et al.*'s [8] CPS transform with control in a natural way, and illustrate the difference. We reflect on the subtleties that arise when defining a classical call-by-need language by exploring some of the non-trivial choices, and outline an alternate sequent calculus in which variables are not values (Section 6). The unity of semantic artifacts provides a robust foundation for such exploration—the same technique can be applied to re-derive the abstract machine and CPS transformation for the modified calculus. [4]

2 Call-by-need sequent calculus ($\overline{\lambda}_{lv}$)

We present a small revision of the classical call-by-need sequent calculus, $\overline{\lambda}_{lv}$, introduced in [3]. The subscript lv stands for "*lazy value*", indicating the fact

[4] The full paper is available online at `http://ix.cs.uoregon.edu/~pdownen/classical-need-artifacts/`, including an appendix with supporting proofs and derivations, as well as code that implements the derivation presented here.

that focus goes to the term (or producer) in a lazy way. In other words, we first reduce the context as much as possible, mimicking outside-in evaluation in the lambda calculus. Then, we begin to reduce a term only when its context is irreducible. The syntax of $\overline{\lambda}_{lv}$ is defined as follows:

$$\begin{array}{rlrl} c \in \mathit{Command} & ::= \langle t \| e \rangle & e \in \mathit{Context} & ::= E \mid \tilde{\mu} x.c \\ t, u \in \mathit{Term} & ::= V \mid \mu \alpha.c & E \in \mathit{CoValue} & ::= \alpha \mid F \mid \tilde{\mu} x.C[\langle x \| F \rangle] \\ V \in \mathit{Value} & ::= x \mid \lambda x.t & F \in \mathit{ForcingContext} & ::= \boldsymbol{\alpha} \mid t \cdot E \\ & C \in \mathit{MetaContext} ::= \Box \mid \langle \mu \alpha.c \| \tilde{\mu} x.C \rangle & & \end{array}$$

A command connects a producer and a consumer together. A co-value E is an irreducible context, which is either a co-variable, a forcing context, or a term-binding context $\tilde{\mu}x.c$ in which the variable x has been forced. A forcing context, either a co-constant $\boldsymbol{\alpha}$ or an applicative context $t \cdot E$, drives computation forward, eagerly demanding a value. The form of applicative contexts is restricted from the general form $t \cdot e$. For example, $t \cdot \tilde{\mu}x.\langle x \| \boldsymbol{\alpha} \rangle$ is a valid application, whereas $t \cdot \tilde{\mu}x.\langle y \| \boldsymbol{\alpha} \rangle$ is not — in $\langle \mu\alpha.c \| t \cdot \tilde{\mu}x.\langle y \| \boldsymbol{\alpha} \rangle \rangle$, it forces evaluation of c even though its value is not needed.

The C in $\tilde{\mu}x.C[\langle x \| F \rangle]$ is a meta-context, which identifies the standard redex in a command. In a call-by-need sequent calculus, the next reduction is not necessarily at the top of the command, but may be buried under several bound computations $\mu\alpha.c$.

$\overline{\lambda}_{lv}$ reduction, written as $\rightarrow_{lv}$, denotes the compatible closure of the following rules: [5]

$$\begin{array}{rl} (\beta) & \langle \lambda x.t \| u \cdot E \rangle \rightarrow_{\beta} \langle u \| \tilde{\mu} x.\langle t \| E \rangle \rangle \\ (\tilde{\mu}_v) & \langle V \| \tilde{\mu} x.c \rangle \rightarrow_{\tilde{\mu}_v} c[V/x] \\ (\mu_l) & \langle \mu \alpha.c \| E \rangle \rightarrow_{\mu_l} c[E/\alpha] \end{array}$$

For all reductions in the paper, $\twoheadrightarrow$ is the reflexive transitive closure, $\twoheadrightarrow^{+}$ is the transitive closure, and $\rightarrow^{n/m}$ is a reduction sequence of n or m steps.

The β rule binds x to the argument and then proceeds with the evaluation of t in context E. Once the consumer is a co-value, focus goes to the producer. So for example, one has: $\langle \mu\beta.\langle I \| I \cdot \beta \rangle \| t \cdot \alpha \rangle \rightarrow_{lv} \langle I \| I \cdot t \cdot \alpha \rangle$. Notice that context switching also occurs in a command of the form $\langle \mu\beta.\langle I \| I \cdot \beta \rangle \| \tilde{\mu}x.\langle x \| F \rangle \rangle$. Unlike the call-by-need calculus of Ariola and Felleisen [1], values are substituted eagerly, and variables are values. In the command $\langle \mu\alpha.c \| \tilde{\mu}x.\langle x \| \tilde{\mu}z.\langle z \| F \rangle \rangle \rangle$, variable x is not demanded, and substituting x for z leads to the command $\langle \mu\alpha.c \| \tilde{\mu}x.\langle x \| F \rangle \rangle$ which now demands x.

The calculus presented here guarantees that a co-value is closed with respect to substitution. In the original calculus [3], $\tilde{\mu}x.\langle x \| \alpha \rangle$ is a co-value. However, if for example the context $\tilde{\mu}z.\langle z \| \delta \rangle$ is substituted for α, as in the reduction:

[5] For simplicity, we do not discuss the issue of explicit hygiene in this paper. For a discussion on maintaining hygiene in a call-by-need calculus, see [5], and for an explicitly hygienic implementation of the semantics presented here, see the supplemental code.

$\langle \mu\alpha.\langle \mu\alpha.t \| \tilde{\mu}x.\langle x \| \alpha \rangle \rangle \| \tilde{\mu}z.\langle z \| \delta \rangle \rangle \rightarrow_{lv} \langle \mu\alpha.t \| \tilde{\mu}x.\langle x \| \tilde{\mu}z.\langle z \| \delta \rangle \rangle \rangle$, then it is no longer a co-value since variable x is not needed — indeed x must first be substituted for z. The solution followed here is to restrict the notion of co-value until we have more information on the rest of the computation. Thus, a context of the form $\tilde{\mu}x.\langle x \| \alpha \rangle$ is not a co-value, because we do not know whether or not x will be forced.

