Control Controls Extensional Execution

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

1. Abstract

Call-by-name evaluation, weak head normal forms, and functional extensionality: pick at most two. All three seem desirable, but are inconsistent together, representing a trilemma. We can respect extensionality while computing to weak head normal forms—where evaluation stops once it hits a lambda—by switching to call-by-value evaluation, but sticking with call-by-name forces us to abandon one of the other two. Weak head normal forms are appealing since they let us run closed programs without going under binders, which seems well suited for implementations. The essential problem, though, is that "stopping at a lambda" is fundamentally inconsistent with the η axiom which can erase top-level lambdas.

When we move from the pure, untyped λ-calculus to one with first-class control effects, the situation becomes even worse. Not only are we faced with our original trilemma, but η reduction breaks confluence as well. Recent work [2, 10] restores confluence in control calculi by replacing η with a different interpretation of extensionality and β with a different interpretation of functions. We bring new insight into these solutions by recasting functions as *pattern matching* abstractions on their calling contexts. Further, by translating this alternate viewpoint back to the λ-calculus, we derive an abstract machine for call-by-name evaluation which respects η, keeps evaluation at the top level, and avoids any descent under binders. Our machine sidesteps the trilemma by going further and computing to head normal forms. Effectively, control has taught us a better way to implement the pure λ-calculus.

2. Weak Head Normal Forms and Extensionality

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 [8]. 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 [1] .

For the second goal, it is important to have notions of normal forms, specifying the possible results of normalization, that can be efficiently computed. Beyond the usual β-normal forms, we utilize ideas such as head normal and weak head normal forms. A λ-calculus term is a *head normal form* (HNF) if it is a variable, a non-redex application whose left branch is a HNF, or a lambda whose body is a HNF. Going even further, a *weak head normal form* (WHNF) is defined just like a head normal form except that we consider *all* lambda abstractions to be WHNFs.

Weak head normal forms give an appealing approach to evaluating closed λ-calculus terms. A closed term is a WHNF if and only if it is a lambda abstraction, so there is no need to ever look inside lambdas during evaluation. This is a great boon for implementors: so long as we start with a closed term, we never encounter a free variable during evaluation. This lends itself nicely to abstract machines, such as the Krivine machine [7], which specifies how to find the next redex and computes WHNFs with only two rules (Figure 1). Note that we present this machine using substitution for clarity: implementing efficient substitution by environments and closures follows straightforwardly from the usual technique. By

$$
\begin{array}{rl}
c \in \text{Commands} ::= & \langle v \| e \rangle \\
v \in \text{Terms} ::= & x \mid \lambda x.v \mid v\ v \\
E \in \text{CoValues} ::= & \mathtt{tp} \mid v \cdot E
\end{array}
\qquad
\begin{array}{c}
\langle v\ v' \| E \rangle \rightsquigarrow \langle v \| v' \cdot E \rangle \\
\langle \lambda x.v \| v' \cdot E \rangle \rightsquigarrow \langle v[v'/x] \| E \rangle
\end{array}
$$

Figure 1. The Krivine Abstract Machine

comparison, computation of HNFs seems to inherently require reduction under lambda. Thus, computation of WHNFs appears favorable for practical implementations in a way that HNFs do not.

However, there is a fundamental tension between WHNFs and the η law. Evaluation to WHNF always finishes when it reaches a lambda, but fundamentally, η 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 WHNF that is *done* while Ω isn't. Thus, if we want to use WHNFs as our stopping point, the η law becomes suspect. This puts us in a worrying 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 evaluation to WHNF and η. Evaluating the term `print "hello"`; $(\lambda y.y)$ to

