First Class Call Stacks: Exploring Head Reduction

Philip Johnson-Freyd, Paul Downen and Zena M. Ariola
University of Oregon
Oregon, USA
{philipjf,pdownen,ariola}@cs.uoregon.edu

Weak-head normalization is inconsistent with functional extensionality in the call-by-name λ-calculus. We explore this problem from a new angle via the conflict between extensionality and effects. Leveraging ideas from work on the λ-calculus with control, we derive and justify alternative operational semantics and a sequence of abstract machines for performing head reduction. Head reduction avoids the problems with weak-head reduction and extensionality, while our operational semantics and associated abstract machines show us how to retain weak-head reduction's ease of implementation.

1 Introduction

Programming language designers are faced with multiple, sometimes contradictory, goals. On the one hand, it is important that users be able to reason about their programs. On the other hand, we want our languages to support simple and efficient implementations. For the first goal, *extensionality* is a particularly desirable property. We should be able to use a program without knowing how it was written, only how it behaves. In the λ-calculus, extensional reasoning is partially captured by the η law, a strong equational property about functions which says that functional delegation is unobservable: $\lambda x.\, f\ x = f$. It is essential to many proofs about functional programs: for example, the well-known "state monad" only obeys the monad laws if η holds [13]. The η law is compelling because it gives the "maximal" extensionality possible in the untyped λ-calculus: if we attempt to equate any additional terms beyond β, η, and α, the theory will collapse [4]. For the second goal, it is important to have normal forms, specifying the possible results of execution, that can be efficiently computed. To that end, most implementations stop execution when they encounter a lambda-abstraction. This is called *weak-head normalization* and has the advantage that evaluation never encounters a free variable so long as it starts with a closed term. Only needing to deal with closed programs is a great boon for implementers. Beyond the added simplicity that comes from knowing we won't run into a variable, in a β reduction $(\lambda x.v)\ v' \to v[v'/x]$ the fact that v' is closed means that the substitution operation $[v'/x]$ need not rename variables in v. More generally, weak-head normalization on closed programs avoids the *variable capture problem*, which is quite convenient for implementations. In addition, programmers wanting to work with infinite data structures, which can improve modularity by separating unbounded producers from consumers [11], might be more inclined to use a non-strict programming language like Haskell rather than ML.

For these reasons, (1) call-by-name (or call-by-need) evaluation, (2) weak-head normal forms and (3) functional extensionality are all desirable properties to have. However, the combination of all three is inconsistent, representing a trilemma, so we can pick at most two. Switching to call-by-value evaluation respects extensionality while computing to weak-head normal forms. But, sticking with call-by-name forces us to abandon one of the other two. There is a fundamental tension between evaluation to weak-head normal form, which always finishes when it reaches a lambda, and the η axiom, which tells us that a lambda might not be done yet. For example, the η law says that $\lambda x.\Omega x$ is the same as Ω, where Ω is the non-terminating computation $(\lambda y.y\ y)(\lambda y.y\ y)$. Yet, $\lambda x.\Omega x$ is a weak-head normal form that is

Olivier Danvy & Ugo de'Liguoro (Eds.): WoC'15 Postproceedings
EPTCS 212, 2016, pp. 18–35, doi:10.4204/EPTCS.212.2

done while Ω isn't. Thus, if we want to use weak-head normal forms as our stopping point, the η law becomes suspect. This puts us in a worrisome situation: η-equivalent programs might have different termination behavior. As such, we cannot use essential properties, like our earlier example of the monad laws for state, for reasoning about our programs without the risk of changing a program that works into one that doesn't. The root of our problem is that we combined extensionality with effects, namely non-termination. This is one example of the recurrent tension that arises when we add effects to the call-by-name λ-calculus. For example, with printing as our effect, we would encounter a similar problem when combining η with evaluation to weak-head normal forms. Evaluating the term "`print "hello"`;$(\lambda y.y)$" to its weak-head normal form would print the string `"hello"`, while evaluating the η-expanded term $\lambda x.$(`print "hello"`;$(\lambda y.y)$)x would not.

Further, η reduction even breaks confluence when the λ-calculus is extended with control effects. Recent works [6, 15] suggest how to solve the problem by adopting an alternative view of functions. We bring new insight into this view: through abstract machines we show how the calling context, i.e. the elimination form of a lambda abstraction, can be given first-class status and lambda abstractions can then be seen as *pattern-matching* on the calling context. We then utilize the view that pattern-matching is simply syntactic sugar for projection operations; this suggests how to continue computing under a lambda abstraction. We present and relate a series of operational semantics and abstract machines for head reduction motivated by this insight into the nature of lambda abstractions.

After reviewing the small-step operational semantics, big-step operational semantics and Krivine abstract machine for weak-head evaluation (Section 2), we turn our investigation to head reduction with the goal of providing the three different styles of semantics. We start our exploration of head reduction with the Krivine abstract machine because it helps us to think about the evaluation context as a first class object, and extend it with a construct that *names* these contexts. This brings out the *negative* nature of functions [8]. Functions are not constructed but are *de-constructors*; it is the contexts of functions which are constructed. To emphasize this view, functions are presented as pattern-matching on the calling context, which naturally leads to a presentation of functions that translates pattern-matching into projections; analogously to the two ways tuples are treated in programming languages. By utilizing this approach based on projections, we modify our Krivine machine with control to continue evaluation instead of getting stuck on a top-level lambda (Section 3). Having gathered intuition on contexts, we focus on the control-free version of this machine. This leads to our first abstract machine for head reduction, and we utilize the syntactic correspondence [3] to derive an operational semantics for head reduction in the lambda-calculus (Section 4). The obtained operational semantics is, however, more complicated than would be desirable, and so we define a simpler but equivalent operational semantics. By once again applying the syntactic correspondence, we derive an abstract machine for head reduction which is not based on projections and, by way of the functional correspondence [1], we generate a big-step semantics which is shown to be equivalent to Sestoft's big step semantics for head reduction [18]. Finally, we conclude with a more efficient implementation of the projection based machine that coalesces multiple projections into one (Section 5).

2 Weak-head Evaluation: Small and Big-step Operational Semantics and an Abstract Machine

The semantics of a programming language can come in different flavors. It can be given by creating a mapping from a syntactic domain of programs into an abstract domain of mathematical structures (denotational semantics), or it can be given only in terms of syntactic manipulations of programs (operational

$$E \in \text{EvaluationContexts} ::= \Box \mid E\ v$$

$$E[(\lambda x.v)\ v'] \mapsto E[v[v'/x]]$$

Figure 1: Small-step weak-head reduction

$$\text{N} \in \text{Neutral} ::= x \mid \text{N}\ v$$
$$\text{WHNF} \in \text{WeakHeadNormalForms} ::= \text{N} \mid \lambda x.v$$

Figure 2: Weak-head normal forms

semantics). Operational semantics can be further divided into *small-step* or *big-step*. A small-step operational semantics shows step-by-step how a program transitions to the final result. A big-step operational semantics, contrarily, only shows the relation between a program and its final result with no intermediate steps shown, as in an evaluation function. We first start in Figure 1 with a small-step call-by-name semantics for λ-calculus, whose terms are defined as follows:

$$v \in \text{Terms} ::= x \mid v\ v' \mid \lambda x.v \quad .$$

The *evaluation context*, denoted by E, is simply a term with a hole, written as $\Box$, which specifies where work occurs in a term. A bare $\Box$ says that evaluation occurs at the top of the program, and if that is not reducible then $E\ v$ says that the search should continue to the left of an application. The semantics is then given by a single transition rule which specifies how to handle application. According to this semantics, terms of the form $x\ ((\lambda x.x)y)$ or $\lambda x.x\ ((\lambda x.x)y)$ are not reducible. The final answer obtained is called *weak-head normal form* (whnf for short); a lambda abstraction is in whnf and an application of the form $x\ v_1 \cdots v_n$ is in whnf. We can give a grammar defining a whnf by using the notion of "neutral" from Girard for λ-calculus terms other than lambda abstractions [9] (see Figure 2).

