WPI-CS-TR-03-04

February 2003
emended, 1 March 2003

Decomposing lambda — the Kernel programming language

by

John N. Shutt

Computer Science Technical Report Series

WORCESTER POLYTECHNIC INSTITUTE

Computer Science Department
100 Institute Road, Worcester, Massachusetts 01609-2280

Decomposing lambda — the Kernel programming language

John N. Shutt
jshutt@cs.wpi.edu
http://www.cs.wpi.edu/~jshutt/
Computer Science Department
Worcester Polytechnic Institute
Worcester, MA 01609

February 2003

emended
March 1, 2003

Abstract

The Kernel programming language attempts to expand on the simplicity and versatility of Scheme by exploiting a decomposition of Scheme's `lambda` constructor into two orthogonal constructors. The primary constructor, called `$vau`, builds compound combiners that act directly on their unevaluated operands ("call-by-text", or *operative*, compound combiners); while a second constructor, called simply `wrap`, induces evaluation of arguments. This report describes how these constructors work, and explores some of their consequences, including some design differences of Kernel from Scheme that help the programmer to manage the inherent volatility of first-class operative combiners.

This report is an expanded form of material presented to the NEPLS 7 conference, held at WPI in October 2002.

Contents

1 Introduction **1**

2 Background **1**
2.1 Terminology 1
2.2 History of combiner constructors 3

3 The decomposition **5**
3.1 Applicatives . . . 6
3.2 Operatives . . . 7
3.3 `$lambda` . . . 8
3.3.1 An extended example . . . 9
3.4 `apply` . . . 13

4 Well-behavedness **15**
4.1 Hygiene . . . 15
4.1.1 Variable capturing . . . 15
4.1.2 Context capturing . . . 17
4.2 Stability . . . 19
4.2.1 Isolating environments . . . 20
4.2.2 Restricting environment mutation . . . 22

5 Conclusions and future work **25**

A Kernel meta-circular evaluator (in Scheme) **28**
A.1 Evaluator central logic . . . 28
A.2 Interpreter top level . . . 29
A.3 Encapsulated Kernel data types . . . 31
A.4 Kernel standard environment . . . 35
A.5 Termination . . . 37

B References **38**

List of Equivalences

1 *wrap* . . . 6
2 *unwrap* . . . 6
3 *$lambda* . . . 9
4 *apply* (fixpoint) . . . 14
5 *apply* (general case) . . . 14
6 *apply* (default environment) . . . 14

1 Introduction

The Kernel programming language attempts to expand on the simplicity and versatility of Scheme by exploiting a decomposition of Scheme's `lambda` constructor into two orthogonal constructors. The primary constructor, called `$vau`, builds compound combiners that act directly on their unevaluated operands ("call-by-text", or *operative*, compound combiners); while a second constructor, called simply `wrap`, induces evaluation of arguments. This report describes how these constructors work, and explores some of their consequences, including some design differences of Kernel from Scheme that help the programmer to manage the inherent volatility of first-class operative combiners.

This report is an expanded form of material presented to the NEPLS 7 conference, held at WPI in October 2002. [NEPLS]

§2 defines basic terminology and reviews some Lisp history. §3 explains the new Kernel primitives, and shows how they can be used to build the primitives that they replace. §4 discusses well-behavedness in Kernel versus Scheme. §5 assesses the state of the Kernel project and its prospects for the future. Appendix A gives a meta-circular evaluator for Kernel, written in Scheme.

2 Background

2.1 Terminology

This subsection establishes terminology for discussing various entities involved in the evaluation process.

In Scheme (and Kernel), as in most Lisps, computation consists of evaluating expressions, and expression evaluation has, more or less,[1] just three cases: Most expressions self-evaluate, i.e., they evaluate to themselves; expressions of type *symbol* are looked up in the current environment; and expressions of type *pair* are the only case with internal structure, and the case with which we will be primarily concerned here.

A pair to be evaluated is a *combination*; its unevaluated car is an *operator*; and in the usual case that the combination is a list, any further elements of the list after the operator are *operands*. In the usual case that all the operands are evaluated, and all other actions use the results rather than the operands themselves, the results of

[1]The formal definition of Scheme in the R5RS, [KeClRe98, §7.2], is not quite as uniform as what is described here; it focuses on legal syntax, and consequently excludes certain cases (such as improper lists) from the set of evaluable expressions. Kernel extends evaluation ruthlessly to all objects, counting evaluation among the rights and privileges of first-class objects. The types of expressions excluded by Scheme, and therefore rejected by Scheme prior to evaluation, are either detected by Kernel during evaluation (which may be anticipated in some decidable cases); or in some cases, noted below in §3.4, expression types rejected by Scheme are allowed by Kernel.

evaluating the operands are *arguments*. This much terminology may be considered standard for Scheme, as it is taken from the Wizard Book, [AbSu96, §1.1.1].

There is no general Scheme term for the result of evaluating an operator, for the straightforward reason that in Scheme, not all operators are evaluated; special-form operators are not. In Kernel, where all operators are evaluated, the result of evaluating an operator is (if type-correct) a *combiner*; and this term extends naturally to the context of Scheme, or any other Lisp where special-form operators are not evaluated, by saying in the general case that a combiner is the action *designated* by an operator.

This report will use `monospace` lettering for symbols, and ***`italicized monospace`*** lettering for combiners. Thus, for example, ***`apply`*** is the combiner named (in whatever Lisp context we're considering, most often the standard environment of Kernel) by symbol `apply`; ***`cond`*** is the (second-class) combiner designated by symbol `cond`;[2] and so on.