$$
\begin{array}{rlll}
c \in \text{Commands} ::= \langle v \| e \rangle & \langle \mu\alpha.c \| E \rangle \to_{\mu} c[E/\alpha] & \mathtt{cdr}(v \cdot E) \to_{\mathtt{cdr}} E \\
v \in \text{Terms} ::= x \mid \mu\alpha.c \mid \mu[(x \cdot \alpha).c] \mid \mathtt{car}(E) & \mathtt{car}(v \cdot E) \to_{\mathtt{car}} v & \mu\alpha.\langle v \| \alpha \rangle \to_{\eta_\mu} v \\
E \in \text{CoValues} ::= \alpha \mid v \cdot E \mid \mathtt{cdr}(E) & \mathtt{cdr}(E) \cdot \mathtt{cdr}(E) \to_{\eta\cdot} E & \mu[(x \cdot \alpha).c] \to_{\text{exp}} \mu\beta.c[\mathtt{car}(\beta)/x, \mathtt{cdr}(\beta)/\alpha]
\end{array}
$$

Figure 2. Projection Based Calculus

WHNF would print the string `"hello"`, while evaluating the η-expanded term $\lambda x.(\mathtt{print}\ \mathtt{"hello"}; (\lambda y.y))x$ would not.

3. Call Stacks as Structures

The situation of η becomes even worse when we extend the λ-calculus with control effects. Specifically, in Parigot's $\lambda\mu$-calculus [11], which extends the call-by-name λ-calculus with control abstraction and application, η-reduction breaks confluence [4]. Thus, we find even worse problems with call-by-name evaluation and extensionality in the λ-calculus with control effects.

To combat this lamentable turn of events, we will invert the way we think about functions. Instead of focusing on the way functions are created (via lambda abstractions), we will focus on the way they are used (via application). That is to say, we will take call stacks, as found in the Krivine abstract machine, as the essential structure of functions. Consider the way lambdas are implemented in the second rule of Figure 1. One interpretation of this rule is that a lambda pops off the top argument of its call stack. In other words, a lambda $\lambda x.v$ deconstructs its call stack by *pattern matching*, transforming $v' \cdot E$ into the command $\langle v[v'/x] \| E \rangle$. We capture this intuition in

$$
\begin{array}{rl}
c \in \text{Commands} ::= \langle v \| e \rangle & \langle \mu\alpha.c \| E \rangle = c[E/\alpha] \\
v \in \text{Terms} ::= x \mid \mu\alpha.c \mid \mu[(x \cdot \alpha).c] & \mu\alpha.\langle v \| \alpha \rangle = v \\
E \in \text{CoValues} ::= \alpha \mid v \cdot E & \mu[(x \cdot \alpha).\langle v \| x \cdot \alpha \rangle] = v \\
\multicolumn{2}{c}{\langle \mu[(x \cdot \alpha).c] \| v \cdot E \rangle = c[v/x, E/\alpha]}
\end{array}
$$

Figure 3. Sequent Calculus with Functions as Pattern Matching

a variant of Curien and Herbelin's sequent calculus $\bar{\lambda}\mu\tilde{\mu}$ [3] (Figure 3). This calculus contains the commands and co-values of the Krivine machine directly in the syntax of programs, and expresses control abstraction as the μ-binder, $\mu\alpha.c$, which gives a name α to its call stack before running c. Furthermore, to stress functions as deconstructions, we write them as a pattern-matching version of the μ-bindings, so that the lambda $\lambda x.v$ becomes $\mu[(x \cdot \alpha).\langle v \| \alpha \rangle]$. Thus, functions are implemented as the pattern-matching substitution of both components of a call stack, and the η law is expressed as erasing the no-op match, $\mu[(x \cdot \alpha).\langle v \| x \cdot \alpha \rangle] = v$.

Note that we have not yet solved the confluence problem: orienting the equations in Figure 3 from left to right yields a non-confluent rewriting theory. However, Nakazawa and Nagai's $\Lambda\mu_{\text{cons}}$-calculus [10] provides an interesting solution. Their calculus can be understood as replacing pattern matching on the call stack with the primitive operations $\mathtt{car}(-)$ and $\mathtt{cdr}(-)$ for getting the first and the second components. Just like how we can take apart tuples with either pattern matching or `fst` and `snd` projections, so too can we deconstruct a call stack with matching or projection. We transport this solution into the sequent calculus (Figure 2). Note that the main extensional reasoning principle is no longer a property about lambdas, but rather a reduction on call stacks (as given by the $\eta\cdot$ rule) and a transformation of pattern matching into projection (as given by exp rule [5, 9]).