Note that decomposing a program into an evaluation context and a redex is a meta-level operation in the small-step operational semantics. We can instead make this operation an explicit part of the formalization in an *abstract machine*. In Figure 3 we give the Krivine abstract machine [12], which can be derived directly from the operational semantics by reifying the evaluation context into a data structure called a *co-term* [3]. The co-term tp is understood as corresponding to the empty context $\Box$, while the call-stack $v \cdot E$ can be thought of as the context $E[\Box\ v]$. With this view, the formation of a command $\langle v \parallel E \rangle$ corresponds to plugging v into E to obtain $E[v]$. The reduction rules of the Krivine machine are justified by this correspondence: we have a rule that recognizes that $E[v\ v'] = (E[\Box\ v'])[v]$ and a rule for actually performing β reduction inside an evaluation context. We can further describe how to run a λ-calculus term in the Krivine machine by plugging the term into an empty context

$$v \rightsquigarrow \langle v \parallel \mathsf{tp} \rangle$$

then after performing as many evaluation steps as possible, we "readback" a lambda term using the rules

$$\langle v \parallel v' \cdot E \rangle \hookrightarrow \langle v\ v' \parallel E \rangle \qquad\qquad \langle v \parallel \mathsf{tp} \rangle \hookrightarrow v$$

Note that the first rule is only needed if we want to interpret open programs, because execution only terminates in commands of the form $\langle \lambda x.v \parallel \mathsf{tp} \rangle$ and $\langle x \parallel E \rangle$, and only the first of these can be closed.

$$
\begin{array}{ll}
c \in \text{Commands} & ::= \langle v \parallel E \rangle \\
E \in \text{CoTerms} & ::= \mathsf{tp} \mid v \cdot E
\end{array}
$$

$$
\begin{array}{ll}
\langle v\ v' \parallel E \rangle & \to \langle v \parallel v' \cdot E \rangle \\
\langle \lambda x.v \parallel v' \cdot E \rangle & \to \langle v[v'/x] \parallel E \rangle
\end{array}
$$

Figure 3: Krivine abstract machine

$$
\frac{}{x \Downarrow_{wh} x} \qquad \frac{}{\lambda x.v \Downarrow_{wh} \lambda x.v}
$$

$$
\frac{v_1 \Downarrow_{wh} \lambda x.v_1' \qquad v_1'[v_2/x] \Downarrow_{wh} v}{v_1\ v_2 \Downarrow_{wh} v}
$$

$$
\frac{v_1 \Downarrow_{wh} v_1' \qquad v_1' \not\equiv \lambda x.v}{v_1\ v_2 \Downarrow_{wh} v_1'\ v_2}
$$

Figure 4: Big-step semantics for weak-head reduction

The syntactic correspondence of Biernacka and Danvy [3] derives the Krivine machine from the small-step operational semantics of weak-head reduction by considering them both as functional programs and applying a series of correctness-preserving program transformations. The interpreter corresponding to the small-step semantics "decomposes" a term into an evaluation context and redex, reduces the redex, "recompose" the resulting term back into its context, and repeats this process until an answer is reached. Recomposing and decomposing always happen in turns, and decomposing always undoes recomposing to arrive again at the same place, so they can be merged into a single "refocus" function that searches for the next redex in-place. This non-tail recursive interpreter can then be made tail-recursive by inlining and fusing refocusing and reduction together. Further simplifications and compression of intermediate transitions in the tail-recursive interpreter yields an implementation of the abstract machine. Equivalence of the two semantic artifacts follows from the equivalence of the associated interpreters, which is guaranteed by construction due to the correctness of each program transformation used. Note that we use $\twoheadrightarrow$ and $\hookrightarrow\!\!\!\!\twoheadrightarrow$ as the reflexive-transitive closures of $\to$ and $\hookrightarrow$ respectively.

Theorem 1 (Equivalence of Krivine machine and small-step operational semantics)**.** *For any λ-calculus terms v, v' the following conditions are equivalent:*

1. *$v \mapsto\!\!\!\!\twoheadrightarrow v'$ such that there is no v'' where $v' \mapsto v''$;*
2. *there exists a command c such that $\langle v \parallel \mathsf{tp} \rangle \twoheadrightarrow c \hookrightarrow\!\!\!\!\twoheadrightarrow v'$ where there is no c' such that $c \to c'$.*

Let us now turn to the big-step weak-head semantics (see Figure 4). It is not obvious that this semantics corresponds to the small step semantics, a proof of correctness is required. Interestingly, Reynolds's functional correspondence [1, 16] links this semantics to the Krivine abstract machine by way of program transformations, similar to the connection between the small-step semantics and abstract machine. The big-step semantics is represented as a compositional interpreter which is then converted into continuation-passing style and defunctionalized (where higher-order functions are replaced with data structures which correspond to co-terms), yielding an interpreter representing the Krivine machine.

$$\mathrm{N} \in \mathrm{Neutral} ::= x \mid \mathrm{N}\ v$$
$$\mathrm{HNF} \in \mathrm{HeadNormalForms} ::= \mathrm{N} \mid \lambda x.\mathrm{HNF}$$

Figure 5: Head Normal Forms

Correctness follows from construction and is expressed analogously to Theorem 1 by replacing the small-step reduction (i.e. $\mapsto\!\!\!\!\rightarrow$) with the big-step (i.e. $\Downarrow$).

Theorem 2 (Equivalence of Krivine machine and big-step operational semantics)**.** *For any λ-calculus terms v, v' the following conditions are equivalent:*

1. $v \Downarrow_{wh} v'$*;*
2. *there exists a command c such that $\langle v \parallel \mathsf{tp} \rangle \twoheadrightarrow c \hookrightarrow\!\!\!\!\rightarrow v'$ where there is no c' such that $c \to c'$.*

In conclusion, we have seen three different semantics artifacts, small-step, big-step and an abstract machine, which thanks to the syntactic and functional correspondence define the *same* language. Our goal is to provide the three different styles of semantics for a different notion of final result: *head normal forms* (hnf for short) (see Figure 5). Note that head reduction unlike weak-head reduction allows execution under a lambda abstraction, e.g. $\lambda x.(\lambda z.z)x$ is a whnf but is not a hnf, whereas $\lambda x.(x(\lambda z.z)x)$ is both a whnf and a hnf.

3 Functions as Pattern-Matching on the Calling Context

We have seen how, in the Krivine machine, evaluation contexts take the form of a call-stack consisting of a list of arguments to the function being evaluated. Inspired by this direct representation of contexts, we can enhance the language with the ability to give a name to the calling context, analogous to naming terms with a let-construct. We introduce a new sort of variables (written using greek letters $\alpha, \beta, \ldots$), called *co-variables*, which name co-terms, and a new abstraction $\mu\alpha.c$. Operationally, a μ-term captures its evaluation context

$$\langle \mu\alpha.c \parallel E \rangle \to c[E/\alpha]$$

and substitutes it in for the variable α in the associated command. This is analogous to the evaluation of the term $\texttt{let}\ x = v\ \texttt{in}\ v'$, where v is substituted for each occurrence of x in v'. Interestingly, even though the μ rule seems very different from the application rule they are indeed very similar, when seen side-by-side:

$$\begin{array}{lcl} \langle \mu\alpha.c \parallel E \rangle & \to & c[E/\alpha] \\ \langle \lambda x.v \parallel v' \cdot E \rangle & \to & \langle v[v'/x] \parallel E \rangle \end{array}$$

