Modular Denotational Semantics for Compiler Construction *

Sheng Liang   Paul Hudak

Yale University
Department of Computer Science
New Haven, CT 06520-8285
{liang-sheng, hudak}@cs.yale.edu

September 1995

Abstract

We show the benefits of applying *modular monadic semantics* to compiler construction. Modular monadic semantics allows us to define a language with a rich set of features from reusable building blocks, and use program transformation and equational reasoning to improve code. Compared to denotational semantics, reasoning in monadic style offers the added benefits of highly modularized proofs and more widely applicable results. To demonstrate, we present an axiomatization of environments, and use it to prove the correctness of a well-known compilation technique. The monadic approach also facilitates generating code in various target languages with different sets of built-in features.

Keywords: denotational semantics, monads, compiler generation.

*This work was supported by the Advanced Research Project Agency and the Office of Naval Research under Arpa Order 8888, Contract N00014-92-C-0153.

1 Introduction

We propose a modular semantics which allows language designers to:

- add (or remove) programming language features without causing global changes to the existing specification,
- derive a compilation scheme from semantic descriptions,
- prove the correctness of program transformation and compilation strategies, and
- generate code in a variety of target languages with different sets of built-in features.

Our goals are similar to that of Action Semantics [24] and related efforts by, for example, Wand [34], Lee [17], Appel & Jim [1], and Kelsey & Hudak [15]. None of the existing approaches are completely satisfactory in achieving the above goals. For example, it has long been recognized that traditional denotational semantics [31] is not suitable for compiler generation for a number of crucial reasons [17], among which is the lack of modularity and extensibility.

We take advantage of a new development in programming language theory, a monadic approach [21] to structured denotational semantics, that achieves a high level of modularity and extensibility. The source language we consider in this paper has a variety of features, including both call-by-name and call-by-value versions of functions:

$$
\begin{array}{lcll}
e & ::= & n \mid e_1 + e_2 & \text{(arithmetic operations)} \\
 & \mid & v \mid \lambda v.e \mid (e_1\ e_2)_n \mid (e_1\ e_2)_v & \text{(cbn and cbv functions)} \\
 & \mid & \text{callcc} & \text{(first-class continuations)} \\
 & \mid & e_1 := e_2 \mid \text{ref}\ e \mid \text{deref}\ e & \text{(imperative features)}
\end{array}
$$

Figure 1 shows how our modular monadic semantics is organized. Language designers specify semantic modules by using a set of "kernel-level" operations. The expression "$e_1 := e_2$", for example, is interpreted by the low-level primitive operation "update".

While it is a well-known practice to base programming language semantics on a kernel-language, the novelty of our approach lies in how the kernel-level primitive operations are organized. In our framework, depending on how much support the upper layers need, any set of primitive operations can be put together in a modular way using an abstraction mechanism called *monad transformers* [21] [18]. Monad transformers provide the power needed to represent the abstract notion of programming language features, but still allow us to access low-level semantic details. In fact, since monad transformers are defined as higher-order functions, our monadic semantics is no more than a structured version of denotational semantics, and all conventional reasoning methods (such as β substitution) apply.

We will investigate how an interpreter based on the modular monadic semantics can be turned into a compiler. In Section 2, we will define a compositional high-level semantics for our source language which guarantees that we can unfold all recursive calls to the evaluator, and thus avoid the overhead of dispatching on the abstract syntax tree. In section 3, we show how monad laws and axioms can be used to optimize intermediate code. To demonstrate the reasoning powers of monad transformers, in Section 4 we generalize Wand's [35] proof of the correctness of a well-known technique to overcome the overhead of dynamic variable lookups by transforming variables in the source language into variables in the meta-language. In section 5 we discuss how to utilize the modularity provided by monad transformers to target different languages.

The contributions of this paper are:

- proposing a monad-based modular approach to semantics directed compiler generation,
- applying monad laws to program transformation,

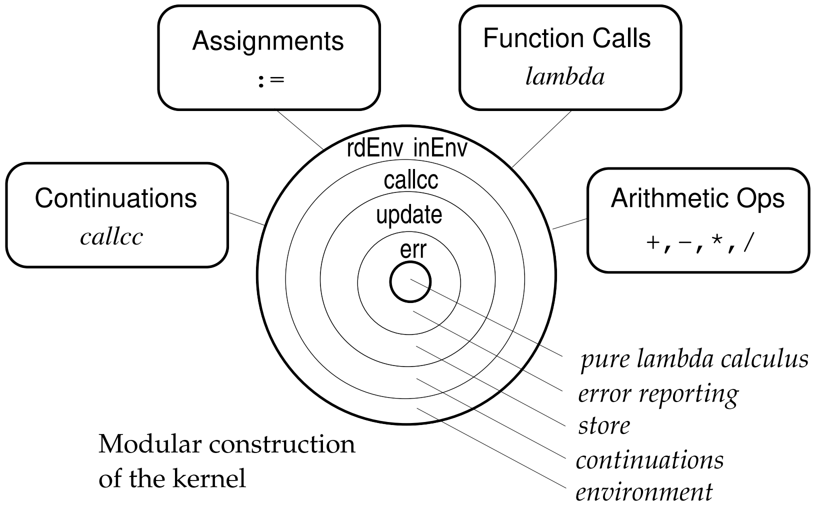

Figure 1: The organization of modular monadic semantics

- presenting a monadic style axiomatization of environments,
- demonstrating that reasoning in monadic style enables us to better structure proofs and obtain more general results than in denotational semantics, and
- using monad transformer properties (for example, naturality of liftings) to utilize target language features.

We have a prototype implementation of our system which takes a high-level semantic description and generates efficient Scheme and SML/NJ code.

We present our results in the traditional denotational semantics style [31], augmented with a Haskell-like [11] type declaration syntax to express monads as type constructors. No prior knowledge of monads is assumed.

2 A Modular Monadic Semantics

In this section we use some of the results from our earlier work on modular interpreters [18] to define a modular semantics for our source language.