It turns out that, equationally, this projection based calculus is conservative over the pattern matching calculus in Figure 3. Further, considering that the λ-calculus with surjective pairing is not confluent [6], the projection based calculus is rather amazingly confluent. Indeed, exp-normal forms point out a confluent subsyntax, previously discovered as the Stack Calculus [2], which gives a simple proof of confluence for the whole system.

$$
\begin{array}{rl}
c \in \text{Command} ::= \langle v \| E \rangle & \\
v \in \text{Terms} ::= x \mid v\ v \mid \lambda x.v \mid \mathtt{pick}^n(\mathtt{tp}) & \\
E \in \text{CoValues} ::= v \cdot E \mid \mathtt{drop}^n(\mathtt{tp}) & \\
\langle v\ v' \| E \rangle \rightsquigarrow \langle v \| v' \cdot E \rangle & \langle \lambda x.v \| v' \cdot E \rangle \rightsquigarrow \langle v[v'/x] \| E \rangle \\
\multicolumn{2}{c}{\langle \lambda x.v \| \mathtt{drop}^n(\mathtt{tp}) \rangle \rightsquigarrow \langle v[\mathtt{pick}^n(\mathtt{tp})/x] \| \mathtt{drop}^{n+1}(\mathtt{tp}) \rangle}
\end{array}
$$

Figure 4. The Head Reduction Machine for the λ-calculus

4. Head Reduction Abstract Machine

An interesting consequence of the projection based reduction theory in Figure 2 is that reduction at the "top level" of a program no longer stops when encountering a function abstraction. The top-level reduction of the sequent counterpart of $\lambda x.\Omega x$ never halts:

$$\langle \mu[(x \cdot \alpha).\langle \Omega \| x \cdot \alpha \rangle] \| \delta \rangle \to^*_{\text{exp},\mu} \langle \Omega \| \mathtt{car}(\delta) \cdot \mathtt{cdr}(\delta) \rangle \to \cdots$$

Taking advantage of this phenomenon, we now derive an implementation of the pure λ-calculus which respects the η law while never going under binders and only reducing at the top level.

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:

$$
\begin{array}{rl}
\mathtt{drop}^0(E) \triangleq E & \mathtt{pick}^n(E) \triangleq \mathtt{car}(\mathtt{drop}^n(E)) \\
\multicolumn{2}{c}{\mathtt{drop}^{n+1}(E) \triangleq \mathtt{cdr}(\mathtt{drop}^n(E))}
\end{array}
$$

Now, observe that in the control-free setting, the only co-variable we need is `tp`, which represents the "top level" of the program. It follows that for top-level reduction, $\mathtt{pick}^n(\mathtt{tp})$ and $\mathtt{drop}^n(\mathtt{tp})$ are the only call-stack projections we need in the syntax.

We now construct an abstract machine (Figure 4) for head evaluation of the λ-calculus which is exactly like the Krivine machine with one additional rule. The difference is that, unlike the Krivine machine which stops when it hits a top-level lambda, the head reduction machine "splits" the top level into a call stack and continues. Thus, the WHNF $\lambda x.(\lambda y.y)x$ at the top-level is not done:

$$
\begin{array}{rl}
\langle \lambda x.(\lambda y.y)x \| \mathtt{drop}^0(\mathtt{tp}) \rangle & \rightsquigarrow \langle (\lambda y.y)(\mathtt{pick}^0(\mathtt{tp})) \| \mathtt{drop}^1(\mathtt{tp}) \rangle \\
& \rightsquigarrow^* \langle \mathtt{pick}^0(\mathtt{tp}) \| \mathtt{drop}^1(\mathtt{tp}) \rangle
\end{array}
$$