Note that they both inspect the context or co-term. Similar to the way a μ-term corresponds to a let-term, the lambda abstraction corresponds to a term of the form $\texttt{let}\ (x,y) = v\ \texttt{in}\ v'$, which decomposes v while naming its sub-parts. So in contrast to the μ-term, the lambda abstraction decomposes the co-term instead of just naming it. To emphasize this view we write a function as a special form of μ-abstraction which *pattern-matches* on the context. The term $\mu[(x \cdot \alpha).c]$ gives names to both its argument x and the remainder of its context α in the command c. Operationally, $\mu[(x \cdot \alpha).c]$ can be thought of as waiting for a context $v \cdot E$ at which point computation continues in c with v substituted in for x and E substituted in for α. This view emphasizes that a function is given primarily by how it is used rather than how it is

$$
\begin{array}{lll}
c \in \text{Commands} & ::= \langle v \parallel E \rangle \\
v \in \text{Terms} & ::= x \mid v\ v' \mid \mu[(x \cdot \alpha).c] \mid \mu\alpha.c \\
E \in \text{CoTerms} & ::= \alpha \mid v \cdot E \mid \mathsf{tp}
\end{array}
$$

$$
\begin{array}{lcl}
\langle v\ v' \parallel E \rangle & \to & \langle v \parallel v' \cdot E \rangle \\
\langle \mu\alpha.c \parallel E \rangle & \to & c[E/\alpha] \\
\langle \mu[(x \cdot \alpha).c] \parallel v \cdot E \rangle & \to & c[E/\alpha, v/x]
\end{array}
$$

Figure 6: Krivine abstract machine for λ-calculus with control

defined; the call-stack formation operator $\cdot$ is the most important aspect in the theory of functions (rather than lambda abstraction). However, the two views of functions are equivalent. We can write the lambda abstraction $\lambda x.v$ as $\mu[(x \cdot \alpha).\langle v \parallel \alpha \rangle]$, given α not free in v. The application rule of the Krivine machine can be clearly seen as an example of pattern-matching when written in this style.

$$\langle \lambda x.v \parallel v' \cdot E \rangle = \langle \mu[(x \cdot \alpha).\langle v \parallel \alpha \rangle] \parallel v' \cdot E \rangle \to \langle v[v'/x] \parallel E \rangle$$

Similarly, we can also write $\mu[(x \cdot \alpha).c]$ as $\lambda x.\mu\alpha.c$, making it clear that these two formulations are the same. Interestingly, abstraction over co-terms is the only necessary ingredient to realizing control operations. We thus arrive at an extension of the Krivine machine with control given in Figure 6.

3.1 Surjective Call Stacks

There are two different ways to take apart tuples in programming languages. The first, as we've seen, is to provide functionality to decompose a tuple by matching on its structure, as in the pattern-matching let-term $\texttt{let}\ (x,y) = v'\ \texttt{in}\ v$. By pattern-matching, $\texttt{let}\ (x,y) = (v_1, v_2)\ \texttt{in}\ v$ evaluates to v with v_1 and v_2 substituted for x and y, respectively. The second is to provide primitive projection operations for accessing the components of the tuple, as in $\texttt{fst}(v)$ and $\texttt{snd}(v)$. The operation $\texttt{fst}(v_1, v_2)$ evaluates to v_1 and $\texttt{snd}(v_1, v_2)$ evaluates to v_2. These two different views on tuples are equivalent in a sense. The `fst` and `snd` projections can be written in terms of pattern-matching

$$\texttt{fst}(z) = (\texttt{let}\ (x,y) = z\ \texttt{in}\ x) \qquad\qquad \texttt{snd}(z) = (\texttt{let}\ (x,y) = z\ \texttt{in}\ y)$$

and likewise, "lazy" pattern-matching can be implemented in terms of projection operations

$$\texttt{let}\ (x,y) = v'\ \texttt{in}\ v \to v[\texttt{fst}(v')/x, \texttt{snd}(v')/y]$$

The projective view of tuples has the advantage of a simple interpretation of extensionality, that the tuple made from the parts of another tuple is the same, by the *surjectivity* law for pairs: $v = (\texttt{fst}(v), \texttt{snd}(v))$.

By viewing functions as pattern-matching constructs in a programming language, analogous to pattern-matching on tuples, we likewise have another interpretation of functions based on projection. That is, we can replace lambda abstractions or pattern-matching with projection operations, $\texttt{car}(E)$ and $\texttt{cdr}(E)$, for accessing the components of a calling context. The operation $\texttt{car}(v \cdot E)$ evaluates to the argument v and $\texttt{cdr}(v \cdot E)$ evaluates to the return context E. Analogously to the different views on tuples, this projective view on functional contexts can be used to implement pattern-matching:

$$\langle \mu[(x \cdot \alpha).c] \parallel E \rangle \to c[\texttt{car}(E)/x, \texttt{cdr}(E)/\alpha]$$

$$
\begin{array}{lll}
c \in \text{Commands} & ::= & \langle v \parallel E \rangle \\
v \in \text{Terms} & ::= & x \mid v\ v' \mid \mu\alpha.c \mid \mu[(x \cdot \alpha).c] \mid \mathtt{car}(S) \\
E \in \text{CoTerms} & ::= & \alpha \mid v \cdot E \mid S \\
S \in \text{StuckCoTerms} & ::= & \mathsf{tp} \mid \mathtt{cdr}(S)
\end{array}
$$

$$
\begin{array}{lcl}
\langle v\ v' \parallel E \rangle & \to & \langle v \parallel v' \cdot E \rangle \\
\langle \mu\alpha.c \parallel E \rangle & \to & c[E/\alpha] \\
\langle \mu[(x \cdot \alpha).c] \parallel v \cdot E \rangle & \to & c[E/\alpha, v/x] \\
\langle \mu[(x \cdot \alpha).c] \parallel S \rangle & \to & c[\mathtt{car}(S)/x, \mathtt{cdr}(S)/\alpha]
\end{array}
$$

Figure 7: Head reduction abstract machine for λ-calculus with control (projection based)

And since lambda abstractions can be written in terms of pattern-matching, they can also be implemented in terms of $\mathtt{car}(-)$ and $\mathtt{cdr}(-)$:

$$\langle \lambda x.v \parallel E \rangle = \langle \mu[(x \cdot \alpha).\langle v \parallel \alpha \rangle] \parallel E \rangle \to \langle v[\mathtt{car}(E)/x] \parallel \mathtt{cdr}(E) \rangle$$

This projective view of functions has been previously used in Nakazawa and Nagai's $\Lambda\mu_{\text{cons}}$-calculus [15] to establish confluence in a call-by-name λ-calculus with control. In this setting, extensional reasoning is captured by a surjectivity law on co-terms that a co-term is always equal to the call-stack formed from its projections, analogous to the law for surjective pairs: $E = \mathtt{car}(E) \cdot \mathtt{cdr}(E)$. Note that the equational soundness of the η-respecting translation of the pattern-matching lambda into projection has also been discovered in the context of the sequent calculus [10, 14]. Therefore, the evaluation contexts of extensional functions are *surjective call-stacks*.

In the case where the co-term is $v \cdot E$, the reduction of pattern-matching into projection justifies our existing reduction rule by performing reduction inside of a command:

$$\langle \mu[(x \cdot \alpha).c] \parallel v \cdot E \rangle \to c[\mathtt{car}(v \cdot E)/x, \mathtt{cdr}(v \cdot E)/\alpha] \twoheadrightarrow c[v/x, E/\alpha]$$

However, when working with abstract machines we want to keep reduction at the top of a program, therefore we opt to keep using the rule which combines both steps into one. Instead, rewriting lambda abstractions as projection suggests what to do in the case where E is *not* a stack extension. Specifically, in the case where E is the top-level constant tp we have the following rule

$$\langle \mu[(x \cdot \alpha).c] \parallel \mathsf{tp} \rangle \to c[\mathtt{car}(\mathsf{tp})/x, \mathtt{cdr}(\mathsf{tp})/\alpha]$$

which has no equivalent in the Krivine machine where the left-hand side of this reduction is stuck.