The notion of weak head standard reduction is defined as

$$\frac{c \rightarrow_{\beta} c'}{C[c] \mapsto_{lv} C[c']} \qquad \frac{c \rightarrow_{\tilde{\mu}_v} c'}{C[c] \mapsto_{lv} C[c']} \qquad \frac{c \rightarrow_{\mu_l} c'}{C[c] \mapsto_{lv} C[c']}$$

A weak head normal form (whnf) for $\overline{\lambda}_{lv}$ is either $C[\langle \lambda x.t \| \boldsymbol{\alpha} \rangle]$ or $C[\langle z \| F \rangle]$ where z is not bound in C. The standardization is complete with respect to reduction: using standard proof techniques (see e.g. [2]), one can show that if $c \twoheadrightarrow_{lv} c'$ and c' is a whnf, then there exists a whnf c'' such that $c \mapsto\!\!\!\!\twoheadrightarrow_{lv} c''$ and $c'' \twoheadrightarrow_{lv} c'$.

3 A multicut sequent calculus ($\overline{\boldsymbol{\lambda}}_{[lv\tau]}$)

We now explore a calculus, which we call the multicut calculus, which keeps the standard redex at the top of a command and avoids searching through the meta-context for work to be done. We design the calculus in three steps. First, we introduce a syntactic notation for forced let-expressions ($\tilde{\mu}x.C[\langle x \| F \rangle]$) and the associated reduction rules. Second, we apply Danvy's [5] technique to systematically derive the associated abstract machine. Third, the multicut calculus comes out as a generalization of the abstract machine.

We modify the syntax of $\overline{\lambda}_{lv}$ by writing forced lets as $\tilde{\mu}[x].\langle x \| F \rangle \tau$ so that the forced command and its surrounding environment of bindings are kept separate, giving us $\overline{\lambda}_{[lv]}$. The separation between the command and its environment makes it explicit that the command is brought to the top of a forced let — there is no unbounded search for the command that was forced. The syntax of $\overline{\lambda}_{[lv]}$ is:

$$E \in CoValue ::= \alpha \mid F \mid \tilde{\mu}[x].\langle x \| F \rangle \tau$$
$$\tau \in Environment ::= \epsilon \mid [x = \mu\alpha.c]\tau$$

where c, t, V, e and F are defined as before. The relationship between the syntax of $\overline{\lambda}_{[lv]}$ and $\overline{\lambda}_{lv}$ includes:

$$\tilde{\mu}[x].\langle x \| F \rangle \tau \approx \tilde{\mu}x.\overline{\tau}[\langle x \| F \rangle]$$
$$\overline{\Box} = \epsilon \qquad \overline{\epsilon} = \Box$$
$$\overline{\langle \mu\alpha.c \| \tilde{\mu}x.C \rangle} = \overline{C}[x = \mu\alpha.c] \qquad \overline{\tau[x = \mu\alpha.c]} = \langle \mu\alpha.c \| \tilde{\mu}x.\overline{\tau} \rangle$$

where all other identical syntactic forms are related.

The $\overline{\lambda}_{[lv]}$ reduction system, in addition to the previous $\beta, \tilde{\mu}_v$ and μ_l rules, contains two new reductions:

$$(\tilde{\mu}_{[]}) \quad \langle \mu\alpha.c \| \tilde{\mu}x.C[\langle x \| F \rangle] \rangle \rightarrow_{[lv]} \langle \mu\alpha.c \| \tilde{\mu}[x].\langle x \| F \rangle \overline{C} \rangle$$
$$(\tilde{\mu}_{[v]}) \quad \langle V \| \tilde{\mu}[x].\langle x \| F \rangle \tau \rangle \rightarrow_{[lv]} (\overline{\tau}[\langle V \| F \rangle])[V/x]$$

Note that in a term of the form $\langle \mu\alpha.c \| \tilde{\mu}x.\langle \mu\beta.c' \| \tilde{\mu}z.\langle x \| F \rangle \rangle \rangle$, there is no context switch, meaning we do not substitute for α. First, the fact that x is needed is recorded via the rule $\tilde{\mu}_{[]}$ leading to: $\langle \mu\alpha.c \| \tilde{\mu}[x].\langle x \| F \rangle [z = \mu\beta.c'] \rangle$. Afterward, the μ_l rule applies, as in $\overline{\lambda}_{lv}$.

The weak head standard reduction of $\overline{\lambda}_{[lv]}$ is defined as before. It simulates the weak head standardization of $\overline{\lambda}_{lv}$.

Theorem 1. *Given c_1 from $\overline{\lambda}_{[lv]}$ and c_2 from $\overline{\lambda}_{lv}$ such that $c_1 \approx c_2$:*

- *If $c_2 \mapsto_{lv} c_2'$ then there exists c_1' such that $c_1 \mapsto_{[lv]}^{1/2} c_1'$ and $c_1' \approx c_2'$;*
- *If $c_1 \mapsto_{[lv]} c_1'$ then there exists c_2' such that $c_2 \mapsto_{lv}^{0/1} c_2'$ and $c_1' \approx c_2'$*

We use Danvy's technique for inter-deriving semantic artifacts in order to generate an abstract machine for $\overline{\lambda}_{[lv]}$. The first step in deriving an abstract machine from its calculus is to represent the operational semantics for that calculus as a small-step interpreter. To begin, we capture the standard reduction in a search function that, given a program, identifies the next redex to contract. By CPS transforming and then defunctionalizing the search function, the call stack of the searching procedure is reified as a data structure, which corresponds with the inside-out (meta-)context defining the standardization. The defunctionalized search function is extended into a decomposition function that splits a program into a meta-context and redex. Decomposition, contraction, and recomposition together define an iterative small-step interpreter and an operational semantics for the calculus. The next step in the process is to transform the iterative small-step interpreter into a mutually recursive big-step interpreter, which represents the abstract machine. First, the recomposition-decomposition step is deforested into a refocusing step. Rather than taking the contracted redex and recompose it into the full program, only to immediately decompose that program again, the search for the next redex starts from the current sub-program in focus via refocusing. Next, the iterative interpreter is fused to form a mutually recursive, tail-call interpreter. To finish the process, we compress corridor transitions, eliminate dead code, flatten program states, and convert the meta-context into a sequence of frames. The resulting big-step interpreter defines the abstract machine for the calculus. Applying this technique to the $\overline{\lambda}_{[lv]}$-calculus results in the following abstract machine (a state of the abstract machine is a command paired with an environment τ)