2.1 A High-level Monadic Semantics

Traditional denotational semantics maps, say, a term, an environment and a continuation to an answer. In contrast, monadic semantics maps terms to computations, where the details of the environment, store, etc. are "hidden". Specifically, our semantic evaluation function E has type:

$$E \quad : \quad \mathit{Term} \rightarrow \mathit{Compute\ Value}$$

where *Value* denotes the result of the computation. The type constructor *Compute* is called a *monad*. It abstracts away the details of a computation, exposing only the result type. We will define monads more formally later, but for now we note that *Compute* comes equipped with two basic operations:

$$\begin{array}{lll} \mathsf{then} & : & \mathit{Compute}\ a \rightarrow (a \rightarrow \mathit{Compute}\ b) \rightarrow \mathit{Compute}\ b \\ \mathsf{return} & : & a \rightarrow \mathit{Compute}\ a \end{array}$$

We usually write "then" in an infix form. Intuitively, "c_1 then $\lambda v.c_2$" is a computation that first computes c_1, binds the result to v, and then computes c_2. "Return v" is a trivial computation that simply returns v as result.

In the standard semantics, *Value* is the domain sum of basic values and functions. Functions map computations to computations:[1]

$$\begin{array}{lcl} \textbf{type}\ \mathit{Fun} & = & \mathit{Compute\ Value} \rightarrow \mathit{Compute\ Value} \\ \textbf{type}\ \mathit{Value} & = & \mathit{Int} + \mathit{Bool} + \mathit{Addr} + \mathit{Fun} + \ldots \end{array}$$

The standard semantics for arithmetic expressions is as follows:

$$\begin{array}{lcll} E[\![n]\!] & = & \mathsf{return}(n\ \textbf{in}\ \mathit{Value}) & \\ E[\![e_1 + e_2]\!] & = & E[\![e_1]\!]\ \mathsf{then}\ \lambda v_1. & \\ & & E[\![e_2]\!]\ \mathsf{then}\ \lambda v_2. & \\ & & \textbf{if}\ \mathsf{checkType}(v_1, v_2) & \textbf{then}\ \mathsf{err}\ \text{"\textit{type error}"} \\ & & & \textbf{else}\ \mathsf{return}((v_1 \mid \mathit{Int}) + (v_2 \mid \mathit{Int})\ \textbf{in}\ \mathit{Value}) \end{array}$$

We use a *primitive monadic combinator* (a semantic function directly supported by the underlying monad *Compute*):

$$\mathsf{err} \ : \ \mathit{String} \rightarrow \mathit{Compute}\ a$$

to report type errors. For clarity, from now on we will omit domain injection/projection and type checking.

"$E[\![n]\!]$" just returns the number n (injected into the *Value* domain) as the result of a trivial computation. To evaluate "$e_1 + e_2$", we evaluate e_1 and e_2 in turn, and then sum the results. In denotational semantics, the interpretations for arithmetic expressions are slightly different depending on whether we are passing an environment around, or whether we write in direct or continuation-passing styles. *In contrast, our monadic semantics for arithmetic expressions stays the same no matter what details of computation (e.g., continuations, environments, states) are captured in the underlying monad.*

Function abstractions and applications need access to an environment *Env* which maps variable names to computations, and two more primitive monadic combinators which retrieve the current environment and perform a computation in a given environment, respectively:

$$\begin{array}{lcl} \textbf{type}\ \mathit{Env} & = & \mathit{Name} \rightarrow \mathit{Compute\ Value} \\ \mathsf{rdEnv} & : & \mathit{Compute\ Env} \\ \mathsf{inEnv} & : & \mathit{Env} \rightarrow \mathit{Compute\ Value} \rightarrow \mathit{Compute\ Value} \end{array}$$

The standard semantics for functions is as follows:

$$\begin{array}{lcl} E[\![v]\!] & = & \mathsf{rdEnv}\ \mathsf{then}\ \lambda\rho.\rho\ v \\ E[\![\lambda v.e]\!] & = & \mathsf{rdEnv}\ \mathsf{then}\ \lambda\rho.\ \mathsf{return}(\lambda c.\mathsf{inEnv}\ (\rho[c/v])\ E[\![e]\!]) \\ E[\![(e_1\ e_2)_n]\!] & = & E[\![e_1]\!]\ \mathsf{then}\ \lambda f.\mathsf{rdEnv}\ \mathsf{then}\ \lambda\rho.f(\mathsf{inEnv}\ \rho\ E[\![e_2]\!]) \\ E[\![(e_1\ e_2)_v]\!] & = & E[\![e_1]\!]\ \mathsf{then}\ \lambda f.E[\![e_2]\!]\ \mathsf{then}\ \lambda v.f(\mathsf{return}\ v) \end{array}$$

The difference between call-by-value and call-by-name is clear: the former reduces the argument before invoking the function, whereas the latter packages the argument with the current environment to form a closure.

To simplify the presentation somewhat, we assume that imperative features can be defined using the primitive monad combinator:

$$\mathsf{update} \ : \ (\mathit{Store} \rightarrow \mathit{Store}) \rightarrow \mathit{Compute\ Store}$$

[1] This generality allows us to model both call-by-name and call-by-value.

for some suitably chosen *Store*. We can read the store by passing update the identity function, and change the store by passing it a state transformer. Although update returns the entire state, properly defined store-manipulating functions can guarantee that the store is never duplicated (see, for example, [28]).

With the kernel-level function:

$$\mathsf{callcc} \quad : \quad ((\mathit{Value} \rightarrow \mathit{Compute\ Value}) \rightarrow \mathit{Compute\ Value}) \rightarrow \mathit{Compute\ Value}$$

The semantics of "callcc" is a function expecting another function as an argument, to which the current continuation will be passed:

$$E[\![\text{callcc}]\!] \quad = \quad \mathsf{return}(\lambda f.f \ \mathsf{then}\ \lambda f'.\mathsf{callcc}(\lambda k.f'(\lambda a.a \ \mathsf{then}\ k)))$$

Our high-level monadic semantics somewhat resembles to action semantics, except that it uses only "then" and "return" to thread computations. Together with primitive monadic combinators, these two operations are powerful enough to model various kinds of control flows (e.g., error handling, function calls and callcc) in sequential languages. Like in action semantics, we make an effort to give a high-level view of the source language semantics.

We require that a semantics specified in terms of monadic combinators be *compositional*: the arguments in recursive calls to E are substructures of the argument received on the left-hand side. From a theoretical point of view, it makes inductive proofs on program structures possible. In practice, this guarantees that given any abstract syntax tree, we can recursively unfold all calls to the interpreter, effectively removing the run-time dispatch on the abstract syntax tree.