We thus arrive at another abstract machine in Figure 7 which works just like the Krivine machine with control except that now we have a new syntactic sort of stuck co-terms. The idea behind stuck co-terms is that E is stuck if $\mathtt{cdr}(E)$ does not evaluate further. For example, the co-term $\mathtt{cdr}(\mathtt{cdr}(\mathsf{tp}))$ is stuck, but $\mathtt{cdr}(v \cdot \mathsf{tp})$ is not. Note, however, that we do not consider co-variables to be stuck co-terms since they may not continue to be stuck after substitution. For instance, $\mathtt{cdr}(\alpha)[v \cdot \mathsf{tp}/\alpha] = \mathtt{cdr}(v \cdot \mathsf{tp})$ is not stuck, so neither is $\mathtt{cdr}(\alpha)$. This means that we have no possible reduction for the command $\langle \mu[(x \cdot \beta).c] \parallel \alpha \rangle$, but this is not a problem since we assume to work only with programs which do not have free co-variables. Further, because we syntactically restrict the use of the projection to stuck co-terms, we do not need reduction rules like $\mathtt{car}(v \cdot E) \to v$ since $\mathtt{car}(v \cdot E)$ is not syntactically well formed

$$
\begin{array}{rcl}
c \in \text{Command} & ::= & \langle v \parallel E \rangle \\
v \in \text{Terms} & ::= & x \mid v\ v \mid \lambda x.v \mid \mathtt{car}(S) \\
E \in \text{CoTerms} & ::= & v \cdot E \mid S \\
S \in \text{StuckCoTerms} & ::= & \mathsf{tp} \mid \mathtt{cdr}(S)
\end{array}
$$

$$
\begin{array}{lcl}
\langle v\ v' \parallel E \rangle & \to & \langle v \parallel v' \cdot E \rangle \\
\langle \lambda x.v \parallel v' \cdot E \rangle & \to & \langle v[v'/x] \parallel E \rangle \\
\langle \lambda x.v \parallel S \rangle & \to & \langle v[\mathtt{car}(S)/x] \parallel \mathtt{cdr}(S) \rangle
\end{array}
$$

Figure 8: Head reduction abstract machine for λ-calculus (projection based)

in our machine. Intuitively, anytime we would have generated $\mathtt{car}(v \cdot E)$ or $\mathtt{cdr}(v \cdot E)$ we instead eagerly perform the projection reduction in the other rules.

With the projection based approach, we are in a situation where there is always a reduction rule which can fire at the top of a command, except for commands of the form $\langle x \parallel E \rangle$ or $\langle \mathtt{car}(S) \parallel E \rangle$, which are done, or $\langle \mu[(x \cdot \beta).c] \parallel \alpha \rangle$ which is stuck on a free co-variable. Specifically, we are no longer stuck when evaluating a pattern-matching function term at the top-level, i.e. $\langle \mu[(x \cdot \alpha).c] \parallel \mathsf{tp} \rangle$. As a consequence of this, reduction of co-variable closed commands now respects η. If we have a command of the form $\langle \mu[(x \cdot \alpha).\langle v \parallel x \cdot \alpha \rangle] \parallel E \rangle$ with no free co-variables and where x and α do not appear free in v, there are two possibilities depending on the value of E. Either E is a call-stack $v' \cdot E'$, so we β reduce

$$\langle \mu[(x \cdot \alpha).\langle v \parallel x \cdot \alpha \rangle] \parallel v' \cdot E' \rangle \to \langle v \parallel v' \cdot E' \rangle$$

or E must be a stuck co-term, so we reduce by splitting it with projections

$$\langle \mu[(x \cdot \alpha).\langle v \parallel x \cdot \alpha \rangle] \parallel S \rangle \to \langle v \parallel \mathtt{car}(S) \cdot \mathtt{cdr}(S) \rangle$$

meaning that we can continue to evaluate v. Thus, the use of projection out of surjective call-stacks offers a way of implementing call-by-name reduction while also respecting η.

4 Head Evaluation: Small and Big-step Operational Semantics and Abstract Machines

We have considered the Krivine machine with control to get an intuition about dealing with co-terms, however, our primary interest is to work with the pure λ-calculus. Therefore, from the Krivine machine with control and projection, we derive an abstract machine for the pure λ-calculus which performs head reduction (see Figure 8). Observe that the only co-variable is tp, which represents the "top-level" of the program. Here we use $\mathtt{car}(-)$ and $\mathtt{cdr}(-)$ (as well as the top-level context tp) as part of the implementation since they can appear in intermediate states of the machine. However, we still assume that the programs being evaluated are pure lambda terms. Observe that the projection based approach works just like the original Krivine machine (see Figure 3), except in the case where we need to reduce a lambda in a context that is not manifestly a call-stack ($\langle \lambda x.v \parallel S \rangle$), and in that situation we continue evaluating the body of the lambda. To avoid the problem of having free variables, we replace a variable (x) with the

projection into the context ($\mathtt{car}(S)$) and indicate that we are now evaluating under the binder with the new context ($\mathtt{cdr}(S)$).

Not all commands derivable from the grammar of Figure 8 are sensible; for example, $\langle \mathtt{car}(\mathsf{tp}) \parallel \mathsf{tp} \rangle$ and $\langle x \parallel \mathtt{car}(\mathtt{cdr}(\mathsf{tp})).\mathtt{cdr}(\mathsf{tp}) \rangle$ are not. Intuitively, the presence of $\mathtt{car}(\mathsf{tp})$ means that reduction has gone under a lambda abstraction so the co-term needs to witness that fact by terminating in $\mathtt{cdr}(\mathsf{tp})$, making $\langle \mathtt{car}(\mathsf{tp}) \parallel \mathtt{cdr}(\mathsf{tp}) \rangle$ a legal command. Analogously, the presence of $\mathtt{car}(\mathtt{cdr}(\mathsf{tp}))$ indicates that reduction has gone under two lambda abstractions and therefore the co-term needs to terminate in $\mathtt{cdr}(\mathtt{cdr}(\mathsf{tp}))$ making $\langle x \parallel \mathtt{car}(\mathtt{cdr}(\mathsf{tp})).\mathtt{cdr}(\mathtt{cdr}(\mathsf{tp})) \rangle$ a legal command. To formally define the notion of a legal command, we make use of the notation $\mathtt{cd}^n\mathtt{r}(-)$ for n applications of the $\mathtt{cdr}$ operation, and we write $\mathtt{cd}^n\mathtt{r}(\mathsf{tp}) \leq \mathtt{cd}^m\mathtt{r}(\mathsf{tp})$ if $n \leq m$ and similarly $\mathtt{cd}^n\mathtt{r}(\mathsf{tp}) < \mathtt{cd}^m\mathtt{r}(\mathsf{tp})$ if $n < m$. We will only consider legal commands, and obviously reduction preserves legal commands.

Definition 1. *A command of the form $\langle v_1 \parallel v_2 \cdots v_n \cdot S \rangle$ is legal if and only if for $1 \leq i \leq n$ and for every $\mathtt{car}(S')$ occurring in v_i, $S' < S$.*

As we saw for the Krivine machine, we have an associated readback function with an additional rule

$$\langle v \parallel \mathtt{cdr}(S) \rangle \hookrightarrow \langle \lambda x.v[x/\mathtt{car}(S)] \parallel S \rangle$$

for extracting resulting λ-calculus terms after reduction has terminated, where $v[x/\mathtt{car}(S)]$ is understood as replacing every occurrence of $\mathtt{car}(S)$ in v with x (which is assumed to be fresh). The readback relation is justified by reversing the direction of the reduction $\langle \lambda x.v \parallel S \rangle \to \langle v[\mathtt{car}(S)/x] \parallel \mathtt{cdr}(S) \rangle$.