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

$$
\begin{array}{c}
\langle v \| v' \cdot E \rangle \hookrightarrow \langle v\ v' \| E \rangle \\
\langle v \| \mathtt{drop}^{n+1}(\mathtt{tp}) \rangle \hookrightarrow \langle \lambda x.v[x/\mathtt{pick}^n(\mathtt{tp})] \| \mathtt{drop}^n(\mathtt{tp}) \rangle
\end{array}
$$

where x is not free in v in the second rule. Note that the substitution $v[x/\mathtt{pick}^n(\mathtt{tp})]$ replaces all occurrences of terms of the form $\mathtt{pick}^n(\mathtt{tp})$ inside v with x. Correctness is ensured by the fact that $c \rightsquigarrow c'$ implies $c \to^\star c'$ and $c \hookrightarrow c'$ implies $c' \to^\star c$, according to the calculus in Figure 2. Thus the abstract machine respects the equational theory of Figure 3, and therefore the λ-calculus with η.

In the end, control and the sequent calculus has revealed to us an essential aspect of pure functions, and the importance of context awareness!

References

[1] C. Böhm. Alcune proprieta delle forme β-η-normali nel λ-k-calcolo. *Pubblicazioni dell'Istituto per le Applicazioni del Calcolo*, 696, 1968.

[2] A. Carraro, T. Ehrhard, and A. Salibra. The stack calculus. In *LSFA'12*, pages 93–108, 2012.

[3] P.-L. Curien and H. Herbelin. The duality of computation. In *ICFP '00*, pages 233–243, New York, NY, USA, 2000. ISBN 1-58113-202-6.

[4] R. David and W. Py. $\lambda\mu$-calculus and Bohm's theorem. *J. Symbolic Logic*, 66(1):407–413, 03 2001.

[5] H. Herbelin. C'est maintenant qu'on calcule : Au cœur de la dualité. In *Habilitation à diriger les reserches*, 2005.

[6] J. Klop and R. de Vrijer. Unique normal forms for lambda calculus with surjective pairing. *Information and Computation*, 80(2):97 – 113, 1989. ISSN 0890-5401.

[7] J.-L. Krivine. A call-by-name lambda-calculus machine. *Higher Order Symbol. Comput.*, 20(3):199–207, Sept. 2007. ISSN 1388-3690.

[8] S. Marlow. State monads don't respect the monad laws in haskell. Haskell mailing list, May 2002.

[9] G. Munch-Maccagnoni. *Syntax and Models of a non-Associative Composition of Programs and Proofs*. PhD thesis, Univ. Paris Diderot, 2013.

[10] K. Nakazawa and T. Nagai. Reduction system for extensional lambda-mu calculus. In *RTA-TLCA*, pages 349–363, 2014.

[11] M. Parigot. Lambda-my-calculus: An algorithmic interpretation of classical natural deduction. In *LPAR*, pages 190–201, 1992.

A. Closure Conversion

The Krivine machine we gave at the beginning used substitution to handle variables in order to simplify the presentation. Sometimes though, we would rather consider abstract machines which use explicit environments and closures to avoid the need to do substitutions at run-time. Thus, we consider the Krivine machine with commands of the form $\langle v|\sigma|E\rangle$, where σ is an extra field representing the current environment. We write an environment as a list of bindings—associating variables with closures—where a closure is a pair of a term and an environment. Environments are treated abstractly, and must support the usual extension operation, written $(x \mapsto t) :: \sigma$, where a new variable-closure binding is added and variable lookup operation, written $\sigma(x)$. The definition of call stacks must be slightly modified to accommodate storing closures as function arguments and not just terms. The resulting abstract

$$
\begin{aligned}
c \in \text{Commands} &::= \langle v|\sigma|E\rangle \\
v \in \text{Terms} &::= x \mid \lambda x.v \mid v\ v \\
E \in \text{CoValues} &::= \mathtt{tp} \mid t \cdot E \\
\sigma \in \text{Environments} &::= [] \mid (x \mapsto t) :: \sigma \\
t \in \text{Closures} &::= (v, \sigma) \\
\langle v\ v'|\sigma|E\rangle &\rightsquigarrow \langle v|\sigma|(v', \sigma) \cdot E\rangle \\
\langle \lambda x.v|\sigma|t \cdot E\rangle &\rightsquigarrow \langle v|(x \mapsto t) :: \sigma|E\rangle \\
\langle x|\sigma|E\rangle &\rightsquigarrow \langle v|\sigma'|E\rangle \quad \text{where } \sigma(x) = (v, \sigma')
\end{aligned}
$$

Figure 5. Krivine Machine with Environments

machine, given in Figure 5, manages the environment during reduction. Instead of substitution, a bound variable is interpreted by looking it up in the current environment.