2.2 Constructing the *Compute* Monad

It is clear that monad *Compute* needs to support the following primitive monad combinators:

$$\begin{array}{lcl}
\mathsf{err} & : & \mathit{String} \rightarrow \mathit{Compute}\ a \\
\mathsf{rdEnv} & : & \mathit{Compute\ Env} \\
\mathsf{inEnv} & : & \mathit{Env} \rightarrow \mathit{Compute\ Value} \rightarrow \mathit{Compute\ Value} \\
\mathsf{update} & : & (\mathit{Store} \rightarrow \mathit{Store}) \rightarrow \mathit{Compute\ Store} \\
\mathsf{callcc} & : & ((\mathit{Value} \rightarrow \mathit{Compute\ Value}) \rightarrow \mathit{Compute\ Value}) \rightarrow \mathit{Compute\ Value}
\end{array}$$

If we follow the traditional denotational semantics approach, now is the time to set up domains and implement the above functions. The major drawback of such monolithic approach is that if we add some source language features later on, all the functions may have to be redefined.

For the sake of modularity, we start from a simple monad and add more and more features. The simplest monad of all is the identity monad. All it captures is function application:

$$\begin{array}{lcl}
\textbf{type}\ \mathit{Id}\ a & = & a \\
\\
\mathsf{return}_{\mathit{Id}}\ x & = & x \\
c\ \mathsf{then}_{\mathit{Id}}\ f & = & f\ c
\end{array}$$

A *monad transformer* takes a monad, and returns a new monad with added features. For example, "*StateT* s" adds a state s to any monad m:

$$\begin{array}{lcl}
\textbf{type}\ \mathit{StateT}\ s\ m\ a & = & s \rightarrow m\ (s, a) \\
\\
\mathsf{return}_{(\mathit{StateT}\ s\ m)}\ x & = & \lambda s.\ \mathsf{return}_m(s, x) \\
c\ \mathsf{then}_{(\mathit{StateT}\ s\ m)}\ k & = & \lambda s_0.c\ s_0\ \mathsf{then}_m\ \lambda(s_1, a).k\ a\ s_1
\end{array}$$

To see how monad transformers work, let us apply *StateT* to the identity monad *Id*:

Continuation:

$$\textbf{type } \mathit{ContT\ ans}\ m\ a = (a \to m\ \mathit{ans}) \to m\ \mathit{ans}$$

$$\begin{array}{l}
\mathsf{return}_{(\mathit{ContT\ ans}\ m)}\ a = \lambda k.ka \\
c\ \mathsf{then}_{(\mathit{ContT\ ans}\ m)}\ f = \lambda k.c(\lambda a.fak) \\
\mathsf{callcc}\ f \quad = \quad \lambda k.f(\lambda a.\lambda k'.ka)k
\end{array}$$

Environment:

$$\textbf{type } \mathit{EnvT}\ e\ m\ a = e \to m\ a$$

$$\begin{array}{lcl}
\mathsf{return}_{(\mathit{EnvT}\ e\ m)}\ a & = & \lambda\rho.\ \mathsf{return}_m\ a \\
c\ \mathsf{then}_{(\mathit{EnvT}\ e\ m)}\ k & = & \lambda\rho.c\rho\ \mathsf{then}_m\ \lambda a.ka\rho \\
\mathsf{rdEnv} & = & \lambda\rho.\ \mathsf{return}_m\ \rho \\
\mathsf{inEnv}\ \rho\ c & = & \lambda\rho'.c\ \rho
\end{array}$$

Errors:

$$\begin{array}{lcl}
\textbf{type } \mathit{Error}\ a & = & \mathit{Ok}\ a \mid \mathit{Error\ String} \\
\textbf{type } \mathit{ErrorT}\ m\ a & = & m\ (\mathit{Error}\ a) \\
\mathsf{return}_{(\mathit{ErrorT}\ m)}\ a & = & \mathsf{return}_m \cdot \mathit{Ok} \\
c\ \mathsf{then}_{(\mathit{ErrorT}\ m)}\ k & = & c\ \mathsf{then}_m\ \lambda a.\ \textbf{case}\ a\ \textbf{of} \\
 & & \quad (\mathit{Ok}\ x) \to kx \\
 & & \quad (\mathit{Error\ msg}) \to \mathsf{return}_m(\mathit{Error\ msg})
\end{array}$$

$$\mathsf{err} \quad = \quad \mathsf{return}_m \cdot \mathit{Error}$$

Figure 2: Monad transformers

$$\begin{array}{lcl}
\textbf{type } \mathit{StateT}\ s\ \mathit{Id}\ a & = & s \to \mathit{Id}\ (s, a) \\
 & = & s \to (s, a) \\
 & & \\
\mathsf{return}_{(\mathit{StateT}\ s\ \mathit{Id})}\ x & = & \lambda s.\ \mathsf{return}_{\mathit{Id}}(s, x) \\
 & = & \lambda s.(s, x) \\
c\ \mathsf{then}_{(\mathit{StateT}\ s\ \mathit{Id})}\ k & = & \lambda s_0.c\ s_0\ \mathsf{then}_{\mathit{Id}}\ \lambda(s_1, a).k\ a\ s_1 \\
 & = & \lambda s_0.\mathrm{let}\ (s_1, a) = m\ s_0\ \mathrm{in}\ k\ a\ s_1
\end{array}$$

Note that "*StateT* s *Id*" is the standard state monad found, for example, in Wadler's work [32].

To make the newly introduced state accessible, "*StateT* s" introduces update on s which applies f to the state, and returns the old state:

$$\begin{array}{lcl}
\mathsf{update} & : & (s \to s) \to \mathit{StateT}\ s\ m\ a \\
\mathsf{update}\ f & = & \lambda s.\ \mathsf{return}(f\ s, s)
\end{array}$$

Figure 2 gives the definitions of several other monad transformers, including those for errors (*ErrorT*), continuations (*ContT*) and environments (*EnvT*). Now we can construct *Compute* by applying a series of monad transformers to the base monad *Id*:

$$\textbf{type } \mathit{Compute}\ a = \mathit{EnvT\ Env}\ (\mathit{ContT\ Answer}\ (\mathit{StateT\ Store}\ (\mathit{ErrorT\ Id})))\ a$$

Env, *Store* and *Answer* are the type of the environment, store and answer, respectively.