Example 1. If we start with the term $\lambda x.(\lambda y.y)\ x$, we get an evaluation trace

$$\begin{aligned}
\langle \lambda x.(\lambda y.y)\ x \parallel \mathsf{tp} \rangle &\to \langle (\lambda y.y)\ \mathtt{car}(\mathsf{tp}) \parallel \mathtt{cdr}(\mathsf{tp}) \rangle \\
&\to \langle \lambda y.y \parallel \mathtt{car}(\mathsf{tp}) \cdot \mathtt{cdr}(\mathsf{tp}) \rangle \\
&\to \langle \mathtt{car}(\mathsf{tp}) \parallel \mathtt{cdr}(\mathsf{tp}) \rangle \\
&\hookrightarrow \langle \lambda x.x \parallel \mathsf{tp} \rangle \\
&\hookrightarrow \lambda x.x
\end{aligned}$$

which reduces $\lambda x.(\lambda y.y)\ x$ to $\lambda x.x$. This corresponds to the intuitive idea that head reduction performs weak-head reduction until it encounters a lambda, at which point it recursively performs head reduction on the body of that lambda.

Applying Biernacka and Danvy's syntactic correspondence, we reconstruct the small-step operational semantics of Figure 9. Because we want to treat contexts and terms separately, we create a new syntactic category of indices which replaces the appearance of $\mathtt{car}(S)$ in terms, since stuck co-terms correspond to top-level contexts. The function $\mathit{Count}(-)$ is used for converting between top-level contexts and indices. Note that, as we now include additional syntactic objects beyond the pure λ-calculus, we need a readback relation just like in the abstract machine:

$$S[\lambda.v] \hookrightarrow S[\lambda x.v[x/\mathit{Count}(S)]]$$

Example 2. Corresponding to the abstract machine execution in Example 1 we have:

$$\begin{aligned}
\lambda x.(\lambda y.y)\ x &\mapsto && \text{where } S = \square \\
\lambda.(\lambda y.y)\ \mathrm{zero} &\mapsto && \text{where } S = \lambda.\square \text{ and } E = \square \\
\lambda.\mathrm{zero} &\hookrightarrow \\
\lambda x.x
\end{aligned}$$

Here, the context $\lambda.\square$ zero corresponds to the co-term $\mathtt{car}(\mathsf{tp}) \cdot \mathtt{cdr}(\mathsf{tp})$.

$$
\begin{aligned}
t \in \text{TopTerms} &::= v \mid \lambda.t \\
v \in \text{Terms} &::= x \mid v\ v \mid \lambda x.v \mid i \qquad \mathit{Count} : \text{TopLevelContexts} \to \text{Indices} \\
i \in \text{Indices} &:= \text{zero} \mid \text{succ}(i) \qquad \mathit{Count}(\Box) = \text{zero} \\
E \in \text{Contexts} &::= E\ v \mid \Box \qquad \mathit{Count}(\lambda.S) = \text{succ}(\mathit{Count}(S)) \\
S \in \text{TopLevelContexts} &::= \lambda.S \mid \Box
\end{aligned}
$$

$$
\begin{aligned}
S[E[(\lambda x.v)\ v']] &\mapsto S[E[v[v'/x]]] \\
S[\lambda x.v] &\mapsto S[\lambda.v[\mathit{Count}(S)/x]]
\end{aligned}
$$

Figure 9: Small-step head reduction corresponding to the projection based abstract machine

$$
\begin{aligned}
v \in \text{Terms} &::= x \mid v\ v \mid \lambda x.v \\
E \in \text{Contexts} &::= E\ v \mid \Box \\
S \in \text{TopLevelContexts} &::= E \mid \lambda x.S
\end{aligned}
$$

$$
S[(\lambda x.v)\ v'] \mapsto S[v[v'/x]]
$$

Figure 10: Small-step head reduction

Because abstract machine co-terms correspond to inside out contexts, we can not just define evaluation contexts as

$$E \in \text{Contexts} ::= E\ v \mid S$$

which would make $((\lambda.S)\ v)$ a context which does not correspond to any co-term (and would allow for reduction under binders). Indeed, the variable-free version of lambda abstraction is only allowed to occur at the top of the program. Thus, composed contexts of the form $S[E[\Box]]$ serve as the exact equivalent of co-terms. Analogously to the notion of legal commands, we have the notion of legal top-level terms, and we will only consider legal terms.

Definition 2. *A top-level term t of the form $S[v]$ is legal if and only if for every index i occurring in v, $i < \mathit{Count}(S)$.*

This operational semantics has the virtue of being reconstructed directly from the abstract machine, which automatically gives a correctness result analogous to Theorem 1.

Theorem 3 (Equivalence of small-step semantics and abstract machine based on projections)**.** *For any index free terms v, v' the following conditions are equivalent:*

1. *$v \mapsto\!\!\!\!\to t \hookrightarrow\!\!\!\!\to v'$ such that there is no t' where $t \mapsto t'$;*
2. *there exists a command c such that $\langle v \parallel \mathsf{tp} \rangle \to\!\!\!\!\to c \hookrightarrow\!\!\!\!\to v'$ where there is no c' such that $c \to c'$.*

However, while, in our opinion, the associated abstract machine was extremely elegant, this operational semantics seems unnecessarily complicated. Fortunately, we can slightly modify it to achieve a much simpler presentation. At its core, the cause of the complexity of the operational semantics is the use of indices for top-level lambdas. A simpler operational semantics would only use named vari-

$$
\begin{aligned}
c \in \text{Command} &::= \langle v \parallel E \rangle \\
v \in \text{Terms} &::= x \mid v\ v \mid \lambda x.v \\
E \in \text{CoTerms} &::= v \cdot E \mid S \\
S \in \text{StuckCoTerms} &::= \mathsf{tp} \mid \mathtt{Abs}(x, S)
\end{aligned}
$$

$$
\begin{array}{lcl}
\langle v\ v' \parallel E \rangle & \to & \langle v \parallel v' \cdot E \rangle \\
\langle \lambda x.v \parallel v' \cdot E \rangle & \to & \langle v[v'/x] \parallel E \rangle \\
\langle \lambda x.v \parallel S \rangle & \to & \langle v \parallel \mathtt{Abs}(x, S) \rangle
\end{array}
$$

Figure 11: Head reduction abstract machine

ables (see Figure 10). To show that the two semantics are indeed equivalent we make use of Sabry and Wadler's *reduction correspondence* [17]. For readability, we write $\mapsto_S$ for the evaluation according to Figure 9 and $\mapsto_T$ for the evaluation according to Figure 10. We define the translations $*$ and # from top-level terms, possibly containing indices, to pure lambda-calculus terms, and from pure lambda terms to top-level terms, respectively. More specifically, $*$ corresponds to the normal form of the readback rule and # to the normal form with respect to non-β $\mapsto_S$ reductions.

Theorem 4. *The reduction systems $\mapsto\!\!\!\!\rightarrow_S$ and $\mapsto\!\!\!\!\rightarrow_T$ are sound and complete with respect to each other:*

1. *$t \mapsto\!\!\!\!\rightarrow_S t'$ implies $t^* \mapsto\!\!\!\!\rightarrow_T t'^*$;*
2. *$v \mapsto\!\!\!\!\rightarrow_T v'$ implies $v^\# \mapsto\!\!\!\!\rightarrow_S v'^\#$;*
3. *$t \mapsto\!\!\!\!\rightarrow_S t^{*\#}$ and $v^{\#*} = v$.*

Proof. First, we observe that the non-β reduction of $\mapsto_S$ is inverse to the readback rule, so that for any top-level terms t and t', $t \mapsto_S t'$ by a non-β reduction if and only if $t' \hookrightarrow t$. Second, we note that for any top-level context $S = \lambda \ldots \lambda.\Box$ and term v of Figure 9, there is a top-level context $S' = \lambda x_0 \ldots \lambda x_n.\Box$ of Figure 10 and substitution $\sigma = [x_0/0, \ldots, x_n/n]$ such that $(S[v])^* = S'[v^\sigma]$ by induction on S. Similarly, for any top-level context S of Figure 10 and neutral term N, there is a top-level context S' and context E' of Figure 9 such that $(S[N])^\# = S'[E'[N^\sigma]]$ by induction on S.