Kernel also needs to distinguish between combiners that act on their operands (as do special-form combiners), and combiners that act on their arguments (as do Scheme procedures[3]). Combiners that act on their operands are sometimes said to use *call-by-text*; however, this term is not used in the Kernel project because —besides the word "text" conjuring visions of ASCII code— there is no similarly formed adjective for an arbitrary combiner that acts on its arguments. Adjectives of the form *call-by-X* are used to specify *when* the operands are evaluated to arguments, especially whether they are evaluated eagerly (call-by-value) or lazily (call-by-name) [CrFe91][4]; but the eager/lazy distinction is orthogonal to most of our discussion here, which concerns only whether the operands are evaluated at all, so *call-by-X* will not serve our purpose.

Instead, the Kernel project calls a combiner that acts directly on its operands an *operative combiner* (or, simply an *operative*); and a combiner that acts on its arguments an *applicative combiner* (an *applicative*).[5]

The principal terms presented in this subsection are summarized by Table 1.

[2]The corresponding (first-class) Kernel standard combiner is ***`$cond`***, i.e., its standard name is `$cond`.

[3]The use of the term *procedure* in Scheme does not coexist comfortably with the common use of the term *procedure* in the (non-Scheme) literature, where it means what we call a combiner. For the Kernel project it was found that using *procedure* in either sense tended to promote confusion; hence Kernel avoids the term *procedure* entirely.

[4]Following [CrFe91], *call-by-X* specifies when the operands are evaluated (*evaluation strategy*), while *pass-by-X* specifies how the operands/arguments are bound to the parameters (*binding strategy*, usually either pass-by-worth or pass-by-reference).

[5]The term *applicative* could be taken to refer to the Scheme term *application* (a combination that isn't a special form), though that term is not used in Kernel. By original intent, *applicative* refers to the type required of the first argument to Kernel's ***`apply`*** combiner; see §3.4. Some looseness was admitted in the symmetry between terms *operative* and *applicative*, because the more stringently symmetrical term *argumentative* proved difficult to use with a straight face.

combination: pair to be evaluated

parts of a combination

	car	cdr
unevaluated	operator	operands
evaluated	combiner	arguments

types of combiner

acts on	type
operands	operative
arguments	applicative

Table 1: Terminology

2.2 History of combiner constructors

In most Lisp languages, the primary constructor of combiners is the ***lambda*** operative. ***lambda*** constructs applicatives, and is generally a *universal* constructor of applicatives, in the sense that it can implement all applicatives that are possible within the computational model of whatever Lisp dialect it occurs in. It is therefore unnecessary to have any other combiner constructors in the language, provided one is willing to accept that all constructed combiners will be applicative. ***lambda*** was the only combiner constructor in the original description of Lisp [McC60], and it is —or rather was, until quite recently [KeClRe98]— the only combiner constructor in Scheme. It is of particular interest here that Scheme had only one call mechanism for compound combiners, the one being used by compound applicatives constructed via ***lambda***, as this singularity appears to be an important element of the elegance for which the Scheme design is noted: A language with multiple compound call mechanisms must answer awkward questions about how they interact (a problem that will appear in various guises in this report), but there can be no interactions between mechanisms if there is only one mechanism.

In practice, though, it's convenient to be able to construct new operatives. So, in the 1960s and 70s, Lisps developed two rival strategies for constructing compound operatives [Pi80]: *macros* and *fexprs*.

Lisp macros are a generalization of the technique commonly used in assembly languages.[6] A macro takes in its operands —as abstract syntax— performs some arbitrary computation on them, and returns a new expression that is then subject to evaluation. Note that this technique forces an explicitly two-phase evaluation process.

Compound operatives under the other strategy have gone by a variety of names. MacLisp called them FEXPRs (the name used here, and commonly found elsewhere in

[6] The sixties and into the early seventies were something of a heyday for macro facilities, and macros in higher-level languages were not too remarkable. Remember, this is the era when C and TeX were created. The apotheosis of object-orientation had not yet occurred; its later social niche among programming language visionaries was being filled by *extensible languages* [Sta75], and macros were a commonly used extension technique. Another especially general macro facility, outside the Lisp world, belonged to PL/I [Bar79].

modern literature, e.g. [Wa98]); Interlisp, the other major Lisp camp in the seventies,[7] called them NLAMBDAs; and some experimental languages in the eighties called them *reflective*, or *reifying*, procedures (e.g., [WaFr86]).

The idea behind fexprs is that the programmer defines a compound combiner using the same form as with ***lambda***, except that (1) the usual formal parameters will be bound to the unevaluated operands, rather than to the arguments, and (2) there is an additional formal parameter that will be bound to the dynamic environment from which the call is made.[8] The fexpr thus has access to all the same information that must be provided, in general, to evaluate a call to a special-form combiner; so that, in effect, a fexpr *is* a programmer-defined special-form combiner. Anything that could be done using special-forms (therefore, anything that could be done using macros) can evidently be done using fexprs.

Macros in the 1970s had a few behavioral problems arising from name-space collisions (called *variable capturing*, which will be discussed in §4.1.1); but programmers had been living with those problems since before the advent of higher-level languages, and in any case the problems would be eliminated by hygienic macro devices in the late eighties [ClRe91]. Fexprs, however, were not merely badly behaved in themselves; adding them to a Lisp language undermined the well-behavedness of the rest of the language. The case against fexprs was carefully and clearly laid out in [Pi80], which recommended omitting fexprs from future dialects of Lisp:

> it has become clear that such programming constructs as `NLAMBDA`'s and `FEXPR`'s are undesirable for reasons which extend beyond mere questions of aesthetics, for which they are forever under attack.