One issue remains to be addressed. The update function introduced by *StateT* does not work on *Compute*, which contains features added later by other monad transformers. In general, this is the problem of *lifting* operations through monad transformers. Figure 3 gives a brief summary of useful liftings (See [18] for a detailed description.) For example, in the *Compute* monad above, "update f" is:

$$\lambda s.\mathit{Ok}\ (f\ s, s)$$

when first introduced by *StateT*. After *Compute* is finally constructed, "update f" becomes:

$$\lambda\rho.\lambda k.\lambda s.k\ s\ (f\ s)$$

In summary, monad transformers allow us to easily construct monads with a certain set of primitive monadic combinators, defined as higher-order functions.

Every monad transformer t has a function:

$$\mathsf{lift}_{(t\ m)} \quad : \quad m\ a \rightarrow t\ m\ a$$

which embeds a computation in monad m into "$t\ m$". Functions err, update and rdEnv are easily lifted using lift:

$$\begin{array}{lcl} \mathsf{err}_{(t\ m)} & = & \mathsf{lift}_{(t\ m)} \cdot \mathsf{err}_m \\ \mathsf{update}_{(t\ m)} & = & \mathsf{lift}_{(t\ m)} \cdot \mathsf{update}_m \\ \mathsf{rdEnv}_{(t\ m)} & = & \mathsf{lift}_{(t\ m)} \mathsf{rdEnv}_m \end{array}$$

Some liftings of callcc, inEnv and the definitions of lift for each monad transformer are listed in the following table:

$t\ m$	$\mathsf{callcc}_{(t\ m)}\ f$	$\mathsf{inEnv}_{(t\ m)}\ \rho\ c$	$\mathsf{lift}_{(t\ m)}\ c$
$\mathit{EnvT}\ e\ m$	$\lambda\rho.\mathsf{callcc}_m(\lambda k.f(\lambda a.\lambda\rho'.ka)\rho)$	$\lambda\rho'.\mathsf{inEnv}_m\ \rho\ (c\rho')$	$\lambda\rho.c$
$\mathit{ContT}\ ans\ m$			$\lambda k.c\ \mathsf{then}_m\ k$
$\mathit{StateT}\ s\ m$	$\lambda s_0.\mathsf{callcc}_m(\lambda k.\ f(\lambda a.\lambda s_1.k(s_1, a))s_0)$	$\lambda s.\mathsf{inEnv}_m\ \rho\ (cs)$	$\lambda s.c\ \mathsf{then}_m\ \lambda x.\ \mathsf{return}_m(s, x)$
$\mathit{ErrorT}\ m$	$\mathsf{callcc}_m\ (\lambda k.f(\lambda a.k(\mathit{Ok}\ a)))$	$\mathsf{inEnv}_m\ \rho\ c$	$\mathsf{map}_m\ \mathit{Ok}$

Figure 3: Liftings

3 Using Monad Laws to Transform Programs

Following the monadic semantics presented in the previous section, by unfolding all calls to the semantic function E, we can transform source-level programs into monadic-style code. For example, "$((\lambda x.x + 1)\ 2)_v$" is transformed to:

$$\begin{array}{l} \mathsf{rdEnv\ then}\ \lambda\rho. \\ \mathsf{return}(\lambda c.\mathsf{inEnv}\ \ (\rho[c/\text{"}x\text{"}]) \\ \qquad\qquad\qquad (\mathsf{rdEnv\ then}\ \lambda\rho. \\ \qquad\qquad\qquad\ \ \rho\text{"}x\text{"}\ \mathsf{then}\ \lambda v_1. \\ \qquad\qquad\qquad\ \ \mathsf{return}\,1\ \mathsf{then}\ \lambda v_2. \\ \qquad\qquad\qquad\ \ \mathsf{return}(v_1 + v_2)))\ \mathsf{then}\ \lambda f. \\ \mathsf{return}\,2\ \mathsf{then}\ \lambda v. \\ f(\mathsf{return}\ v) \end{array}$$

In this section we formally introduce monads and their laws, and show how to use the laws to simplify the above program.

3.1 Monads and Monad Laws

Definition 3.1 A **monad** M is a type constructor, together with two operations:

$$\begin{array}{lcl} \mathsf{then} & : & M\ a \rightarrow (a \rightarrow M\ b) \rightarrow M\ b \\ \mathsf{return} & : & a \rightarrow M\ a \end{array}$$

satisfying the following laws [32]:

$$\begin{array}{rcll} (\mathsf{return}\,a)\ \mathsf{then}\ k & = & k\ a & \text{(left unit)} \\ c\ \mathsf{then}\ \ \mathsf{return} & = & c & \text{(right unit)} \\ c_1\ \mathsf{then}\ \lambda v_1.(c_2\ \mathsf{then}\ \lambda v_2.c_3) & = & (c_1\ \mathsf{then}\ \lambda v_1.c_2)\ \mathsf{then}\ \lambda v_2.c_3 & \text{(associativity)} \end{array}$$

Intuitively, the (left and right) unit laws say that trivial computations can be skipped in certain contexts; and the associativity law captures the very basic property of sequencing, one that we usually take for granted in programming languages.

We can verify, by equational reasoning, for example, that return_{Id} and then_{Id} satisfy the above laws, and *EnvT*, *ContT* etc. indeed transform monads to monads.

3.2 Applying Monad Laws to Program Transformation

Monad laws are useful for transforming (and optimizing) monadic style intermediate programs. For example, our compiler translates the expression "$2 + 3$" to:

$$\mathsf{return}\,2\ \mathsf{then}\ \lambda v_1.\ \mathsf{return}\,3\ \mathsf{then}\ \lambda v_2.\ \mathsf{return}(v_1 + v_2)$$

We can apply the left unit law twice, and reduce the above to:

$$\mathsf{return}(2 + 3)$$

which can of course be further optimized to "$\mathsf{return}\,5$".

Each application of a monad law usually corresponds to a number of β reductions. Monad laws allow us to perform β reductions at the "right" places, and avoid those corresponding to actual computations in the source program (such as the final reduction of "$2 + 3$" to "5" in the above example), which in turn may lead to non-termination.

Without knowledge about the environment-handling operations inEnv and rdEnv, however, monad laws alone can only simplify the example in the beginning of the section to:

$$\begin{array}{l}
\mathsf{rdEnv}\ \mathsf{then}\ \lambda\rho.\\
(\lambda c.\mathsf{inEnv}\ \ (\rho[c/\text{“}x\text{”}])\\
\qquad\qquad (\mathsf{rdEnv}\ \mathsf{then}\ \lambda\rho.\\
\qquad\qquad\ \ \rho\text{“}x\text{”}\ \mathsf{then}\ \lambda v.\\
\qquad\qquad\ \ \mathsf{return}(v + 1)))\\
(\mathsf{return}\,2))
\end{array}$$

To further simplify the above program, we need to look at the laws environment-related operations should satisfy.