1. Follows from the fact that each step of $\mapsto_S$ corresponds to zero or one step of $\mapsto_T$:

$$
\begin{array}{ccc}
S[E[(\lambda x.v)v']] & \overset{S}{\longmapsto} & S[E[v[v'/x]]] \\
\downarrow * & & \downarrow * \\
S'[E^\sigma[(\lambda x.v^\sigma)v'^\sigma]] & \overset{T}{\dashmapsto} & S'[E^\sigma[v^\sigma[v'^\sigma/x]]]
\end{array}
\qquad
\begin{array}{ccc}
S[\lambda x.v] & \overset{S}{\longmapsto} & S[\lambda.v[\mathit{Count}(S)/x]] \\
\downarrow * & & \downarrow * \\
S'[\lambda x.v^\sigma] & \overset{T}{=\!=} & S'[\lambda x.v^\sigma]
\end{array}
$$

2. Follows from the fact that each step of $\mapsto_T$ corresponds to one step of $\mapsto_S$, similar to the above.
3. Follows from induction on the reductions of the $*$ and # translations along with the facts that the non-β reduction of $\mapsto_S$ is inverse to $\hookrightarrow$, $t^{*\#}$ is the normal form of t with respect to non-β reductions of $\mapsto_S$, and the top-level term v is the normal form of the readback. □

$$\frac{v \Downarrow_{wh} \lambda x.v' \qquad v' \Downarrow_h v''}{v \Downarrow_h \lambda x.v''} \qquad \frac{v \Downarrow_{wh} v' \qquad v' \not\equiv \lambda x.v''}{v \Downarrow_h v'}$$

Figure 12: Big-step semantics for head reduction

$$\frac{}{x \Downarrow_{sf} x} \qquad \frac{v \Downarrow_{sf} v'}{\lambda x.v \Downarrow_{sf} \lambda x.v'}$$

$$\frac{v_1 \Downarrow_{wh} v_1' \qquad v_1' \not\equiv (\lambda x.v)}{v_1\ v_2 \Downarrow_{sf} v_1'\ v_2} \qquad \frac{v_1 \Downarrow_{wh} \lambda x.v_1' \qquad v_1'[v_2/x] \Downarrow_{sf} v}{v_1\ v_2 \Downarrow_{sf} v}$$

Figure 13: Sestoft's big-step semantics for head reduction

Remark 1. The small-step semantics of Figure 10 captures the definition of head reduction given by Barendregt (Definition 8.3.10) [2] who defines a head redex as follows: If M is of the form

$$\lambda x_1 \cdots x_n.(\lambda x.M_0)M_1 \cdots M_m \ ,$$

for $n \geq 0$, $m \geq 1$, then $(\lambda x.M_0)M_1$ is called the *head redex* of M. Note that the context $\lambda x_1 \cdots x_n.\Box \cdots M_m$ is broken down into $\lambda x_1 \cdots x_n.\Box$ which corresponds to a top-level context S and $\Box \cdots M_n$ which corresponds to a context E. Thus, Barendregt's notion of one-step head reduction:

$$M \to_h N \text{ if } M \to^{\Delta} N \ ,$$

i.e. N results from M by contracting the head redex Δ, corresponds to the evaluation rule of Figure 10.

By once again applying the syntactic correspondence, this time to the simplified operational semantics in Figure 10, we derive a new abstract machine (Figure 11) which works by going under lambdas without performing any substitutions in the process. To extract computed λ-calculus terms we add one rule to the readback relation for the Krivine machine.

$$\langle v \parallel \mathtt{Abs}(x, S) \rangle \hookrightarrow \langle \lambda x.v \parallel S \rangle$$

The equivalence of the abstract machine in Figure 11 and the operational semantics of Figure 10 follows by construction, and is expressed analogously to Theorem 1.

From the abstract machine in Figure 11 we again apply the functional correspondence [1] to derive the big-step semantics in Figure 12. This semantics utilizes the big-step semantics for weak-head reduction ($\Downarrow_{wh}$) from Figure 4 as part of its definition of head reduction ($\Downarrow_h$). This semantics tells us that the way to evaluate a term to head-normal form is to first evaluate it to weak-head normal form, and if the resulting term is a lambda to recursively evaluate the body of that lambda. This corresponds to the behavior of the abstract machine which works just like the Krivine machine (which only reduces to weak head) except in the situation where we have a command consisting of a lambda abstraction and a context which is not a call-stack. Again, the big-step semantics corresponds to the abstract machine by construction, the equivalence can be expressed analogously to Theorem 2.

Interestingly, Sestoft gave a different big-step semantics for head reduction [18] (Figure 13). The only difference between the two semantics is in the way they search for β redexes. Sestoft's semantics is more redundant by repeating the same logic for function application from the underlying weak-head evaluation, whereas the semantics of Figure 12 only adds a loop for descending under the top-level lambdas produced by weak-head evaluation. However, besides this difference in the search for β redexes, they are the same: they eventually find and perform β redexes in exactly the same order. By structural induction one can indeed verify that they generate identical operational semantics.

Theorem 5. *$v \Downarrow_h v'$ if and only if $v \Downarrow_{sf} v'$.*

5 Coalesced Projection

The projection based machine in Figure 8 has the desirable feature that we will never encounter a variable when we start with a closed program. Additionally, unlike the machine in Figure 11, machine states do not have co-terms that bind variables in their opposing term. On the other hand, it has a certain undesirable property in that when we substitute a projection in for a variable

$$\langle \lambda x.v \parallel S \rangle \to \langle v[\mathtt{car}(S)/x] \parallel \mathtt{cdr}(S) \rangle$$

we may significantly increase the size of the term as the stuck co-term has size linear in the number of lambda abstractions we have previously eliminated in this way. The story is even worse in the other direction: the readback relation for the projection machine (and associated operational semantics) depends on performing a deep pattern-match to replace projections with variables

$$\langle v \parallel \mathtt{cdr}(S) \rangle \hookrightarrow \langle \lambda x.v[x/\mathtt{car}(S)] \parallel S \rangle$$

which requires matching *all the way* against S.

We now show that we can improve the abstract machine for head evaluation by combining multiple projections into one. We will utilize the macro projection operations $\mathtt{pick}^n(-)$ and $\mathtt{drop}^n(-)$ which, from the perspective of $\mathtt{car}(-)$ and $\mathtt{cdr}(-)$, coalesce sequences of projections into a single operation:

$$\mathtt{drop}^0(E) \triangleq E \qquad \mathtt{pick}^n(E) \triangleq \mathtt{car}(\mathtt{drop}^n(E))$$
$$\mathtt{drop}^{n+1}(E) \triangleq \mathtt{cdr}(\mathtt{drop}^n(E))$$

Given that our operational semantics is for the pure lambda calculus, $\mathtt{pick}^n(\mathsf{tp})$ and $\mathtt{drop}^n(\mathsf{tp})$ are the only call-stack projections we need in the syntax.