$$\begin{aligned}
\langle \lambda x.t \| u \cdot E \rangle \tau &\rightsquigarrow_{[lv]} \langle u \| \tilde{\mu}x.\langle t \| E \rangle \rangle \tau \\
\langle \mu\alpha.c_1 \| \tilde{\mu}x.c_2 \rangle \tau &\rightsquigarrow_{[lv]} c_2[x = \mu\alpha.c_1]\tau \\
\langle \mu\alpha.c \| E \rangle \tau &\rightsquigarrow_{[lv]} (c[E/\alpha])\tau \\
\langle V \| \tilde{\mu}x.c \rangle \tau &\rightsquigarrow_{[lv]} (c[V/x])\tau \\
\langle V \| \tilde{\mu}[x].\langle x \| F \rangle \tau' \rangle \tau &\rightsquigarrow_{[lv]} \langle V \| F[V/x] \rangle (\tau'[V/x])\tau \\
\langle x \| F \rangle \tau' [x = \mu\alpha.c] \tau &\rightsquigarrow_{[lv]} \langle \mu\alpha.c \| \tilde{\mu}[x].\langle x \| F \rangle \tau' \rangle \tau
\end{aligned}$$

Since the abstract machine was derived from the standardization of $\overline{\lambda}_{[lv]}$ through correctness-preserving transformations, the two correspond directly.

Theorem 2. *Given a $\overline{\lambda}_{[lv]}$ command c,*

- *If $c \mapsto_{[lv]} c'$ then for all c_1, τ where $c = \overline{\tau}[c_1]$, there exists c_1', τ' such that $c_1\tau \rightsquigarrow^{+}_{[lv]} c_1'\tau'$ and $\overline{\tau'}[c_1'] = c'$.*
- *If $c\tau \rightsquigarrow_{[lv]} c'\tau'$ then $\overline{\tau}[c] \mapsto^{0/1}_{[lv]} \overline{\tau'}[c']$.*

We generalize the abstract machine for $\overline{\lambda}_{[lv]}$ into a variant of the original calculus, which we call $\overline{\lambda}_{[lv\tau]}$. The $\overline{\lambda}_{[lv\tau]}$-calculus can express call-by-need reduction without the use of a meta-context. Each command is coupled with an environment of unevaluated computations, bringing the standard redex back to the top of the command. The syntax of $\overline{\lambda}_{[lv\tau]}$ contains the new syntactic category of *Closures* (ranged over the meta-variable l) and is defined as:

$$\begin{array}{rlrl}
l \in \mathit{Closure} & ::= c\tau & \tau \in \mathit{Environment} & ::= \epsilon \mid [x = \mu\alpha.l]\tau \\
c \in \mathit{Command} & ::= \langle t \| e \rangle & e \in \mathit{Context} & ::= E \mid \tilde{\mu}x.l \\
t \in \mathit{Term} & ::= V \mid \mu\alpha.l & E \in \mathit{CoValue} & ::= \alpha \mid F \mid \tilde{\mu}[x].\langle x \| F \rangle\tau \\
V \in \mathit{Value} & ::= x \mid \lambda x.t & F \in \mathit{ForcingContext} & ::= \boldsymbol{\alpha} \mid t \cdot E
\end{array}$$

Reductions in $\overline{\lambda}_{[lv\tau]}$ are generalizations of the steps in the abstract machine for $\overline{\lambda}_{[lv]}$, with the ability to apply reductions anywhere in a closure.

$$\begin{array}{rc}
(\beta) & \langle \lambda x.t \| u \cdot E \rangle\tau \to_{[lv\tau]} \langle u \| \tilde{\mu}x.\langle t \| E \rangle\epsilon \rangle\tau \\
(\tilde{\mu}_\tau) & \langle \mu\alpha.l \| \tilde{\mu}x.c\tau' \rangle\tau \to_{[lv\tau]} c\tau'[x = \mu\alpha.l]\tau \\
(\mu_l) & \langle \mu\alpha.l \| E \rangle\tau \to_{[lv\tau]} l[E/\alpha]\tau \\
(\tilde{\mu}_v) & \langle V \| \tilde{\mu}x.l \rangle\tau \to_{[lv\tau]} l[V/x]\tau \\
(\tilde{\mu}_{[v]}) & \langle V \| \tilde{\mu}[x].l \rangle\tau \to_{[lv\tau]} l[V/x]\tau \\
(\tilde{\mu}_{[]}) & \langle x \| F \rangle\tau'[x = \mu\alpha.l]\tau \to_{[lv\tau]} \langle \mu\alpha.l \| \tilde{\mu}[x].\langle x \| F \rangle\tau' \rangle\tau
\end{array}$$

Note that $l\tau$, for $l = c\tau'$, is defined as $c\tau_1$ where τ_1 is the concatenation of τ' and τ.

Proposition 1. *$\overline{\lambda}_{[lv\tau]}$ is confluent.*

We trivially have a weak head standard reduction for $\overline{\lambda}_{[lv\tau]}$ in which closures are reduced in the empty meta-context: $l \mapsto_{[lv\tau]} l'$ if $l \to_{[lv\tau]} l'$. A whnf for $\overline{\lambda}_{[lv\tau]}$ is either $\langle \lambda x.t \| \boldsymbol{\alpha} \rangle\tau$ or $\langle z \| F \rangle\tau$ where z is free in τ. The standardization of $\overline{\lambda}_{[lv\tau]}$ operates in lock-step with the abstract machine for $\overline{\lambda}_{[lv]}$, and so it is also complete with respect to reduction, where closures and commands are related by $c\tau \approx \overline{\tau}[c]$ and environments are related if they bind the same variables in the same order to related terms.