3.3 Environment Axioms

We axiomatize the environment-manipulating functions as follows:

Definition 3.2 Monad M is an **environment monad** if it has two operations: rdEnv and inEnv, which satisfy the following axioms:

$$\begin{array}{rcll}
(\mathsf{inEnv}\,\rho)\cdot\mathsf{return} & = & \mathsf{return} & \text{(unit)}\\
\mathsf{inEnv}\,\rho\ (c_1\ \ \mathsf{then}\ \ \lambda v.c_2) & = & \mathsf{inEnv}\,\rho\ c_1\ \mathsf{then}\ \lambda v.\mathsf{inEnv}\,\rho\ c_2 & \text{(distribution)}\\
\mathsf{inEnv}\,\rho\ \mathsf{rdEnv} & = & \mathsf{return}\,\rho & \text{(cancellation)}\\
\mathsf{inEnv}\,\rho\ (\mathsf{inEnv}\,\rho'\ e) & = & \mathsf{inEnv}\,\rho'\ e & \text{(overriding)}
\end{array}$$

Intuitively, a trivial computation cannot depend on the environment (the unit law); the environment stays the same across a sequence of computations (the distribution law); the environment does not change between a set and a read if there are no intervening computations (the cancellation law); and an inner environment supercedes an outer one (the overriding law).

Proposition 3.1 *The monads supporting* rdEnv *and* inEnv *constructed using monad transformers* ErrorT, EnvT, StateT *and* ContT *are environment monads.*

As with the monad laws, the environment axioms can be verified by equational reasoning.

Equipped with the environment axioms, we can further transform the example monadic code to:

$$\begin{array}{l}\textsf{rdEnv then } \lambda\rho. \\ (\lambda c.c \textsf{ then } \lambda v.\,\textsf{return}(v+1))(\textsf{return}\,2)\end{array}$$

Note that explicit environment accesses have disappeared. Instead, the meta-language environment is directly used to support function calls. This is exactly what many partial evaluators achieve when they transform interpreters to compilers.

Once again note that the true computation in the original expression "$((\lambda x.x+1)\ 2)_v$" is left unreduced. In the traditional denotational semantics framework, it is harder to distinguish the redexes introduced by the compilation process from computations in the source program. In the above example, we could safely further reduce the intermediate code:

$$\begin{array}{lll} & (\lambda c.c \textsf{ then } \lambda v.\,\textsf{return}(v+1))(\textsf{return}\,2) & \\ \Rightarrow & \textsf{return}\,2 \textsf{ then } \lambda v.\,\textsf{return}(v+1) & (\beta) \\ \Rightarrow & \textsf{return}\,3 & (\text{left unit}) \end{array}$$

However, in general, unrestricted β reduction for arbitrary source programs could result in unwanted compile-time exceptions, such as in the case of "$((\lambda x.10/x)\ 0)_v$"

4 Using Monad Laws to Reason about Computations

We successfully transformed away the explicit environment in the above example, but can we do the same for arbitrary source programs? It turns out that we can indeed prove such a general result by using monad laws and environment axioms. We follow Wand [35], define a "natural semantics" which translates source language variables to lexical variables in the meta-language, and prove that it is equivalent to the standard semantics.

4.1 A Natural Semantics

We adopt Wand's definition of a *natural semantics* (different from Kahn's [5] notion) to our functional sub-language. For any source language variable name v, we assume there is a corresponding variable name v_m in the meta-language.

Definition 4.1 A **natural semantics** uses the environment of the meta-language for variables in the source language, and is given as follows:

$$\begin{array}{lll} N[\![v]\!] & = & v_m \\ N[\![\lambda v.e]\!] & = & \textsf{return}(\lambda v_m.N[\![e]\!]) \\ N[\![(e_1\ e_2)_n]\!] & = & N[\![e_1]\!] \textsf{ then } \lambda f.f(N[\![e_2]\!]) \\ N[\![(e_1\ e_2)_v]\!] & = & N[\![e_1]\!] \textsf{ then } \lambda f.N[\![e_2]\!] \textsf{ then } \lambda v.f(\textsf{return}\ v) \end{array}$$

Other source-level constructs, such as $+$, $:=$, and callcc, do not explicitly deal with the environment, and have the same natural semantics as standard semantics.

4.2 Correspondence between Natural and Standard Semantics

The next theorem, a variation of Wand's [35], guarantees that it is safe to implement function calls in the source language using the meta-language environment.

Theorem 4.1 *Let e be a program in the source language, $E[\![e]\!]$ be its standard semantics in an environment monad, $N[\![e]\!]$ be its natural semantics in the same monad,*[2] *and ρ be the mapping from the source language variable names v to v_m, we have:*

$$\mathsf{inEnv}\ \rho\ E[\![e]\!] = N[\![e]\!]$$

To emphasize the modularity provided in our framework, we first prove the theorem for the functional sub-language, and then extend the result to the complete language.

4.2.1 Proof for the Functional Sub-language

We can establish the theorem for the functional sub-language by induction on the structure of programs composed out of variables, lambda abstractions, and function applications. The full proof is given in the Appendix. The basic technique is the same as Wand's, except that in addition to the basic rules of lambda calculus (e.g., β reduction), we also use monad laws and environment axioms.

The proof is possible because both the source language and meta language are lexically scoped. If the source language instead supported dynamically scoped functions:

$$E[\![\lambda v.e]\!] \quad = \quad \mathsf{return}(\lambda c.\mathsf{rdEnv\ then}\ \lambda\rho.\mathsf{inEnv}\ (\rho[c/v])\ E[\![e]\!])$$

where the caller-site environment is used within the function body, the theorem would fail to hold.

4.2.2 Extension to the Complete Language

Consider another source language construct "callcc". Since in proving the theorem we only used the axioms of environment monads, none of the cases already analyzed need to be proved again. We only have to verify that:

- the monad supporting continuations is still an environment monad, and
- the induction hypothesis holds for "callcc".