In other words, fexprs are (were) badly behaved and ugly — this at a time, remember, when mainstream Lisps were dynamically scoped, which in retrospect seems to add a certain extra sting to the aesthetic criticism. The Lisp community mostly followed his recommendation (and, incidentally, accumulated quite a lot of citations of his paper), although, as noted, fexprs occurred under other names in the reflective Lisps of the 1980s; and the misbehavior of fexprs has continued to attract a modicum of attention, both in regard to the feature itself [Wa98] and as a paradigmatic example of undesirable behavior [Mi93].

However, fexprs also have a potential advantage that macros *cannot* match. It was suggested earlier that Scheme derived its elegance partly from the fact that it had only one compound call mechanism. The macro strategy precludes this form of

[7] On the overall shape of the Lisp community over the decades, see the Lisp paper in HOPL II, [StGa93].

[8] Even in the age of dynamically scoped Lisps, it was necessary to explicitly pass in the dynamic environment. MacLisp did support "one-argument FEXPRs", whose one parameter was bound to the entire list of operands, but the name of the one parameter could be captured if it occurred in an operand; so MacLisp also supported "two-argument FEXPRs", whose second parameter was bound to the unextended dynamic environment. [Pi83]

elegance: It *requires* two compound call mechanisms, one in each of the two phases of evaluation, and the call mechanisms are inherently distinct from each other by the very fact that they *are* in explicitly separate phases of evaluation. The fexpr strategy, however, does not require explicitly multi-phase evaluation, and so does not inherently preclude a single compound call mechanism; and this is the window of opportunity through which Kernel attempts to climb.

3 The decomposition

Evaluation of a call to an applicative involves two logically separable parts: evaluation of the operands, and action dependent on the resulting arguments. This is not a language-specific statement; it is a paraphrase of the definition of the term *applicative*. If the combiner call is dependent on the operands in any way other than through the values of the arguments, the combiner isn't applicative. Note that the two parts of the call evaluation are only required to be separable, not necessarily consecutive: Operand evaluation for an applicative can, in general, be lazy or eager, as long as it is the only activity that depends on the operands directly.

The central idea in Kernel is that, by using two orthogonal primitive constructors to support the two parts of an applicative call, one can reconcile fexprs with the elegance of a single compound call mechanism. The second part of an applicative call —action on the arguments— is allowed to range over all combiners, and is therefore thoroughly supported by Kernel's fexpr-style primitive constructor, ***$vau***. Applicatives can then be constructed near-trivially from operatives, avoiding both the inelegance of two highly non-orthogonal primitive constructors (***$vau*** and ***$lambda***,[9] the latter being non-primitive in Kernel), and the conceptual difficulties attendant on defining operatives as a variant of applicatives (cf. §5).

Breaking the classical ***lambda*** constructor into two parts is a deep change to the Lisp computation model, and cannot be accomplished —in an elegant way, which is the point of the exercise— as a small localized amendment to the language (in this case, Scheme). ***lambda*** is almost the entire core of the language; the only other operatives in the minimal semantics of Scheme ([KeClRe98, §7.2]) are ***if*** and ***set!*** — and each of the others handles just one task, whereas ***lambda*** covers *everything else*.[10] In order to make the Kernel design work out cleanly —and especially, to allow the programmer to manage the inherent volatility of first-class operatives— it was sometimes necessary to make sweeping global changes to the Scheme design; so that, while Kernel *is* unmistakably a variant of Scheme, it is nowhere near achieving Scheme source compatibility.

[9]The Kernel name for the classical constructor is `$lambda`. When we speak of ***lambda***, we are referring to the constructor in Lisps generally, or in Scheme particularly.

[10]A quick list of roles covered by ***lambda*** would include variable binding, compound control construction, recursion, and encapsulation.

3.1 Applicatives

The primitive constructor for Kernel's *applicative* data type is ***wrap***; it simply takes any combiner c, and returns an applicative that evaluates its operands and passes the list of results on to c. More precisely, the equivalence

$$\begin{array}{rl} & \texttt{((}\textit{wrap}\ x_0\texttt{)}\ x_1\ x_2 \ldots x_n\texttt{)} \\ = & \texttt{(}\textit{eval}\ \texttt{(}\textit{list}\ x_0\ x_1\ x_2 \ldots x_n\texttt{)}\ \texttt{(}\textit{get-current-environment}\texttt{))} \end{array} \quad (1)$$

must hold for all x_k, provided x_0 evaluates to a combiner.[11] That is, to evaluate a combination with an applicative, build a combination that passes the *arguments* to the underlying combiner, and evaluate the new combination in the current environment. (This equivalence constitutes, in itself, something like half the nontrivial logic in the heart of the Kernel meta-circular evaluator in §A.1.)

Equivalence 1 is not between "syntactic" expressions in the usual sense. Because operators ***wrap***, ***eval***, ***list***, and ***get-current-environment*** are italicized, they specify the standard combiners of those names, of type *applicative* rather than type *symbol*. Had the operators been unitalicized, the two expressions would only be equivalent if evaluated in an environment where symbols `wrap`, `eval`, etc. are bound to the respective standard combiners. As stated, evaluating the two expressions in *any* environment should produce the same results, since the combiners will evaluate to themselves regardless of what their standard symbolic names might be bound to.