We now construct an abstract machine (Figure 14) for head evaluation of the λ-calculus which is exactly like the Krivine machine with one additional rule utilizing the coalesced projections. As in the machine of Figure 8, the coalesced machine "splits" the top-level into a call-stack and continues. Analogously to Example 1, one has:

$$\begin{aligned}\langle \lambda x.(\lambda y.y)x \parallel \mathtt{drop}^0(\mathsf{tp}) \rangle &\to \langle (\lambda y.y)(\mathtt{pick}^0(\mathsf{tp})) \parallel \mathtt{drop}^1(\mathsf{tp}) \rangle \\ &\twoheadrightarrow \langle \mathtt{pick}^0(\mathsf{tp}) \parallel \mathtt{drop}^1(\mathsf{tp}) \rangle\end{aligned}$$

If the machine terminates with a result like $\langle \mathtt{pick}^n(\mathsf{tp}) \parallel E \rangle$, it is straightforward to read back the corresponding head normal form:

$$\langle v \parallel v' \cdot E \rangle \hookrightarrow \langle v\ v' \parallel E \rangle \qquad \langle v \parallel \mathsf{tp} \rangle \hookrightarrow v \qquad \langle v \parallel \mathtt{drop}^{n+1}(\mathsf{tp}) \rangle \hookrightarrow \langle \lambda x.v[x/\mathtt{pick}^n(\mathsf{tp})] \parallel \mathtt{drop}^n(\mathsf{tp}) \rangle$$

$$
\begin{aligned}
c \in \text{Command} &::= \langle v \parallel E \rangle \\
v \in \text{Terms} &::= x \mid v\ v \mid \lambda x.v \mid \mathtt{pick}^n(\mathsf{tp}) \\
E \in \text{CoTerms} &::= v \cdot E \mid \mathtt{drop}^n(\mathsf{tp})
\end{aligned}
$$

$$
\begin{array}{lcl}
\langle v\ v' \parallel E \rangle & \to & \langle v \parallel v' \cdot E \rangle \\
\langle \lambda x.v \parallel v' \cdot E \rangle & \to & \langle v[v'/x] \parallel E \rangle \\
\langle \lambda x.v \parallel \mathtt{drop}^n(\mathsf{tp}) \rangle & \to & \langle v[\mathtt{pick}^n(\mathsf{tp})/x] \parallel \mathtt{drop}^{n+1}(\mathsf{tp}) \rangle
\end{array}
$$

Figure 14: Coalesced projection abstract machine

where x is not free in v in the third rule. Note that the substitution $v[x/\mathtt{pick}^n(\mathsf{tp})]$ replaces all occurrences of terms of the form $\mathtt{pick}^n(\mathsf{tp})$ inside v with x. Correctness is ensured by the fact that reduction in the machine with coalesced projections corresponds to reduction in the machine with non-coalesced projections, which in turn corresponds to the operational semantics of head reduction.

Theorem 6 (Equivalence of Coalesced and Non Coalesced Projection Machines)**.**

1. *$c \to c'$ in the coalesced machine if and only if $c \to c'$ in the non-coalesced machine and*
2. *$c \hookrightarrow c'$ in the coalesced machine if and only if $c \hookrightarrow c'$ in the non-coalesced machine*

where conversion is achieved by interpreting $\mathtt{drop}^n(-)$ and $\mathtt{pick}^n(-)$ as macros.

Proof. By cases. Note that the interesting case in each direction is $\langle \lambda x.v[x/\mathtt{pick}^n(\mathsf{tp})] \parallel \mathtt{drop}^n(\mathsf{tp}) \rangle \to \langle v \parallel \mathtt{drop}^{n+1}(\mathsf{tp}) \rangle$ which follows since

$$
\begin{aligned}
\langle \lambda x.v[x/\mathtt{pick}^n(\mathsf{tp})] \parallel \mathtt{drop}^n(\mathsf{tp}) \rangle &\to \langle v[x/\mathtt{pick}^n(\mathsf{tp})][\mathtt{car}(\mathtt{drop}^n(\mathsf{tp}))/x] \parallel \mathtt{cdr}(\mathtt{drop}^n(\mathsf{tp})) \rangle \\
&= \langle v[x/\mathtt{pick}^n(\mathsf{tp})][\mathtt{pick}^n(\mathsf{tp})] \parallel \mathtt{drop}^{n+1}(\mathsf{tp}) \rangle \\
&= \langle v \parallel \mathtt{drop}^{n+1}(\mathsf{tp}) \rangle
\end{aligned}
$$

□

Our abstract machine, in replacing variables with coalesced sequences of projections, exhibits a striking resemblance to implementations of the λ-calculus based on de Bruijn indices [5]. Indeed, our abstract machine can be seen as given a semantic justification for de Bruijn indices as offsets into the call-stack. However, our approach differs from de Bruijn indices in that, in general, we still utilize named variables. Specifically, numerical indices are only ever used for representing variables bound in the leftmost branch of the lambda term viewed as a tree. As such, we avoid the complexities of renumbering during β reduction. However, implementations which do use de Bruijn to achieve a nameless implementation of variables in general have the advantage that the substitution operations $[\mathtt{pick}^n(\mathsf{tp})/x]$ as used in the abstract machine could be replaced with a no-op. The Krivine machine is often presented using de Bruijn despite being used only for computing whnfs. With the addition of our extra rule—which in a de Bruijn setting does not require substitution—we extend it to compute hnfs.

Further, we can extract from our abstract machine an operational semantics which utilizes de Bruijn indices only for top-level lambdas by way of the syntactic correspondence. The resulting semantics, in Figure 15, keeps track of the number of top-level lambdas as part of $\lambda^n.v$ while performing head reduction on v. As expected, the equivalence between the coalesced abstract machine and the operational semantics is by construction. The coalesced projection machine, and de Bruijn based operational semantics, are

$$
\begin{array}{rcl}
t \in \text{TopTerms} ::= & \lambda^n.v & \\
v \in \text{Terms} ::= & x \mid n \mid v\ v \mid \lambda x.v & \\
E \in \text{Contexts} ::= & \Box \mid E\ v & \\
\lambda^n.\lambda x.v & \mapsto & \lambda^{n+1}.v[n/x] \\
\lambda^n.E[(\lambda x.v)\ v'] & \mapsto & \lambda^n.E[v[v'/x]]
\end{array}
$$

Figure 15: Operational semantics of head reduction with partial de Bruijn

essentially equivalent to the projection machine and associated operational semantics. The difference is that by coalescing multiple projections into one, we replace the use of unary natural numbers with abstract numbers which could be implemented efficiently. In this way, we both greatly reduce the size of terms and make the pattern-matching to readback final results efficient. Numerical de Bruijn indices are an efficient implementation of the idea that lambda abstractions use variables to denote projections out of a call stack.

6 Conclusion

The desirable combination of weak-head normal forms, call-by-name evaluation, and extensionality is not achievable. This is a fundamental, albeit easily forgettable, constraint in programming language design. However, we have seen that there is a simple way out of this trilemma: replace weak-head normal forms with head normal forms. Moreover, reducing to head normal form is motivated by an analysis of the meaning of lambda abstraction. We took a detour through control so that we could directly reason about not just terms but also their context. This detour taught us to think about the traditional β rule as a rule for pattern-matching on contexts. By analogy to the different ways we can destruct terms we recognized an alternative implementation based on projections out of call-stacks. Projection allows us to run a lambda abstraction even when we don't yet have its argument.

Returning to the pure λ-calculus we derived an abstract machine from this projection-based approach. Using the syntactic correspondence we derived from our abstract machine an operational semantics, and showed how that could be massaged into a simpler operational semantics for head reduction. With a second use of the syntactic correspondence we derived an abstract machine which implemented head reduction in what seems a more traditional way. By the functional correspondence we showed finally that our entire chain of abstract machines and operational semantics correspond to Sestoft's big-step semantics for head reduction. The use of automated techniques for deriving one form of semantics from another makes it easy to rapidly explore the ramifications that follow from a shift of strategy; in our case from weak-head to head reduction. So in the end, we arrive at a variety of different semantics for head reduction, all of which are equivalent and correspond to Barendregt's definition of head reduction for the λ-calculus.