The same technique allows us to implement the head reduction machine without substitutions, shown in Figure 6. Of course, various alternative designs are possible such, as ones based on De Bruijn indices. However, the essential core of the machines remain the same.

$$
\begin{aligned}
c \in \text{Commands} &::= \langle v|\sigma|E\rangle \\
v \in \text{Terms} &::= x \mid \lambda x.v \mid v\ v \mid \mathtt{pick}^n(\mathtt{tp}) \\
E \in \text{CoValues} &::= \mathtt{drop}^n(\mathtt{tp}) \mid t \cdot E \\
\sigma \in \text{Environments} &::= [] \mid (x \mapsto t) :: \sigma \\
t \in \text{Closures} &::= (v, \sigma) \\
\langle v\ v'|\sigma|E\rangle &\rightsquigarrow \langle v|\sigma|(v', \sigma) \cdot E\rangle \\
\langle \lambda x.v|\sigma|t \cdot E\rangle &\rightsquigarrow \langle v|(x \mapsto t) :: \sigma|E\rangle \\
\langle \lambda x.v|\sigma|\mathtt{drop}^n(\mathtt{tp})\rangle &\rightsquigarrow \langle v|(x \mapsto (\mathtt{pick}^n(\mathtt{tp}), \sigma)) :: \sigma|\mathtt{drop}^{n+1}(\mathtt{tp})\rangle \\
\langle x|\sigma|E\rangle &\rightsquigarrow \langle v|\sigma'|E\rangle \quad \text{where } \sigma(x) = (v, \sigma')
\end{aligned}
$$

Figure 6. Head Reduction Machine with Environments

B. Machine for Head Normalization with Control

The head normalization machine we gave was motivated by control, but it did not implement a λ-calculus with control. Of course, the projection based rewriting theory shows us that we can implement control in an extensional way without reducing inside of binders. However, it turns out that we do not need to use the full feature set of the projection based calculus in order to achieve extensional reduction with control. We observe that the only time we ever need to use projections on the call stack, rather than pattern matching, is when dealing with the top-level, and this is still true in the setting with control. Thus, we give a small modification of our head reduction machine that performs extensional head reduction for a call-by-name lambda calculus with control (Figure 7).

$$
\begin{aligned}
c \in \text{Command} &::= \langle v \| E\rangle \\
v \in \text{Terms} &::= x \mid v\ v \mid \lambda x.v \mid \mathtt{pick}^n(\mathtt{tp}) \mid \mu\alpha.c \\
E \in \text{CoValues} &::= v \cdot E \mid \mathtt{drop}^n(\mathtt{tp}) \mid \alpha \\
\langle v\ v' \| E\rangle &\rightsquigarrow \langle v \| v' \cdot E\rangle \\
\langle \lambda x.v \| v' \cdot E\rangle &\rightsquigarrow \langle v[v'/x] \| E\rangle \\
\langle \lambda x.v \| \mathtt{drop}^n(\mathtt{tp})\rangle &\rightsquigarrow \langle v[\mathtt{pick}^n(\mathtt{tp})/x] \| \mathtt{drop}^{n+1}(\mathtt{tp})\rangle \\
\langle \mu\alpha.c \| E\rangle &\rightsquigarrow c[E/\alpha]
\end{aligned}
$$