We can now be more specific about our claim, in §2.1, that the operative/applicative distinction is orthogonal to lazy/eager operand evaluation. Equivalence 1 would be unchanged if applicatives constructed with ***wrap*** used lazy rather than eager operand evaluation, provided that standard applicative ***list*** used lazy operand evaluation too; later equivalences in this report will be similarly invariant under choice of lazy/eager policy. (Kernel follows Scheme in practicing eager operand evaluation.)

It may be noted, in passing, that Equivalence 1 does not require x_0 to evaluate to an operative. It is entirely possible in Kernel to wrap an operative, say, twice, in which case the equivalence dictates that the resulting combiner will evaluate its operands, then evaluate the results of those first evaluations, and pass the results of the second evaluations to the operative.

To make operand evaluation fully separable from the rest of an applicative call, Kernel provides an accessor ***unwrap***, that extracts the underlying combiner of a given applicative. This behavior is expressed by the equivalence:

$$\texttt{(}\textit{unwrap}\ \texttt{(}\textit{wrap}\ x\texttt{))} \;=\; x \quad (2)$$

Equivalence 2 holds for all x, provided x evaluates to a combiner.[12]

[11] The restriction arises because in the second expression, the result of evaluating x_0, call it x'_0, is itself evaluated when it occurs as the operator of the combination constructed by ***list***. If x'_0 is a combiner, then it self-evaluates, so the redundant evaluation doesn't spoil the equivalence.

[12] This time, the restriction comes from error-checking: If x evaluates without error to an object x' of some non-combiner type, a dynamic type error will occur when x' is passed to ***wrap***.

3.2 Operatives

The general form of a ***$vau*** expression is:

(***$vau*** *ptree* *eparm* x_1 x_2 ... x_n)

As noted (of fexprs generally) in §2.2, this works almost the same way as classical ***lambda***:

- *ptree* is the *formal parameter tree*, a slight generalization of the formal parameter list used in Scheme.

- When the compound operative is called, the formal parameters in the tree will be bound to the *operands* of the call, rather than to the arguments.

- *eparm* is an additional formal parameter, called the *environment parameter*, that will be bound to the *dynamic environment*, i.e., the environment from which the compound operative is called.

In all other respects (notably static scoping), ***$vau*** works as does Scheme ***lambda***.

Here are some simple examples; the behavior of ***$vau*** will be shown in detail for a more sophisticated example in §3.3.1.

```
($define! $quote
   ($vau (x) #ignore x))
```

When this compound operative is called, a local environment is created whose parent environment is the *static environment* in which the vau expression was evaluated; then the local environment is extended by matching the formal parameter tree, `(x)`, against the *operand list* of the call. There must be exactly one operand, which is locally bound by symbol `x`. Kernel has a special atomic object `#ignore` that can be used in place of a formal parameter name; it matches anything and makes no local binding.[13] In this case, the dynamic environment —from which the call is made— is not given a local name. There is one expression in the body of the compound operative, `x`, which is evaluated in the local environment, where it has been bound to the operand of the call, and so the unevaluated operand of the call is returned as the result of the call — just the behavior one expects of the quotation combiner.[14]

[13]The use of `#ignore` in the formal parameter tree is a matter of uniformity and occasional convenience; but its use in place of the environment parameter is needed to preserve proper tail recursion. If *every* local environment contained a dynamic reference, it would be impossible for the language interpreter to garbage-collect local environments during a supposedly iterative process.

[14]However, the current draft of the Kernel design, [Sh0x], specifically omits operative ***$quote***. Operative ***$quote*** will be considered several times in relation to various undesirable behaviors in §4.1.2.

The leading dollar-signs (`$`) on operators `$define!`, `$quote`, and `$vau` are a naming convention. With two potentially infinite domains of constructed combiners existing side by side in the language, and the question of operand evaluation riding on the distinction between them, it's important to know which combiners are of which type; so Kernel observes a universal convention that operative names begin with `$`. There is (following Scheme design philosophy) no enforcement of the convention, just as predicates aren't *required* to end with `?`[15]; but, by articulating the programmer's intent, the convention makes programs much easier to manage.

(The character `$` was originally meant to be a stylized "S", mnemonic for "Special", as in "special form". Since Greek letter *vau* is the ancestor of Roman letter `F`, the name of Kernel's "special form constructor", so to speak, could be read as "SF".)

```
($define! get-current-environment
   (wrap ($vau () e e)))
```

In this definition, the wrap expression constructs an applicative from an underlying compound operative. The operative takes zero operands —that is, the cdr of the calling combination must be nil— so the applicative takes zero arguments.[16] The first instance of `e` is the environment parameter, which is locally bound to the dynamic environment from which the call was made. The second instance of `e` is evaluated in the local environment, so that the dynamic environment is returned as the result of the call. Standard applicative ***get-current-environment*** was used in Equivalence 1, in §3.1.

```
($define! list
   (wrap ($vau x #ignore x))
```

Here, the underlying compound operative locally binds `x` to the entire list of operands to the operative — which are themselves the results of evaluating the operands to the enclosing applicative. So the overall effect of the constructed applicative is to evaluate its operands, and return a list of the resulting arguments.

3.3 $lambda

In the final ***list*** example of §3.2, above, Scheme programmers may have noted that standard applicative ***list*** is much easier to construct in Scheme, using ***lambda***:

[15]Kernel is, incidentally, generally more uniform than Scheme about its naming conventions, appending `?`'s to its numeric ordering predicates (`<? <=? =? >? >=?`), and `!` to the name of its primitive mutator of environments (`$define!`).