The former is stated in Proposition 3.1, and can be proved once and for all as we come up with monad transformers. The latter can be easily proved: "callcc" does not explicitly deal with the environment, and has exactly the same natural semantics as the standard semantics. In addition, it is a trivial computation (see the definition in the last section). Thus the induction hypothesis holds following the unit axiom of environment monads.

Similarly we can extend the theorem to cover other features such as ":=".

4.3 Benefits of Reasoning in Monadic Style

4.3.1 Modular Proofs

In denotational semantics, adding a feature may change the structure of the entire semantics, forcing us to redo the induction for every case of abstract syntax. For example, Wand [35] pointed out that he could change the semantics into continuation-based, and prove the theorem, but only by modifying the proofs accordingly.

Modular monadic semantics, on the other hand, offers highly modularized proofs and more general results. This is particularly applicable to real programming languages, which usually carry a large set of features and undergo constantly evolving designs.

[2] This means that in natural semantics, we are still implicitly passing around an environment, even though it is never used. Thus the theorem as stated does not strictly correspond to Wand's result [35]. Fortunately, the *naturality of liftings* (see our earlier work [18] for details) guarantees that adding and removing a feature does not affect computations which do not use that particular feature. Therefore the theorem still holds if we remove the explicit environment support from the underlying monad in natural semantics. (The next section addresses this in more detail.)

4.3.2 Axiomatization of Programming Language Features

Denotational semantics captures a computation as a piece of syntax tree coupled with an environment, a store etc. On the other hand, we view computations as abstract entities with a set of equations. Therefore, like *Semantic Algebras* [23] in action semantics, monads provide an *axiomatic view* of denotational semantics.

The environment axioms provide an answer to the question: "what constitutes an environment?" We are investigating useful axioms for other programming language features, such as exceptions and continuations.

5 Targeting Monadic Code

In general, it is more efficient to use target language built-in features instead of monadic combinators defined as higher-order functions. We have seen how the explicit environment can be "absorbed" into the meta-language. This section addresses the question of whether we can do the same for other features, such as the store and continuation.

We can view a target language as having a built-in monad supporting a set of primitive monadic combinators. For example, the following table lists the correspondence between certain ML constructs and primitive monadic combinators:

primitive monadic operators	ML construct
return x	x
c_1 then $\lambda x.c_2$	**let val** $x = c_1$ **in** c_2 **end**
update*	**ref**, **!**, **:=**
callcc	callcc
err	**raise Err**
...	

* ML reference cells support single-threaded states only.

It is easy to verify that the monad laws are satisfied in the above context. For example, the ML **let** construct is associative (assuming no unwanted name capturings occur):

let val $v_2 =$ **let val** $v_1 = c_1$ **in** c_2 **end** **in** c_3 **end**	=	**let val** $v_1 = c_1$ **in let val** $v_2 = c_2$ **in** c_3 **end** **end**

Recall (in Section 2) that the *Compute* monad is constructed as:

type *Compute* a = *EnvT Env* (*ContT Answer* (*StateT Store* (*ErrorT Id*))) a

Now we substitute the base monad *Id* with the built-in ML monad (call it M_{ML}):

type *Compute*$'$ a = *EnvT Env* (*ContT Answer* (*StateT Store* (*ErrorT* M_{ML}))) a

Note that *Compute*$'$ supports two sets of continuation, state and error handling functions. The monadic code can choose to use the ML built-in ones instead of those implemented as higher-order functions. In addition, all liftings we construct satisfy an important property (called the Naturality of Liftings [21] [18]): adding or deleting a monad transformer does not change the result of programs which do not use its operations. Since none of the monad transformers in *Compute*$'$ is used any more, it suffices to run the target program on *Compute*$''$:

type $Compute'' \ a = M_{ML} \ a$

which directly utilizes the more efficient ML built-in features.

The above transformation is possible because ML has a strictly richer set of features than our source language. If the source language requires a non-updatable version of state (for example, for the purpose of debugging), the corresponding state monad transformer will remain, and ensure the state is threaded correctly through all computations. If we instead target our source language to C, both the environment and continuation transformers have to be kept.

Therefore by using a monad with a set of primitive monadic combinators, we can expose the features embedded in the target language. It then becomes clear what is directly supported in the target language, and what needs to be compiled explicitly.

The above process seems trivial, but would have been impossible had we been working with traditional denotational semantics. Various features clutter up and make it hard to determine whether it is safe to remove certain interpretation overhead, and how to achieve that.

Earlier work [16] [8] [18] has shown that the order of monad transformers (in particular, some cases involving *ContT*) has an impact on the resulting semantics. In practice, we need to make sure when we discard monad transformers, that the resulting change of ordering does not have unwanted effects on semantics.

6 Related work

Early efforts [22] [27] [34] in semantics-directed compiler generation were based on traditional denotational semantics.

Mosses's *Action Semantics* [24] allows modular specification of programming language semantics, from which efficient compilers can be generated. In action semantics, source language constructs are translated into *actions*, built from a set of primitives and combinators. While action semantics is easy to construct, extend, understand and implement, we note the following comments made by Mosses ([24], page 5):

> Although the foundations of action semantics are firm enough, the *theory* for reasoning about actions (and hence about programs) is still rather weak, and needs further development. This situation is in marked contrast to that of denotational semantics, where the theory is strong, but severe pragmatic difficulties hinder its application to realistic programming languages.

Action semantics provided much of the inspiration for our work, which essentially attempts to formulate actions in a denotational semantics framework. Monad transformers roughly correspond to *facets* in action semantics, although issues such as concurrency are beyond the power of our approach.

Action semantics [4] [26] [25] and a related approach by Lee [17] have been successfully used to generate efficient compilers.

Moggi [21] first used monads and monad transformers to structure denotational semantics. Wadler [33] [32] popularized Moggi's ideas in the functional programming community by using monads to structure functional programs, in particular, interpreters. Our work on monad-based modular interpreters [18] follows a series of earlier attempts by Steele [30], Jones and Duponcheel [12], and Espinosa [8].

Moggi [21] raised the issue of reasoning in the monadic framework. Wadler [33] stated the laws satisfied by state monads. Hudak [10] suggested a more general framework — mutable abstract data types (MADTs) — to reason about states.

Meijer [19] combined the standard initial algebra semantics approach with aspects of Action Semantics to derive compilers from denotational semantics. An interesting area of future research is to combine the nice algebraic properties in Meijer's framework with the modularity offered in ours.