Theorem 3. *Given c_1, τ_1 from $\overline{\lambda}_{[lv\tau]}$ and c_2, τ_2 from $\overline{\lambda}_{[lv]}$ such that $c_1 \approx c_2$ and $\tau_1 \approx \tau_2$:*

- *If $c_1\tau_1 \mapsto_{[lv\tau]} l_1$ then there exists $l_1'\tau_1' = l_1$ and c_2', τ_2' such that $c_2\tau_2 \rightsquigarrow_{[lv]} c_2'\tau_2'$, $l_1' \approx c_2'$, and $\tau_1' \approx \tau_2'$.*

- *If* $c_2\tau_2 \rightsquigarrow_{[lv]} c_2'\tau_2'$ *then there exists* l_1', τ_1' *such that* $c_1\tau_1 \mapsto_{[lv\tau]} l_1'\tau_1'$, $l_1' \approx c_2'$, *and* $\tau_1' \approx \tau_2'$.

Remark 1. Using a special syntactic form to remember that a variable is needed makes the connection to the final abstract machine and continuation-passing style transformation more direct. However, if one is interested in the multicut *per se* then the calculus can be simplified by abandoning the special syntax for forced lets and performing a context switch when a co-variable and a forcing context F is encountered. We call the resulting calculus $\overline{\lambda}_{lv\tau}$ — its reduction theory contains β, $\tilde{\mu}_v$, $\tilde{\mu}_\tau$, and μ_l, along with the following replacement for $\tilde{\mu}_{[]}$:

$$\langle x \| F \rangle \tau'[x = \mu\alpha.l]\tau \to_{lv\tau} l[\tilde{\mu}x.\langle x \| F \rangle \tau'/\alpha]\tau$$

However, by removing the special annotation that explicitly marks lets that have been forced, there is some ambiguity in parsing the language. We can interpret a context of the form $\tilde{\mu}x.\langle x \| F \rangle \tau$ as either a forced let, since its command forces the bound variable, or instead as some general let that just happens to have the command $\langle x \| F \rangle$ in focus. Although the two different readings lead to two different reductions, they quickly converge. The μ_l-reduction applies if we interpet the context as a co-value. Otherwise, we have the reduction sequence of $\tilde{\mu}_\tau$ followed by $\tilde{\mu}_{[]}$, which brings us to the same closure.

$$\langle \mu\alpha.l \| \tilde{\mu}x.\langle x \| F \rangle \tau' \rangle \tau \xrightarrow{\mu_l}_{lv\tau} l[\tilde{\mu}x.\langle x \| F \rangle \tau'/\alpha]\tau$$

$$\langle \mu\alpha.l \| \tilde{\mu}x.\langle x \| F \rangle \tau' \rangle \tau \xrightarrow{\tilde{\mu}_\tau}_{lv\tau} \langle x \| F \rangle \tau'[x = \mu\alpha.l]\tau \xrightarrow{\tilde{\mu}_{[]}}_{lv\tau} l[\tilde{\mu}x.\langle x \| F \rangle \tau'/\alpha]\tau$$

To illustrate the usefulness of a multicut calculus (either $\overline{\lambda}_{lv\tau}$ or $\overline{\lambda}_{[lv\tau]}$) we show that Example 2 terminates and produces (I, I) as a result, where the reductions do not need an unbounded search for the standard redex.

$$\begin{aligned}
&\langle \mu\alpha.\langle (I, \lambda x.\mu_.\langle x \| \alpha \rangle) \| \alpha \rangle \| \tilde{\mu}a.\langle \mu\beta.\langle a \| \mathsf{fst} \cdot \beta \rangle \| \tilde{\mu}f.\langle \mu\delta.\langle a \| \mathsf{snd} \cdot \delta \rangle \| \tilde{\mu}q.\langle f \| q \cdot (I, I) \cdot \mathbf{tp} \rangle \rangle \rangle \rangle \\
&\mapsto\!\!\!\!\to \langle f \| q \cdot (I, I) \cdot \mathbf{tp} \rangle [q = \mu\delta.\langle a \| \mathsf{snd} \cdot \delta \rangle][f = \mu\beta.\langle a \| \mathsf{fst} \cdot \beta \rangle][a = \mu\alpha.\langle (I, \lambda x.\mu_.\langle x \| \alpha \rangle) \| \alpha \rangle] \\
&\mapsto\!\!\!\!\to \langle (I, \lambda x.\mu_.\langle x \| \alpha \rangle) \| \alpha \rangle \quad \textbf{where } \alpha = \tilde{\mu}[a].\langle a \| \mathsf{fst} \cdot \tilde{\mu}[f].\langle f \| q \cdot (I, I) \cdot \mathbf{tp} \rangle [q = \mu\delta.\langle a \| \mathsf{snd} \cdot \delta \rangle] \rangle \\
&\mapsto\!\!\!\!\to \langle q \| (I, I) \cdot \mathbf{tp} \rangle [q = \mu\delta.\langle (I, \lambda x.\mu_.\langle x \| \alpha \rangle) \| \mathsf{snd} \cdot \delta \rangle] \\
&\mapsto\!\!\!\!\to \langle (I, I) \| \alpha \rangle \quad \textbf{where } \alpha = \tilde{\mu}[a].\langle a \| \mathsf{fst} \cdot \tilde{\mu}[f].\langle f \| q \cdot (I, I) \cdot \mathbf{tp} \rangle [q = \mu\delta.\langle a \| \mathsf{snd} \cdot \delta \rangle] \rangle \\
&\mapsto\!\!\!\!\to \langle q \| (I, I) \cdot \mathbf{tp} \rangle [q = \mu\delta.\langle (I, I) \| \mathsf{snd} \cdot \delta \rangle] \\
&\mapsto\!\!\!\!\to \langle (I, I) \| \mathbf{tp} \rangle
\end{aligned}$$

Notice that the second time α is reduced, q starts fresh from its initial unevaluated computation and can see the change in a.