Figure 7. The Head Reduction Machine for the λ-calculus with control

We make the syntactic restriction that we only ever project out of the top level context $\mathtt{tp}$ and not out of bound co-variables internal to the program. This means that we never have to deal with the situation where we substitute into a projection operation. Doing so lets us syntactically prohibit co-values like $\mathtt{drop}^n(v \cdot E)$ or $\mathtt{drop}^n(\mathtt{drop}^m(E))$, and thus we do not need reduction rules to deal with them. Of course, that means there is no reduction rule that can apply at the top level of a command such as

$$\langle \lambda x.v \| \alpha\rangle$$

but in a machine for evaluating closed programs, we will never encounter these. Additionally, we can assume that our source program does not contain any call stacks, which are unnecessary for writing programs since the usual syntax for function application is available. Thus, we see that extensional execution of closed programs with control can be achieved using the standard interpretation for functions everywhere, except when dealing with the top-level context. Note that the read-back relation:

$$
\begin{aligned}
\langle v \| v' \cdot E\rangle &\hookrightarrow \langle v\ v' \| E\rangle \\
\langle v \| \mathtt{drop}^{n+1}(\mathtt{tp})\rangle &\hookrightarrow \langle \lambda x.v[x/\mathtt{pick}^n(\mathtt{tp})] \| \mathtt{drop}^n(\mathtt{tp})\rangle \\
&\text{where } x \notin FV(v)
\end{aligned}
$$

is unchanged by the addition of control.

It seems that everything works out rather simply: it took only a single additional rule to convert the Krivine machine into a machine for performing head reduction (and thus respecting η). Similarly, it takes only a single additional rule to add control to the Krivine machine. Adding both these rules produces a machine for call-by-name head reduction with control which respects η.

Finally, we can give a variant of the head reduction machine with control using environments instead of substitutions. The only complexity is that we need to keep track of separate environments for interpreting the variables and co-variables in the term and co-value of a command. Again, we assume that the source program does not contain any syntactic call stacks, which now hold closures as function arguments instead of terms, and which are a run-time construct of the machine.

$$
\begin{aligned}
\text{Programs} &::= \langle v|\sigma|\sigma|E\rangle \\
c \in \text{Commands} &::= \langle v \| E\rangle \\
v \in \text{Terms} &::= x \mid \lambda x.v \mid v\ v \mid \mathtt{pick}^n(\mathtt{tp}) \mid \mu\alpha.c \\
E \in \text{CoValues} &::= \mathtt{drop}^n(\mathtt{tp}) \mid t \cdot E \mid \alpha \\
\sigma \in \text{Environments} &::= [] \mid (x \mapsto t) :: \sigma \mid (\alpha \mapsto s) :: \sigma \\
t \in \text{Closures} &::= (v, \sigma) \\
s \in \text{CoClosures} &::= (E, \sigma) \\
\langle v\ v'|\sigma|\sigma'|E\rangle &\rightsquigarrow \langle v|\sigma|\sigma'|(v', \sigma) \cdot E\rangle \\
\langle \lambda x.v|\sigma|\sigma'|t \cdot E\rangle &\rightsquigarrow \langle v|(x \mapsto t) :: \sigma|\sigma'|E\rangle \\
\langle \lambda x.v|\sigma|\sigma'|\mathtt{drop}^n(\mathtt{tp})\rangle &\rightsquigarrow \langle v|(x \mapsto (\mathtt{pick}^n(\mathtt{tp}), \sigma')) :: \sigma|\sigma'|\mathtt{drop}^{n+1}(\mathtt{tp})\rangle \\
\langle x|\sigma|\sigma'|E\rangle &\rightsquigarrow \langle v|\sigma''|\sigma'|E\rangle \quad \text{where } \sigma(x) = (v, \sigma'') \\
\langle \lambda x.v|\sigma|\sigma'|\alpha\rangle &\rightsquigarrow \langle \lambda x.v|\sigma|\sigma''|E\rangle \quad \text{where } \sigma'(\alpha) = (E, \sigma'') \\
\langle \mu\alpha.\langle v \| E\rangle|\sigma|\sigma'|E'\rangle &\rightsquigarrow \langle v|\sigma''|\sigma''|E\rangle \\
&\qquad \text{where } \sigma'' = (\alpha \mapsto (E', \sigma')) :: \sigma
\end{aligned}
$$

Figure 8. Head Reduction Machine with Control using Environments