One application of partial evaluation [13] is to automatically generate compilers from interpreters [36] [2] [6] [20]. A partial evaluator has been successfully applied to an action interpreter [3], and similar results can be achieved with monadic interpreters [7].

Staging transformations, first proposed by Jørring and Scherlis [14], are a class of general program transformation techniques for separating a given computation into stages. Monadic transformers make computational stages somewhat more explicit by separating compile-time features, such as the environment, from run-time features.

Several researchers, including Kelsey and Hudak [15], Appel and Jim [1], and others, have built efficient compilers for higher-order languages by transforming the source language into continuation-passing style (CPS). The suitability of a monadic form as an intermediate form has been observed by many researchers (including, for example, Sabry and Felleisen [29] and Hatcliff and Danvy [9]). We will continue to explore along this direction in order to generate machine-level code from a monadic intermediate form.

7 Conclusions

We have shown that the monadic framework provides good support for high-level extensible specifications, program transformations, and reasoning about computations. Monadic-style proofs are better structured and easier to extend. The modular monadic semantics allows us to have an axiomatized formulation of well-known programming languages features such as environments. Our approach also facilitates generating code for various target languages and constructing program analysis tools.

Overall, we believe that modular monadic semantics is particularly suitable for compiler construction.

Acknowledgement

We would like to thank Rajiv Mirani, Zhong Shao, Satish Pai, Dan Rabin and the PEPM'95 anonymous referees for helpful comments on an earlier version of the paper, and Ross Paterson, Tim Sheard and John Launchbury for ideas and discussions.

References

[1] Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In *ACM Symposium on Principles of Programming Languages*, pages 193–302, January 1989.

[2] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[3] Anders Bondorf and Jens Palsberg. Compiling actions by partial evaluation. In *FPCA '93: Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, pages 308–317, New York, June 1993. ACM Press.

[4] Deryck F. Brown, Hermano Moura, and David A. Watt. ACTRESS: An action semantics directed compiler generator. In *Proceedings of the 4th International Conference on Compiler Construction, Edinburgh, U.K.*, pages 95–109. Springer-Verlag, 1992. Lecture Notes in Computer Science 641.

[5] D. Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ML. In *Proceedings of the 1986 ACM Symposium on Lisp and Functional Programming*, pages 13–27, 1986.

[6] Charles Consel. A tour of Schism. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM'93, Copenhagen, Denmark*, pages 145–154, New York, June 1993. ACM Press.

[7] Olivier Danvy, Jürgen Koslowski, and Karoline Malmkjær. Compiling monads. Technical Report CIS-92-3, Kansas State University, Manhattan, Kansas, December 1991.

[8] David Espinosa. Modular denotational semantics. Unpublished manuscript, December 1993.

[9] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon*, pages 458–471, New York, January 1994. ACM Press.

[10] Paul Hudak. Mutable abstract datatypes. Research Report YALEU/DCS/RR-914, Yale University Department of Computer Science, December 1992.

[11] Paul Hudak, Simon Peyton Jones, and Philip Wadler. Report on the programming language Haskell: a non-strict, purely functional language, version 1.2. Technical Report YALEU/DCS/RR-777, Yale University Department of Computer Science, March 1992. Also in ACM SIGPLAN Notices, Vol. 27(5), May 1992.

[12] Mark P. Jones and Luc Duponcheel. Composing monads. Research Report YALEU/DCS/RR-1004, Yale University Department of Computer Science, New Haven, Connecticut, December 1993.

[13] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2:9–50, 1989.

[14] Ulrik Jørring and William Scherlis. Compilers and staging transformations. In *Proceedings Thirteenth ACM Symposium on Principles of Programming Languages, St. Petersburg, Florida*, pages 86–96, 1986.

[15] Richard Kelsey and Paul Hudak. Realistic compilation by program transformation. In *ACM Symposium on Principles of Programming Languages*, pages 181–192, January 1989.

[16] David J. King and Philip Wadler. Combining monads. In John Launchbury and Patrick Sansom, editors, *Functional Programming, Glasgow 1992*, pages 134–143, New York, 1993. Springer-Verlag.

[17] Peter Lee. *Realistic Compiler Generation*. Foundations of Computing. MIT Press, 1989.

[18] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California*, New York, January 1995. ACM Press.

[19] Erik Meijer. More advice on proving a compiler correct: Improving a correct compiler. Submitted to Journal of Functional Programming.

[20] Torben Æ. Mogensen. Self-applicable partial evaluation for pure lambda calculus. In *PEPM'92: ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 116–121, June 1992. Yale University Department of Computer Science Research Report YALEU/DCS/RR-909.

[21] Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, Scotland, 1990.

[22] Peter D. Mosses. SIS—semantics implementation system, reference manual and user's guide. Technical Report DAIMI report MD-30, University of Aarhus, Denmark, 1979.

[23] Peter D. Mosses. A basic abstract semantic algebra. In Gilles Kahn, David B. MacQueen, and Gordon D. Plotkin, editors, *Semantics of Data Types: International Symposium, Sophia-Antipolis, France*, pages 87–107. Springer-Verlag, June 1984. Lecture Notes in Computer Science 173.

[24] Peter D. Mosses. *Action Semantics*, volume 26 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[25] P. Ørbæk. OASIS: An optimizing action-based compiler generator. In *Proceedings of the 5th International Conference on Compiler Construction, Edinburgh, U.K.*, pages 1–15. Springer-Verlag, April 1994. Lecture Notes in Computer Science 786.

[26] Jens Palsberg. A provably correct compiler generator. In B. Krieg-Brückner, editor, *ESOP '92: 4th European Symposium on Programming, Rennes, France, Proceedings*, pages 418–434, New York, February 1992. Springer-Verlag. Lecture Notes in Computer Science 582.

[27] Laurence C. Paulson. A semantics-directed compiler generator. In *Proceedings of the Ninth ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico*, pages 224–233, 1982.

[28] Simon Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings 20th Symposium on Principles of Programming Languages*, pages 71–84. ACM, January 1993.

[29] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, pages 288–298. ACM Press, June 1992. Lisp Pointers Volume V, Number 1.

[30] Guy L. Steele Jr. Building interpreters by composing monads. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon*, pages 472–492, New York, January 1994. ACM Press.

[31] Joseph Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[32] Philip Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico*, pages 1–14, January 1992.

[33] Philip L. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, 1990.

[34] Mitchell Wand. A semantic prototyping system. *SIGPLAN Notices, ACM Symposium on Compiler Construction*, 19(6):213–221, 1984.