4 Environment-based abstract machine ($\overline{\lambda}_{[lv\tau*]}$)

In order to construct a more efficient abstract machine, we need to avoid performing the costly substitution operation. To this end, we modify $\overline{\lambda}_{[lv\tau]}$ so that all substitutions are instead stored in the environment τ, giving $\overline{\lambda}_{[lv\tau*]}$:

$$\tau \in \mathit{Environment} ::= \epsilon \mid [x = t]\tau \mid [\alpha = E]\tau$$

The modified reductions for $\overline{\lambda}_{[lv\tau*]}$ are:

$$\begin{array}{rc}
(\beta) & \langle \lambda x.t \| u \cdot E\rangle \rightarrow_{[lv\tau*]} \langle u \| \tilde{\mu} x.\langle t \| E\rangle \epsilon\rangle \\
(\tilde{\mu}_\tau) & \langle t \| \tilde{\mu} x.c\tau'\rangle\tau \rightarrow_{[lv\tau*]} c\tau'[x = t]\tau \\
(\mu_l) & \langle \mu\alpha.c\tau' \| E\rangle\tau \rightarrow_{[lv\tau*]} c\tau'[\alpha = E]\tau \\
(\tilde{\mu}_{[v]}) & \langle V \| \tilde{\mu}[x].\langle x \| F\rangle\tau'\rangle\tau \rightarrow_{[lv\tau*]} \langle V \| F\rangle\tau'[x = V]\tau \\
(\tilde{\mu}_{[]}) & \langle x \| F\rangle\tau'[x = t]\tau \rightarrow_{[lv\tau*]} \langle t \| \tilde{\mu}[x].\langle x \| F\rangle\tau'\rangle\tau \\
(\tau_\alpha) & \langle V \| \alpha\rangle\tau'[\alpha = E]\tau \rightarrow_{[lv\tau*]} \langle V \| E\rangle\tau'[\alpha = E]\tau
\end{array}$$

The standard reduction of $\overline{\lambda}_{[lv\tau*]}$ is performed in the empty meta-context, as in $\overline{\lambda}_{[lv\tau]}$. Closures in $\overline{\lambda}_{[lv\tau*]}$ relate to closures in $\overline{\lambda}_{[lv\tau]}$ by performing substitution on values and co-values stored in the environment.

$$l[\alpha = E] \approx l[E/\alpha] \qquad l[x = V] \approx l[V/x] \qquad l[x = \mu\alpha.c] \approx l[x = \mu\alpha.c]$$

Theorem 4. *Given a $\overline{\lambda}_{[lv\tau*]}$ closure l_1 and a $\overline{\lambda}_{[lv\tau]}$ closure l_2 such that $l_1 \approx l_2$:*

- *If $l_2 \mapsto_{[lv\tau]} l_2'$ then $l_1 \mapsto\!\!\!\!\rightarrow^{+}_{[lv\tau*]} l_1'$.*
- *If $l_1 \mapsto_{[lv\tau*]} l_1'$ then there exists l_1'', l_2' such that $l_2 \mapsto\!\!\!\!\rightarrow^{0/1}_{[lv\tau]} l_2'$ and $l_1' \mapsto\!\!\!\!\rightarrow^{0/1}_{[lv\tau*]} l_1''$ and $l_1'' \approx l_2'$.*

The standardization of $\overline{\lambda}_{[lv\tau*]}$ gives rise to the following abstract machine.

$$\begin{array}{rc}
\langle t \| \tilde{\mu} x.c\rangle\tau & \rightsquigarrow_{[lv*]} c[x = t]\tau \\
\langle \mu\alpha.c \| E\rangle\tau & \rightsquigarrow_{[lv*]} c[\alpha = E]\tau \\
\langle V \| \alpha\rangle\tau'[\alpha = E]\tau & \rightsquigarrow_{[lv*]} \langle V \| E\rangle\tau'[\alpha = E]\tau \\
\langle V \| \tilde{\mu}[x].\langle x \| F\rangle\tau'\rangle\tau & \rightsquigarrow_{[lv*]} \langle V \| F\rangle\tau'[x = V]\tau \\
\langle x \| F\rangle\tau'[x = t]\tau & \rightsquigarrow_{[lv*]} \langle t \| \tilde{\mu}[x].\langle x \| F\rangle\tau'\rangle\tau \\
\langle \lambda x.t \| u \cdot E\rangle\tau & \rightsquigarrow_{[lv*]} \langle u \| \tilde{\mu} x.\langle t \| E\rangle\rangle\tau
\end{array}$$

Theorem 5. *Given c_1, τ_1 from $\overline{\lambda}_{[lv\tau*]}$ and c_2, τ_2 from $\overline{\lambda}_{[lv\tau]}$ such that $c_1 \approx c_2$ and $\tau_1 \approx \tau_2$:*

- *If $c_1\tau_1 \mapsto_{[lv\tau*]} l_1$ then there exists $l_1'\tau_1' = l_1$ and c_2', τ_2' such that $c_2\tau_2 \rightsquigarrow_{[lv*]} c_2'\tau_2'$, $l_1' \approx c_2'$, and $\tau_1' \approx \tau_2'$.*
- *If $c_2\tau_2 \rightsquigarrow_{[lv*]} c_2'\tau_2'$ then there exists l_1', τ_1' such that $c_1\tau_1 \mapsto_{[lv\tau*]} l_1'\tau_1'$, $l_1' \approx c_2'$, and $\tau_1' \approx \tau_2'$.*

Unlike the abstract machine for $\overline{\lambda}_{[lv]}$, the above abstract machine is context-free, since at each step a decision can be made by examining either the term or the context in isolation. To make this structure more apparent, we divide the machine into a number of context-free phases in Figure 1. Each phase only analyzes one component of the command, the "active" term or context, and is parametric in the other "passive" component. In essence, for each phase of the machine, either the term or the context is *fully in control* and *independent*, regardless of what the other half happens to be.

$$c \rightsquigarrow_{[lv*]} c_e\epsilon$$

$$\begin{array}{c}\langle t \| \tilde{\mu}x.c\rangle_e\tau \rightsquigarrow_{[lv*]} c_e[x=t]\tau \\ \langle t\|E\rangle_e\tau \rightsquigarrow_{[lv*]} \langle t\|E\rangle_t\tau\end{array}$$

$$\begin{array}{c}\langle \mu\alpha.c \| E\rangle_t\tau \rightsquigarrow_{[lv*]} c_e[\alpha=E]\tau \\ \langle V\|E\rangle_t\tau \rightsquigarrow_{[lv*]} \langle V\|E\rangle_E\tau\end{array}$$

$$\begin{array}{c}\langle V\|\alpha\rangle_E\tau'[\alpha=E]\tau \rightsquigarrow_{[lv*]} \langle V\|E\rangle_E\tau'[\alpha=E]\tau \\ \langle V\|\tilde{\mu}[x].\langle x\|F\rangle\tau'\rangle_E\tau \rightsquigarrow_{[lv*]} \langle V\|F\rangle_V\tau'[x=V]\tau \\ \langle V\|F\rangle_E\tau \rightsquigarrow_{[lv*]} \langle V\|F\rangle_V\tau\end{array}$$

$$\begin{array}{c}\langle x\|F\rangle_V\tau'[x=t]\tau \rightsquigarrow_{[lv*]} \langle t\|\tilde{\mu}[x].\langle x\|F\rangle\tau'\rangle_t\tau \\ \langle \lambda x.t\|F\rangle_V\tau \rightsquigarrow_{[lv*]} \langle \lambda x.t\|F\rangle_F\,\tau\end{array}$$

$$\langle \lambda x.t \| u\cdot E\rangle_F\tau \rightsquigarrow_{[lv*]} \langle u\|\tilde{\mu}x.\langle t\|E\rangle\rangle_e\tau$$