Branching from our projection based machine we derived an efficient abstract machine which coalesces projections, so we may have our cake and eat it too. We escape the trilemma by giving up on weak-head reduction, while still retaining its virtues like avoiding the variable capture problem and keeping all reductions at the top-level. Our machine is identical to the Krivine machine, which performs weak-head evaluation, except for a single extra rule which tells us what to do when we have a lambda abstraction and no arguments to feed it. Further, these changes can be seen as a limited usage of (and

$$
\begin{aligned}
c \in \text{Commands} &::= \langle v | \sigma | E \rangle \\
v \in \text{Terms} &::= x \mid \lambda x.v \mid v\ v \\
E \in \text{CoTerms} &::= \mathsf{tp} \mid t \cdot E \\
\sigma \in \text{Environments} &::= [] \mid (x \mapsto t) :: \sigma \\
t \in \text{Closures} &::= (v, \sigma)
\end{aligned}
$$

$$
\begin{aligned}
\langle v\ v' | \sigma | E \rangle &\to \langle v | \sigma | (v', \sigma) \cdot E \rangle \\
\langle \lambda x.v | \sigma | t \cdot E \rangle &\to \langle v | (x \mapsto t) :: \sigma | E \rangle \\
\langle x | \sigma | E \rangle &\to \langle v | \sigma' | E \rangle \quad \text{where } \sigma(x) = (v, \sigma')
\end{aligned}
$$

Figure 16: Krivine abstract machine with environments

$$
\begin{aligned}
c \in \text{Commands} &::= \langle v | \sigma | E \rangle \\
v \in \text{Terms} &::= x \mid \lambda x.v \mid v\ v \mid \mathtt{pick}^n(\mathsf{tp}) \\
E \in \text{CoTerms} &::= \mathtt{drop}^n(\mathsf{tp}) \mid t \cdot E \\
\sigma \in \text{Environments} &::= [] \mid (x \mapsto t) :: \sigma \\
t \in \text{Closures} &::= (v, \sigma)
\end{aligned}
$$

$$
\begin{aligned}
\langle v\ v' | \sigma | E \rangle &\to \langle v | \sigma | (v', \sigma) \cdot E \rangle \\
\langle \lambda x.v | \sigma | t \cdot E \rangle &\to \langle v | (x \mapsto t) :: \sigma | E \rangle \\
\langle \lambda x.v | \sigma | \mathtt{drop}^n(\mathsf{tp}) \rangle &\to \langle v | (x \mapsto (\mathtt{pick}^n(\mathsf{tp}), \sigma)) :: \sigma | \mathtt{drop}^{n+1}(\mathsf{tp}) \rangle \\
\langle x | \sigma | E \rangle &\to \langle v | \sigma' | E \rangle \quad \text{where } \sigma(x) = (v, \sigma')
\end{aligned}
$$

Figure 17: Head reduction abstract machine with environments

application for) de Bruijn indices.

Although our motivating problem and eventual solution were in the context of the pure λ-calculus, our detour through explicit continuations taught us valuable lessons about operational semantics. Having continuations forces us to tackle many operational issues head on because it makes the contexts a first class part of the language. Thus, a meta lesson of this paper is that adding control can be a helpful aid to designing even pure languages.

In practice, the trilemma is usually avoided by giving up on call-by-name or extensionality rather than weak-head normal forms. However, it may be that effective approaches to head evaluation such as those in this paper are of interest even in call-by-value languages or in settings (such as Haskell with `seq`) that lack the η axiom. Exploring the practical utility of head reduction may be a profitable avenue for future work.

Finally, we presented our abstract machines using substitutions and have suggested a possible alternative implementation based on de Bruijn indices, but we can also perform closure conversion to handle variables. For example, the machine in Figure 16 is an implementation of the Krivine machine using closures. We assume the existence of a data structure for maps from variable names to closures which supports at least an extension operator :: and variable lookup, which we write using list-like notation. Similarly, the machine in Figure 17 takes our efficient coalesced machine and replaces the use of substitutions with environments. This would correspond to a calculus of explicit substitutions la $\lambda\rho$ [7]. However, in general, the problem of how to handle variables is orthogonal from questions of opera-

tional semantics, and thus environment based handling of variables can be added after the fact. What the present paper suggests, however, is that the de Bruijn view of representing variables as numbers is semantically motivated by the desire to make reduction for the call-by-name lambda calculus consistent with extensional principles.

References

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy & Jan Midtgaard (2003): *A Functional Correspondence Between Evaluators and Abstract Machines*. In: *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming*, PPDP '03, ACM, New York, NY, USA, pp. 8–19, doi:10.1145/888251.888254.

[2] H.P Barendregt (1984): *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, Amsterdam.

[3] Małgorzata Biernacka & Olivier Danvy (2007): *A Concrete Framework for Environment Machines*. *ACM Transactions on Computational Logic* 9(1), pp. 1–30, doi:10.1145/1297658.1297664.

[4] Corrado Böhm (1968): *Alcune proprieta delle forme β-η-normali nel λ-K-calcolo*. *Pubblicazioni dell'Istituto per le Applicazioni del Calcolo* 696.

[5] N.G de Bruijn (1972): *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem*. *Indagationes Mathematicae (Proceedings)* 75(5), pp. 381 – 392, doi:10.1016/1385-7258(72)90034-0.

[6] Alberto Carraro, Thomas Ehrhard & Antonino Salibra (2012): *The stack calculus*. In: *Proceedings Seventh Workshop on Logical and Semantic Frameworks, with Applications, LSFA 2012, Rio de Janeiro, Brazil, September 29-30, 2012.*, pp. 93–108, doi:10.4204/EPTCS.113.10.

[7] P.L. Curien (1988): *The $\lambda\rho$-calculus: an Abstract Framework for Environment Machines*. Ecole Normale Supérieure (Paris). Laboratoire d'Informatique.

[8] Paul Downen & Zena M. Ariola (2014): *The Duality of Construction*. In Zhong Shao, editor: *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, *Lecture Notes in Computer Science* 8410, Springer, pp. 249–269, doi:10.1007/978-3-642-54833-8_14.

[9] Jean-Yves Girard, Paul Taylor & Yves Lafont (1989): *Proofs and types*, web reprint (2003) edition. Cambridge University Press.

[10] Hugo Herbelin (2005): *C'est maintenant qu'on calcule : Au cœur de la dualité*. In: *Habilitation à diriger les reserches.*

[11] J. Hughes (1989): *Why Functional Programming Matters*. *Comput. J.* 32(2), pp. 98–107, doi:10.1093/comjnl/32.2.98.

[12] Jean-Louis Krivine (2007): *A Call-by-name Lambda-calculus Machine*. *Higher Order Symbol. Comput.* 20(3), pp. 199–207, doi:10.1007/s10990-007-9018-9.

[13] Simon Marlow (2002): *State monads don't respect the monad laws in Haskell*. Haskell mailing list.

[14] Guillaume Munch-Maccagnoni (2013): *Syntax and Models of a non-Associative Composition of Programs and Proofs*. Ph.D. thesis, Univ. Paris Diderot.

[15] Koji Nakazawa & Tomoharu Nagai (2014): *Reduction System for Extensional Lambda-mu Calculus.* In: *RTA-TLCA*, pp. 349–363, doi:10.1007/978-3-319-08918-8_24.

[16] John C. Reynolds (1972): *Definitional Interpreters for Higher-order Programming Languages.* In: *Proceedings of the ACM Annual Conference - Volume 2*, ACM '72, ACM, New York, NY, USA, pp. 717–740, doi:10.1023/A:1010027404223.

[17] Amr Sabry & Philip Wadler (1997): *A Reflection on Call-by-Value*. *ACM Trans. Program. Lang. Syst.* 19(6), pp. 916–941, doi:10.1145/267959.269968.

[18] Peter Sestoft (2002): *Demonstrating Lambda Calculus Reduction*. In Torben ÆMogensen, David A. Schmidt & I. Hal Sudborough, editors: *The Essence of Computation*, Springer-Verlag New York, Inc., New York, NY, USA, pp. 420–435, doi:10.1007/3-540-36377-7_20.