[35] Mitchell Wand. A short proof of the lexical addressing algorithm. *Information Processing Letters*, 35:1–5, June 1990.

[36] Daniel Weise, Roland Conybeare, Erik Ruf, and Scott Seligman. Automatic online partial evaluation. In John Hughes, editor, *Proceedings Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA*, pages 165–191. Springer-Verlag, August 1991. Lecture Notes in Computer Science 523.

The proof of theorem 4.1

We need the following lemma:

$$\textsf{inEnv}\ \rho\ N[\![e]\!] = N[\![e]\!]$$

which follows from that fact that the Natural Semantics never refers to the environment in any way.

The main theorem is then proved by induction on the structure of programs:

Case v:

$$\begin{array}{lll}
\multicolumn{3}{l}{\textsf{inEnv}\ \rho\ E[\![v]\!]} \\
= & \textsf{inEnv}\ \rho\ (\rho v) & \\
= & \textsf{inEnv}\ \rho\ v_m & \\
= & v_m & \text{(lemma)}
\end{array}$$

Case $\lambda v.e$:

$$\begin{array}{lll}
\multicolumn{3}{l}{\textsf{inEnv}\ \rho\ (E[\![\lambda v.e]\!])} \\
= & \textsf{inEnv}\ \rho\ (\textsf{rdEnv then}\ \lambda\rho.\ \textsf{return}(\lambda c.\textsf{inEnv}\ (\rho[c/v])\ E[\![e]\!])) & \\
= & \textsf{inEnv}\ \rho\ \textsf{rdEnv then}\ \lambda\rho'.\textsf{inEnv}\ \rho\ (\textsf{return}(\lambda c.\textsf{inEnv}\ (\rho'[c/v])\ E[\![e]\!])) & \text{(dist.)} \\
= & \textsf{return}\ \rho\ \textsf{then}\ \lambda\rho'.\ \textsf{return}(\lambda c.\textsf{inEnv}\ (\rho'[c/v])\ E[\![e]\!]) & \text{(cancel., unit)} \\
= & \textsf{return}(\lambda c.\textsf{inEnv}\ (\rho[c/v])\ E[\![e]\!]) & \text{(left unit)} \\
= & \textsf{return}(\lambda v_m.\textsf{inEnv}\ (\rho[v_m/v])\ E[\![e]\!]) & (\alpha\ \text{renaming}) \\
= & \textsf{return}(\lambda v_m.N[\![e]\!]) & \text{(ind. hypo.)} \\
= & N[\![\lambda v.e]\!] &
\end{array}$$

Case $(e_1\ e_2)_n$:

$$\begin{array}{lll}
\multicolumn{3}{l}{\textsf{inEnv}\ \rho\ (E[\![(e_1\ e_2)_n]\!])} \\
= & \textsf{inEnv}\ \rho\ (E[\![e_1]\!]\ \textsf{then}\ \lambda f.\textsf{rdEnv then}\ \lambda\rho.f(\textsf{inEnv}\ \rho\ E[\![e_2]\!])) & \\
= & \textsf{inEnv}\ \rho\ (E[\![e_1]\!])\ \textsf{then}\ \lambda f.\textsf{inEnv}\ \rho\ \textsf{rdEnv then}\ \lambda\rho'.\textsf{inEnv}\ \rho\ (f(\textsf{inEnv}\ \rho'\ E[\![e_2]\!])) & \text{(dist.)} \\
= & N[\![e_1]\!]\ \textsf{then}\ \lambda f.\ \textsf{return}\ \rho\ \textsf{then}\ \lambda\rho'.\textsf{inEnv}\ \rho\ (f(\textsf{inEnv}\ \rho'\ E[\![e_2]\!])) & \text{(ind. hypo., cancel.)} \\
= & N[\![e_1]\!]\ \textsf{then}\ \lambda f.\textsf{inEnv}\ \rho\ (f(\textsf{inEnv}\ \rho\ E[\![e_2]\!])) & \text{(left unit)} \\
= & N[\![e_1]\!]\ \textsf{then}\ \lambda f.\textsf{inEnv}\ \rho\ (f(N[\![e_2]\!])) & \text{(ind. hypo.)} \\
= & \textsf{inEnv}\ \rho\ (N[\![e_1]\!])\ \textsf{then}\ \lambda f.\textsf{inEnv}\ \rho\ (f(N[\![e_2]\!])) & \text{(lemma)} \\
= & \textsf{inEnv}\ \rho\ (N[\![e_1]\!]\ \textsf{then}\ \lambda f.f(N[\![e_2]\!]))\text{(dist.)} & \\
= & \textsf{inEnv}\ \rho\ (N[\![(e_1\ e_2)_n]\!]) & \\
= & N[\![(e_1\ e_2)_n]\!] & \text{(lemma)}
\end{array}$$

Case $(e_1\ e_2)_v$:

$$\begin{array}{lll}
\multicolumn{3}{l}{\textsf{inEnv}\ \rho\ (E[\![(e_1\ e_2)_v]\!])} \\
= & \textsf{inEnv}\ \rho\ (E[\![e_1]\!]\ \textsf{then}\ \lambda f.E[\![e_2]\!]\ \textsf{then}\ \lambda v.f(\textsf{return}\ v)) & \\
= & \textsf{inEnv}\ \rho\ (E[\![e_1]\!])\ \textsf{then}\ \lambda f.\textsf{inEnv}\ \rho\ E[\![e_2]\!]\ \textsf{then}\ \lambda v.\textsf{inEnv}\ \rho\ (f(\textsf{return}\ v)) & \text{(dist.)} \\
= & N[\![e_1]\!]\ \textsf{then}\ \lambda f.N[\![e_2]\!]\ \textsf{then}\ \lambda v.\textsf{inEnv}\ \rho\ (f(\textsf{return}\ v)) & \text{(ind. hypo.)} \\
= & \textsf{inEnv}\ \rho\ (N[\![e_1]\!]\ \textsf{then}\ \lambda f.N[\![e_2]\!]\ \textsf{then}\ \lambda v.f(\textsf{return}\ v)) & \text{(lemma, dist.)} \\
= & \textsf{inEnv}\ \rho\ (N[\![(e_1\ e_2)_v]\!]) & \\
= & N[\![(e_1\ e_2)_v]\!] & \text{(lemma)}
\end{array}$$