Fig. 1. Abstract machine for the classical call-by-need sequent calculus $\overline{\lambda}_{[lv\tau*]}$.

5 Environment-based CPS ($\overline{\lambda}_{[lv\tau*]}$)

Having an abstract machine in context-free form is good for more than just aesthetic reasons. Having both the term and context behave independently of each other makes the machine amendable to conversion into a nice CPS transform. Since a CPS transform is a compilation from the source language into the λ-calculus, each expression must have meaning independent of its surroundings. During translation, an expression may take a continuation as a parameter, but it cannot directly examine it — the continuation is a black box that can only be entered by yielding all control to it. The parametric nature of the steps in Figure 1, and the fact that each syntactic form is analyzed exactly once in the machine, means that we can directly derive a CPS transform that corresponds exactly with the machine.

Deriving a CPS translation from a context-free abstract machine is more straightforward than deriving a machine from a calculus. Starting from the big-step interpreter, in each phase, the case analysis is lifted out so that the interpreter becomes a set of one-argument functions on the active expression that produce a function accepting the passive expression as an extra parameter. Then, the syntax is *refunctionalized*: rather than pass the syntactic forms as-is to future stages of the interpreter, each syntactic form is *immediately* given to the interpreter as they become available. The partial evaluation of the interpreter with only the active argument becomes a continuation waiting for the passive

$$[\![\langle t \| e \rangle]\!]_c = [\![e]\!]_e\ [\![t]\!]_t$$

$$[\![\tilde{\mu} x.c]\!]_e\ t = \lambda\tau.[\![c]\!]_c\ ([x = t]\tau)$$
$$[\![E]\!]_e\ t = t\ [\![E]\!]_E$$

$$[\![\mu\alpha.c]\!]_t\ E = [\![c]\!]_c[E/\alpha]$$
$$[\![V]\!]_t\ E = E\ [\![V]\!]_V$$

$$[\![\alpha]\!]_E\ V = \alpha\ V$$
$$[\![\tilde{\mu}[x].\langle x \| F \rangle \tau']\!]_E\ V = \lambda\tau.[\![\tau']\!]_\tau\ V\ [\![F]\!]_F\ ([x = (\lambda E.E\ V)]\tau)$$
$$[\![F]\!]_E\ V = V\ [\![F]\!]_F$$

$$[\![x]\!]_V\ F = \lambda\tau.\tau(x)\ F$$
$$[\![\lambda x.t]\!]_V\ F = F\ (\lambda u.\lambda E.\lambda\tau.[\![t]\!]_t\ E\ ([x = u]\tau))$$

$$[\![\boldsymbol{\alpha}]\!]_F\ v = \boldsymbol{\alpha}\ v$$
$$[\![u \cdot E]\!]_F\ v = v\ [\![u]\!]_t\ [\![E]\!]_E$$

$$[\![\epsilon]\!]_\tau\ V\ F = \lambda\tau.V\ F\ \tau$$
$$[\![\tau'[x = t]]\!]_\tau\ V\ F = \lambda\tau.[\![\tau']\!]_\tau\ V\ F\ ([x = [\![t]\!]_t]\tau)$$
$$[\![\tau'[\alpha = E]]\!]_\tau\ V\ F = \lambda\tau.([\![\tau']\!]_\tau\ V\ F)\ [[\![E]\!]_E/\alpha]\tau$$

$$(\tau'[x = t]\tau)(x) = \lambda F.t\ (\lambda V.\lambda\tau.V\ F\ (\tau'[x = \lambda E.E\ V]\tau))\ \tau$$

Fig. 2. Continuation and environment passing style transform for $\overline{\lambda}_{[lv\tau*]}$.

counterpart. Finally, since co-variables in the environment are treated as ordinary static bindings, we convert context bindings back to implicit substitution. With all syntactic forms interpreted on-site, the resulting interpreter is a CPS transformation from the calculus into the host language.

From the abstract machine in Figure 1, we derive an environment-based CPS transformation for $\overline{\lambda}_{[lv\tau*]}$, given in Figure 2. The concrete representation of environments used in the transform is left abstract — we assume that environments can be extended, joined, and split on a variable. The three-way split is the same one that appears in the abstract machine: for a given variable x, the environment is partitioned into all bindings before the first occurrence of x, the binding of x itself, and all remaining bindings.

$$
\begin{aligned}
[\![x]\!]\ k &= x\ k \\
[\![\lambda x.t]\!]\ k &= k\ (\lambda x.[\![t]\!]) \\
[\![t_1\ t_2]\!]\ k &= [\![t_1]\!]\ (\lambda f.\,\mathbf{let}\ r = \mathsf{delay}\,[\![t_2]\!]\ \mathbf{in}\ f\ (\lambda k.!r\ k)\ k) \\
[\![\mu\alpha.J]\!]\ k &= [\![J]\!][k/\alpha] \\
[\![[\alpha]t]\!] &= [\![t]\!]\ \alpha
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{delay}\ t &= \mathbf{new}\ r\ \mathbf{in}\ r := (\lambda k.\,\mathsf{force}_r\ t\ k); r \\
\mathsf{force}_r\ t\ k &= t\ (\lambda v.r := (\lambda k'.k'\ v); k\ v)
\end{aligned}
$$

Fig. 3. Okasaki *et al.*'s CPS transformation extended with control.