[16]One might ask, if there are no operands to be evaluated or not evaluated, why we bother to wrap ***get-current-environment***. We prefer to make our combiners applicative unless there's a specific need to make them operative. It's a matter of not crying wolf: Anyone reading a program should know to pay special attention when he encounters an explicit operative. Operative definitions will stand out more once we introduce ***$lambda*** in the next subsection (§3.3).

Instead of Kernel

```
(wrap ($vau x #ignore x))
```

one could simply write Scheme

```
(lambda x x)
```

Naturally, Kernel does have a standard operative *$lambda*; it simply isn't primitive.

The behavior we want from *$lambda* is expressed by equivalence

$$\begin{aligned} & (\mathit{\$lambda}\ \mathit{ptree}\ x_1\ x_2\ \ldots\ x_n) \\ = \ & (\mathit{wrap}\ (\mathit{\$vau}\ \mathit{ptree}\ \texttt{\#ignore}\ x_1\ x_2\ \ldots\ x_n)) \end{aligned} \tag{3}$$

In other words, a lambda expression is just a wrapped vau expression that ignores its dynamic environment. The equivalence can be converted straightforwardly into an implementation of *$lambda* as a compound operative:

```
($define! $lambda
   ($vau (ptree . body) static-env
      (wrap (eval (list* $vau ptree #ignore body)
                  static-env))))
```

We describe how this compound operative *$lambda* works, first in brief, and then in detail through an extended example.

When this *$lambda* is called, the first operand in the call is locally bound by symbol `ptree`, and the list of remaining operands (the cddr of the combination) is locally bound by symbol `body`. The dynamic environment of the call to *$lambda* is locally bound by symbol `static-env`; that environment will become the static environment of the applicative constructed by *$lambda*. The body of *$lambda* is then evaluated in the local environment.

*list** is a standard applicative that works almost like *list*, returning a list of its arguments except that the last argument becomes the rest of the list rather than the last element of the list. In this case it constructs a combination whose operator is operative *$vau* (not symbol `$vau`), whose cadr is the intended parameter tree, whose caddr is `#ignore`, and whose cdddr is the intended body. The constructed combination is then evaluated in the intended environment *static-env*, so that *$vau* makes *static-env* the static environment of the compound operative that it constructs. That compound operative is then wrapped, and the resulting applicative is returned as the result of the call to *$lambda*.

3.3.1 An extended example

To show how this works in detail, we trace through an extended example — defining *$lambda*, using *$lambda* to define an applicative *square*, and using *square* to square a number. While elsewhere in this report we have used, and will use, a semi-formal equational approach to describe language semantics, here we use a semi-formal reduction approach to describe language processing.

We use square-bracketed lists[17] both to denote computational states, where the first element of the list is a verb; and to denote records, where the first element is a tag. The two record types are:

[operative *ptree eparm body env*]
[applicative *combiner*]

where the first form denotes a compound operative with given parameter tree, environment parameter, body, and static environment; and the second form denotes an applicative with given underlying combiner. The only two verbs we'll need are "eval" and "combine"; our primitive rewrite rules for combinations are:

$$\begin{array}{ll} & [\text{eval } \texttt{(}\mathit{operator}\ \texttt{.}\ \mathit{operands}\texttt{)}\ e] \\ \Rightarrow & [\text{combine } [\text{eval } \mathit{operator}\ e]\ \mathit{operands}\ e] \end{array}$$

$$\begin{array}{ll} & [\text{combine } [\text{applicative } x_0]\ \texttt{(}x_1\ \ldots\ x_n\texttt{)}\ e] \\ \Rightarrow & [\text{combine } x_0\ \texttt{(}[\text{eval } x_1\ e] \ldots [\text{eval } x_n\ e]\texttt{)}\ e] \end{array}$$

$$\begin{array}{lll} & [\text{combine } [\text{operative } \mathit{ptree}\ \mathit{eparm}\ \texttt{(}x\texttt{)}\ e_1]\ \mathit{operands}\ e_2] & \\ \Rightarrow & [\text{eval } x\ e'_1] & \text{where } e'_1 \text{ extends } e_1 \text{ by matching} \\ & & \quad \mathit{ptree} \text{ against } \mathit{operands}\text{, and} \\ & & \quad \mathit{eparm} \text{ against } e_2 \end{array}$$

The third rule has been simplified for this example, by assuming that the body of a compound operative always contains just one expression.[18] For each standard operative `$foo`, there is also a rule for rewriting [combine `$foo` x e]; we omit these here, as they are self-evident where they occur in the reductions.

As a guide and supplement to the reductions, the entire arrangement of environments and (constructed) combiners through all three phases of the example is illustrated by Figure 1. An environment is depicted as a boxed set of bindings, a constructed operative is an oval, and a constructed applicative is a boxed reference to its underlying combiner. Unused bindings in each environment are omitted; in particular, the figure omits the bindings *created* by the definitions — those of `$lambda` in e_0 and `square` in e_1.

[17] Historically, square brackets have been used in the Lisp world to delimit *M-expressions*, a notion and notation that date all the way back to [McC60]. S-expressions (*S* for *Symbolic*) are passive data to be manipulated, while M-expressions (*M* for *Meta*) are active programs that do the manipulating.

[18] Without this simplification, we'd need a third verb, "sequence". We're glossing over issues of evaluation order, here; the reductions could just as well be in a pure subset of the language.