Since the CPS transform in Figure 2 was derived from the abstract machine in Figure 1 by correctness-preserving transformations, the two correspond directly.

Theorem 6. *If* $c\tau \rightsquigarrow_{[lv*]} c'\tau'$ *then* $[\![c]\!][\![\tau]\!] =_{\beta\eta} [\![c']\!][\![\tau']\!]$.

Store-based CPS Another approach to deriving a CPS for a classical call-by-need calculus is to extend the CPS given by Okasaki *et al.* [8] with control operators. We do so by adding the operators μ and "bracket" from Parigot's $\lambda\mu$-calculus [9] for respectively capturing and re-installing an evaluation context. This extension corresponds to applying a store passing transformation to the transformation given in Figure 3. The natural extension of the delay and force CPS results in a different semantics than the one in Figure 2. In particular, going back to Example 2 and using the store-based semantics in Figure 3, we get the following reduction (where a program is a term t and a store s written $t \parallel s$):

$$
\begin{aligned}
&\mathbf{let}\ a = \mu\alpha.[\alpha](I, \lambda x.\mu\_.[\alpha]x)\ \mathbf{in}\ \mathbf{let}\ f = \mathsf{fst}\,p\ \mathbf{in}\ \mathbf{let}\ q = \mathsf{snd}\,p\ \mathbf{in}\ f\ q\ (I,I) \parallel \epsilon \\
&\twoheadrightarrow f\ q\ (I,I) \parallel [q = \mathsf{force}_q\ \mathsf{snd}\,p][f = \mathsf{force}_p\ \mathsf{fst}\,p][a = \mathsf{force}_a\ \mu\alpha.[\alpha](I, \lambda x.\mu\_.[\alpha]x)] \\
&\twoheadrightarrow (\mathsf{force}_f\ \mathsf{fst}(\mathsf{force}_a\ \mu\alpha.[\alpha](I, \lambda x.\mu\_.[\alpha]x)))\ q\ (I,I) \\
&\quad \parallel [q = \mathsf{force}_q\ \mathsf{snd}\,p][f = \mathsf{force}_f\ \mathsf{fst}\,p][a = \mathsf{force}_a\ \mu\alpha.[\alpha](I, \lambda x.\mu\_.[\alpha]x)] \\
&\twoheadrightarrow (\mathsf{force}_f\ \mathsf{fst}(\mathsf{force}_a(I, \lambda x.\mu\_.(\mathsf{force}_f\ \mathsf{fst}(\mathsf{force}_a\ x))\ q\ (I,I))))\ q\ (I,I) \\
&\quad \parallel [q = \mathsf{force}_q\ \mathsf{snd}\,p][f = \mathsf{force}_f\ \mathsf{fst}\,p][a = \mathsf{force}_a\ \mu\alpha.[\alpha](I, \lambda x.\mu\_.[\alpha]x)] \\
&\twoheadrightarrow q\ (I,I) \parallel [q = \mathsf{force}_q\ \mathsf{snd}\,p][f = I][a = (I, \lambda x.\mu\_.(\mathsf{force}_f\ \mathsf{fst}(\mathsf{force}_a\ x))\ q\ (I,I))] \\
&\twoheadrightarrow (\lambda x.\mu\_.(\mathsf{force}_f\ \mathsf{fst}(\mathsf{force}_a\ x))\ q\ (I,I))\ (I,I) \\
&\quad \parallel [q = \lambda x.(\mathsf{force}_f\ \mathsf{fst}(\mathsf{force}_a\ x))\ q\ (I,I)][f = I][a = (I, \lambda x.\mu\_.(\mathsf{force}_f\ \mathsf{fst}(\mathsf{force}_a\ x))\ q\ (I,I))] \\
&\twoheadrightarrow q\ (I,I) \parallel [q = \lambda x.\mu\_.(\mathsf{force}_f\ \mathsf{fst}(\mathsf{force}_a\ x))\ q\ (I,I)][f = I][a = (I,I)] \\
&\twoheadrightarrow \ldots
\end{aligned}
$$

We saw that using the multicut calculus $\overline{\lambda}_{[lv\tau]}$, the program produces (I, I) as a result — every time the continuation k is invoked, both f and q are reverted to their unevaluated states. However, with the semantics in Figure 3, the continuation bound to k captures only the forcing of f and a. Since q was not involved

in the dereference chain that triggered evaluation of callcc, the thunk bound to q is completely ignored in k. This means that once the thunk bound to q is reduced to **fn** x $\Rightarrow$ throw k x, it retains that value for every invocation of k. Since the value of q never changes, the program will loop forever.

The discrepancy witnessed between our semantics and a store-based semantics raises the concern that our call-by-need sequent calculus does not accurately model sharing, even in the minimal restriction ($\overline{\lambda}_{mlv}$). However, that is not the case. In [3], we have presented the natural deduction counterpart of $\overline{\lambda}_{mlv}$, (λ_{need}) and showed that it is sound and complete for evaluation to answers [6] with respect to the standard reduction of Ariola and Felleisen calculus (λ_{let}) [1].

6 On variables as values

By now, we can see that defining a classical call-by-need calculus is fraught with non-trivial design decisions. We briefly outline an alternative call-by-need calculus in which variables are not values. Declaring that variables are not values would seem to entail that a context of the form $\tilde{\mu}x.\langle y \| \tilde{\mu}z.\langle x \| E\rangle\rangle$ is a co-value demanding x. Both x and y are not substitutable since they are not values, so the co-term is still a co-value even if E is a $\tilde{\mu}$-binding. However we run into the problem of having co-values not closed with respect to substitution. If one substitutes a λ-abstraction V for y, obtaining $\tilde{\mu}x.\langle V \| \tilde{\mu}z.\langle x \| E\rangle\rangle$, then we no longer have a co-value since it contains a new redex — we can now substitute V for z. In Section 2, we introduced a distinction between an evaluation context and a forcing context. We could apply the same idea here by distinguishing variables from computations. Intuitively, $\tilde{\mu}x.\langle y \| \tilde{\mu}z.\langle x \| E\rangle\rangle$ is not a co-value because one needs to know more about y. However, we can adopt a different solution: we do not perform the substitution eagerly but instead only dereference values on a by-need basis. By choosing to define variables as non-values, and only dereferencing bound values, we arrive at a much simpler grammar for the language, where c and e are unchanged.

$$t \in Term ::= V \mid x \mid \mu\alpha.c \qquad E \in CoValue ::= \alpha \mid t \cdot E \mid \tilde{\mu}x.C[\langle x \| E\rangle]$$
$$V \in Value ::= \lambda x.t \qquad C \in MetaContext ::= \Box \mid \langle t \| \tilde{\mu}x.C\rangle$$

The cost of these decisions, however, is that we must redefine $\tilde{\mu}_v$ reduction to dereference values bound to variables only when it is absolutely necessary to move computation forward.

$$(\tilde{\mu}_v) \quad \langle V \| \tilde{\mu}x.C[\langle x \| E\rangle]\rangle \to \langle V \| \tilde{\mu}x.C[\langle V \| E\rangle]\rangle$$

Deriving the abstract machine and CPS transformation for this alternate calculus follows the same basic procedure used with $\overline{\lambda}_{lv}$. [7] One difference to

[6] An answer is a λ-abstraction or a let expression whose body is an answer.

[7] The full derivation for the alternate calculus is available at `http://ix.cs.uoregon.edu/~pdownen/classical-need-artifacts/`.

note, however, is that since this calculus uses a delayed by-need substitution, the derived abstract machine is already context free. By starting with a calculus that uses delayed, dereference-based substitution, generating the CPS transformation is simpler and more direct. The more direct derivation gives a closer relationship between the source calculus and the final CPS transformation.

It is interesting to note that the decision of whether or not variables are considered values has a non-trivial impact on the resulting abstract machine and CPS transformation. The definition of variables in a call-by-need language has an inherent tension — both formulations have their own complications. Variables can be thought of as values, since they stand in for values that may or may not have been computed yet and can be safely copied throughout a program without duplicating work. However, treating variables as values complicates the notion of "forcing" a computation, which shows up in the grammar of contexts. This also requires an extra push to drive computation forward, which was given in the form of co-constants $\boldsymbol{\alpha}$. A program like $\langle \mu\beta.c \| \tilde{\mu}x.\langle x \| \alpha \rangle \rangle$ does not reduce any further since x is a value, even though x is bound to a delayed computation. On the other hand, variables can be thought of as non-values, since they represent a reference to a potentially delayed computation. In this case, driving computation forward is trivial since any non-$\tilde{\mu}$ context demands a value. When x is not considered a value, the program $\langle \mu\beta.c \| \tilde{\mu}x.\langle x \| \alpha \rangle \rangle$ will demand a value for x regardless of what may be substituted for α. Instead, the complication shows up during substitution of values. When a value is substituted for a variable, suddenly a non-value term is replaced with a value. This has intricate interactions with the evaluation context of a program and makes substitution for values a non-trivial operation.

7 Conclusion

In this paper, we demonstrate the usefulness of having a systematic approach for dealing with syntactic theories. Semantics for a language can be presented in different ways, and the semantic artifact that comes from a particular presentation carries with it certain strengths and weaknesses. A standard reduction is useful for reasoning directly about the language, an abstract machine is well-suited as a basis for an efficient implementation, and a CPS transformation provides a theory in terms of the λ-calculus. Since these three forms of semantics are closely intertwined, defining any one of them inherently defines the others — generating the remaining artifacts becomes a straightforward exercise. A systematic approach liberates the language designer from the burden of hand-crafting each semantic artifact from the ground up.

It is interesting to find that the most "natural" extension of call-by-need with control changes depending on how the problem is approached. This development shows an interesting case of the tension between theory and practice. By approaching the problem with the sequent calculus as a reference point and taking the path of least resistance, we arrive at the theory developed in this paper. The resulting semantics comes with an elegant reduction theory, but it is not obvious how to efficiently map the abstract machine to modern computer

hardware. On the other hand, call-by-need is generally implemented with delay and force in practice, and performing the obvious extension leads to a different semantics. The abstract machine that comes from this alternate semantics is easy to efficiently implement in hardware, but the semantics is harder to reason about. As future work, it will be interesting to explore a reduction theory for the store-based semantics and an efficient implementation for the environment-based semantics.

Acknowledgments We are indebted to Olivier Danvy for his many fruitful discussions on the close connection between semantic artifacts. Paul Downen and Zena M. Ariola have been supported by NSF grant CCF-0917329. Keiko Nakata's research was supported by the European Regional Development Fund (ERDF) through the Estonian Centre of Excellence in Computer Science (EXCS), and the Estonian Science Foundation grant no. 9398. This research has also been supported by the INRIA Équipe Associée SEMACODE.

References

1. Z. M. Ariola and M. Felleisen. The call-by-need lambda calculus. *J. Funct. Program.*, 7(3):265–301, 1997.
2. Z. M. Ariola and H. Herbelin. Control reduction theories: the benefit of structural substitution. *J. Funct. Program.*, 18(3):373–419, 2008.
3. Z. M. Ariola, H. Herbelin, and A. Saurin. Classical call-by-need and duality. In *Typed Lambda Calculi and Applications*, volume 6690 of *lncs*, 2011.
4. O. Danvy. From reduction-based to reduction-free normalization. In *Proceedings of AFP'08*, Berlin, Heidelberg, 2009. Springer-Verlag.
5. O. Danvy, K. Millikin, J. Munk, and I. Zerny. Defunctionalized interpreters for call-by-need evaluation. In *Functional and Logic Programming, FLOPS2010*, 2010.
6. R. Garcia, A. Lumsdaine, and A. Sabry. Lazy evaluation and delimited control. In *Proceedings of POPL '09*, pages 153–164, New York, NY, USA, 2009. ACM.
7. J. Maraist, M. Odersky, and P. Wadler. The call-by-need λ-calculus. *J. Funct. Program.*, 8(3):275–317, 1998.
8. C. Okasaki, P. Lee, and D. Tarditi. Call-by-need and continuation-passing style. In *Lisp and Symbolic Computation*, pages 57–81. Kluwer Academic Publishers, 1993.
9. M. Parigot. Lambda-mu-calculus: An algorithmic interpretation of classical natural deduction. In *LPAR 92*, pages 190–201. Springer-Verlag, 1992.