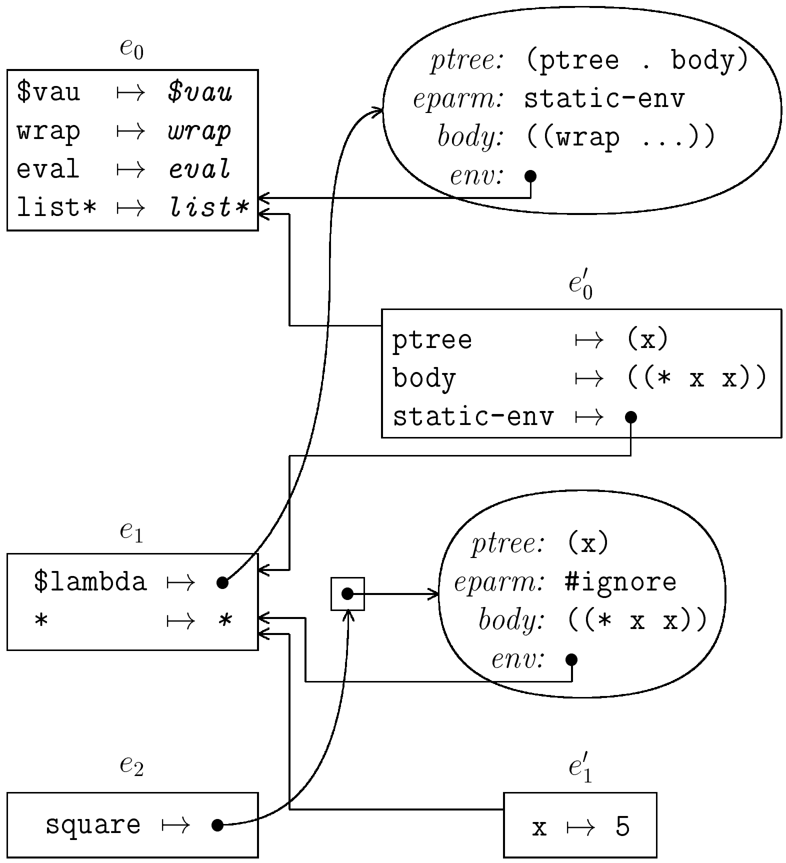

Figure 1: Objects of the extended example.

Suppose our compound definition for *`$lambda`* (which we repeat here),

```
($define! $lambda
   ($vau (ptree . body) static-env
      (wrap (eval (list* $vau ptree #ignore body)
                  static-env))))
```

is evaluated in an environment e_0, in which we will assume the default bindings for all the standard combiners used in the definition — *`$vau`*, *`wrap`*, *`eval`*, and *`list*`*.[19] We have:

  [eval `($vau (ptree . body) static-env ...)` e_0]
⇒ [combine [eval `$vau` e_0] `((ptree . body) static-env ...)` e_0]
⇒ [combine *`$vau`* `((ptree . body) static-env ...)` e_0]
⇒ [operative `(ptree . body) static-env (...)` e_0]

Now, suppose we evaluate another definition,

```
($define! square
   ($lambda (x) (* x x)))
```

in environment e_1. It doesn't matter to us whether e_1 is related to e_0 (they could even be the same environment), so long as symbol `$lambda` is bound in e_1 to the compound operative *`$lambda`* we just constructed, and symbol `*` to standard combiner *`*`*.

For any standard applicative *`foo`*, we'll name its underlying operative *`$foo`*; that is, *`foo`* = [applicative *`$foo`*].

  [eval `($lambda (x) (* x x))` e_1]
⇒ [combine [eval `$lambda` e_1] `((x) (* x x))` e_1]
⇒ [combine [operative `(ptree . body) static-env ((wrap ...))` e_0]
      `((x) (* x x))`
      e_1]
⇒ [eval `(wrap ...)` e_0']   where e_0' extends e_0 with bindings
      `ptree` $\mapsto$ `(x)`
      `body` $\mapsto$ `((* x x))`
      `static-env` $\mapsto$ e_1
⇒ [combine [eval `wrap` e_0'] `((eval (...) static-env))` e_0']
⇒ [combine *`wrap`* `((eval (...) static-env))` e_0']
⇒ [combine *`$wrap`* `(`[eval `(eval (...) static-env)` e_0']`)` e_0']

[19] We don't bother to assume a binding for *`$define!`* because, to simplify the example, we will only describe the evaluation of the body of each definition — that is, evaluation of the second operand passed to *`$define!`*. If we wanted to trace the actual call to *`$define!`*, we'd need another verb, "bind", that binds a given symbol to a given value in a given environment, so that

[combine *`$define!`* (*symbol body*) e] ⇒ [bind *symbol* [eval *body* e] e]

From this point, it would be needlessly cumbersome to carry along the continuation [combine *`$wrap`* `(□)` e'_0] through the entire subsidiary evaluation of `(eval ...)`, so we follow the subsidiary evaluation separately.

[eval `(eval (list* ...) static-env)` e'_0]
$\Rightarrow$ [combine [eval `eval` e'_0] `((list* ...) static-env)` e'_0]
$\Rightarrow$ [combine *`eval`* `((list* ...) static-env)` e'_0]
$\Rightarrow$ [combine *`$eval`* ([eval `(list* ...)` e'_0] [eval `static-env` e'_0]) e'_0]
$\vdots$
$\Rightarrow$ [combine *`$eval`* `((`*`$vau`*` (x) #ignore (* x x))` e_1`)` e'_0]
$\Rightarrow$ [eval `(`*`$vau`*` (x) #ignore (* x x))` e_1]
$\Rightarrow$ [combine [eval *`$vau`* e_1] `((x) #ignore (* x x))` e_1]
$\Rightarrow$ [combine *`$vau`* `((x) #ignore (* x x))` e_1]
$\Rightarrow$ [operative `(x) #ignore ((* x x))` e_1]

Although the details of evaluating `(list* ...)` were omitted above, note that, since it was evaluated in e'_0, that is also the environment in which symbol `$vau` was looked up, so that the operator later evaluated in e_1 was the self-evaluating object *`$vau`*.

Splicing this subsidiary work back into the main reduction,

[eval `($lambda (x) (* x x))` e_1]
$\overset{*}{\Rightarrow}$ [combine *`$wrap`* ([eval `(eval (...) static-env)` e'_0]) e'_0]
$\overset{*}{\Rightarrow}$ [combine *`$wrap`* ([operative `(x) #ignore ((* x x))` e_1]) e'_0]
$\Rightarrow$ [applicative [operative `(x) #ignore ((* x x))` e_1]]

To round out the example, suppose e_2 is an environment where symbol `square` is bound to the applicative we just constructed. (We don't need to assume *any* other bindings in e_2.)

[eval `(square 5)` e_2]
$\Rightarrow$ [combine [eval `square` e_2] `(5)` e_2]
$\Rightarrow$ [combine [applicative [operative `(x) #ignore ((* x x))` e_1]] `(5)` e_2]
$\Rightarrow$ [combine [operative `(x) #ignore ((* x x))` e_1] ([eval 5 e_2]) e_2]
$\Rightarrow$ [combine [operative `(x) #ignore ((* x x))` e_1] `(5)` e_2]
$\Rightarrow$ [eval `(* x x)` e'_1] where e'_1 extends e_1 with binding `x` $\mapsto$ `5`
$\Rightarrow$ [combine [eval `*` e'_1] `(x x)` e'_1]
$\Rightarrow$ [combine *`*`* `(x x)` e'_1]
$\Rightarrow$ [combine *`$*`* ([eval `x` e'_1] [eval `x` e'_1]) e'_1]
$\overset{*}{\Rightarrow}$ [combine *`$*`* `(5 5)` e'_1]
$\Rightarrow$ `25`

3.4 apply

Standard applicative ***`apply`*** is used to replace the usual process of operand evaluation with an arbitrary computation to produce the "argument list" to be passed to the underlying combiner of an applicative. The fixpoint of this replacement —where the usual process is replaced by itself— is expressed by equivalence

$$\begin{array}{cl} & (\textit{\textbf{apply}}\ x_0\ (\textit{\textbf{list}}\ x_1\ \ldots\ x_n)\ (\textit{\textbf{get-current-environment}})) \\ = & (x_0\ x_1\ \ldots\ x_n) \end{array} \tag{4}$$

This equivalence is taken to be basic to the purpose of ***apply***; but it only makes sense if x_0 evaluates to an applicative, because if x_0 evaluated to an operative then the $x_{k\geq 1}$ would be evaluated in the first form but not the second. Kernel therefore signals a dynamic type error if the first argument to ***apply*** is not an applicative.[20]

In full generality, the behavior of Kernel's ***apply*** is expressed by equivalence

$$\begin{array}{cl} & (\textit{\textbf{apply}}\ (\textit{\textbf{wrap}}\ c)\ x\ e) \\ = & (\textit{\textbf{eval}}\ (\textit{\textbf{cons}}\ c\ x)\ e) \end{array} \tag{5}$$

(which holds provided c evaluates to a combiner[21]; cf. Equivalence 1). The environment argument e does not occur in Scheme because all Scheme applicative calls are independent of their dynamic environment.[22] Argument e in Kernel is optional; if it's omitted, an empty environment is manufactured for the call. That is,

$$\begin{array}{cl} & (\textit{\textbf{apply}}\ (\textit{\textbf{wrap}}\ c)\ x) \\ = & (\textit{\textbf{eval}}\ (\textit{\textbf{cons}}\ c\ x)\ (\textit{\textbf{make-empty-environment}})) \end{array} \tag{6}$$

Defaulting to an empty environment favors good hygiene (§4.1) by requiring the programmer to explicitly specify any dynamic environment dependency in the call, as in Equivalence 4.

As with ***$lambda*** in §3.3, the general equivalence for ***apply*** (Equivalence 5) translates straightforwardly into a compound operative implementation (where for simplicity of exposition we ignore the optionality of the third argument):

```
($define! apply
   ($lambda (c x e)
      (eval (cons (unwrap c) x) e)))
```

[20]In MacLisp, where generality tended to be pursued for its own sake, it was permissible to ***APPLY*** a fexpr. The resulting situation was in keeping with the general reputation of fexprs as aesthetically unpleasant.

[21]We specifically *do not* require that x evaluate to a proper list. Scheme requires a list here; but in Scheme, all constructed combiners implicitly expect a proper list of operands because they're all applicative. In Kernel, one can explicitly construct an unlimited range of first-class operatives that have no inherent commitment to a proper list of operands, such as `(`***`$vau`***` x #ignore x)` (which constructs an operative equivalent to the underlying operative of ***list***); restricting Equivalence 5 to proper list arguments would therefore be giving up a great deal. Thus, for example, Kernel evaluates (***apply list*** 2) $\Longrightarrow$ 2 .

[22]Scheme supports an extended syntax for ***apply***, taking three or more arguments, in which all arguments after the first are cons'd into a list as by Kernel's ***list**** applicative. This would seem to defy a natural orthogonality between application and improper list construction; at any rate, in Kernel the ***list**** applicative tends to arise in compound operative constructions where ***apply*** would have to be artificially imposed (such as the compound construction of ***$lambda*** in §3.3).

4 Well-behavedness

The Kernel language design generally strives to preserve, consolidate, and further philosophical principles of the Scheme language design. From Scheme design choices such as latent typing, Kernel extrapolates the principle that features should not be introduced into the language design merely to prohibit the programmer from doing things that are potentially dangerous.[23] Recognizing that "lack of constraint" describes some highly undesirable language features as well as some positive ones, Kernel further extrapolates the guideline that potentially dangerous things should be difficult to do *by accident*.

This section considers some of the potentially dangerous things that can be done in Kernel, and to what extent Kernel helps the programmer to negotiate the dangers. The appraisal of Kernel facilities is frank; some of the dangers are handled well, some are only peripherally mitigated, and in one case the language design ([Sh0x]) is still fluid while possible further mitigating measures are contemplated.[24]

4.1 Hygiene

The programmer ordinarily tends to assume that, when a compound combiner is called, syntax in the operands of the combination will be interpreted in its dynamic environment, syntax in the body of the combiner will be interpreted in its local environment, and neither interpretation will be disrupted by the other environment. This clean partitioning of interpretation concerns according to lexical position is referred to, somewhat informally, as *hygiene*.[25]

4.1.1 Variable capturing

In naive macro facilities, the expanded code is interpreted strictly in the dynamic environment, and this can lead to two similar but distinct forms of bad hygiene called *variable capturing*.[26] We consider both forms below; both are rooted in dynamic

[23] Although this principle is extrapolated from latent typing, it does not preclude static typing *per se*; rather, it precludes the most common motivation for static typing, which is the principle that features *should* be introduced for the express purpose of prohibiting the programmer from potentially dangerous actions. It is not necessarily inconsistent with Scheme/Kernel philosophy for the architect of a compound facility to prohibit his creation from being used in ways he didn't intend and for which he doesn't warrant it to be correct. Treading this fine line, techniques that might be loosely characterized as "static typing" are planned for a future version of Kernel (or a descendant language).

[24] The fluid case concerns environment capturing, noted in §4.1.2.

[25] [KoFrFeDu86] cites Barendregt for the informal term *hygiene*: H.P. Barendregt, "Introduction to the lambda calculus", *Nieuw Archief voor Wisenkunde* 2 4 (1984), pp. 337–372; it also notes the formal property of being-free-for in [Kle52] (where it occurs in Kleene's §34 as an auxiliary to his definition of *free substitution*).

[26] [ClRe91] describes four, rather than two, kinds of variable capturing; but this doubling of cases is an artifact of the explicitly two-phase evaluation model required by macros. Each of the two

scoping, not in macros, so fexpr-style facilities such as Kernel's have the potential for both; but in each case, the danger will be largely defused for Kernel by the fact that *$vau*-based compound operatives are statically scoped (i.e., the body is interpreted primarily in the local, lexical environment), so that interpretation subtasks in the dynamic environment are minimized and explicitly flagged out.

(1) The macro may specify variables to be included in the expanded code on the unsafe assumption that, in the dynamic environment, they will have their standard bindings. If the dynamic environment overrides those standard bindings, the variables are said to be captured by the dynamic environment. For example (recalling §3.3 — and imagining that a suitable macro facility were added to Kernel), one might define a naive macro *$lambda* by transformation

($lambda p . b) $\Rightarrow$ (wrap ($vau p #ignore . b))

but then, if one uses *$lambda* in an environment where symbol `wrap` or symbol `$vau` has a nonstandard binding, the expanded code will not have the intended meaning.

Kernel is particularly resistant to this form of capturing because, when building an expanded expression to be evaluated in the dynamic environment (as in §3.3), one tends to insert the actual combiners (here, *wrap* and *$vau*) into the expansion, where they will later self-evaluate, unless one goes out of one's way to deliberately insert the unevaluated variables. However, in all of the compound implementations of Kernel library operatives to date, there has never been any reason to do this (i.e., to insert a literal symbol into an expression constructed for evalation).[27]

(2) Variables in the operands of a combination may be captured by binding constructs in the expanded code. For example, transformation

($or x y) $\Rightarrow$ ($let ((temp x)) ($if temp temp y))

would capture any free variable `temp` in operand y, so that

```
($or foo temp) ⇒ ($let ((temp foo)) ($if temp temp temp))
               = foo
```

(assuming that using *foo* as the conditional test for *$if* won't cause a dynamic type error.[28])

---

cases we describe here is divided by [ClRe91] into two subcases depending on whether the captured variable is processed in the first or second phase of evaluation.

[27] As of this writing, compound implementations have been written for most of the library combiners in the core module of Kernel. [Sh0x]

[28] Kernel *$if* requires the result of the conditional test to be of type *boolean*. This design detail is based ultimately on consistent adherence to the principle of latent typing, which requires that at runtime every object knows its type. In brief: The basic motivation for allowing arbitrary test results is to let the programmer omit the predicate when cdr'ing down a list, but that trick only works if nil ("the empty list") counts as false — in which case, de facto, an instance of nil at runtime doesn't really know whether it is a list or a boolean.

While the first form of variable capture could occur in almost any macro transformation, this second form is limited to those more elaborate transformations that introduce a binding construct. Both hygienic and unhygienic implementations are correspondingly more elaborate; but still, the unhygienic implementations in Kernel are generally more arduous than the hygienic ones.[29]

Here is a hygienic Kernel implementation of the above two-operand ***$or***:

```
($define! $or
   ($vau (x y) e
      ($let ((temp (eval x e)))
         ($if temp temp (eval y e)))))
```

The hygiene of this implementation stems directly from the fact that the introduced binding construct is interpreted in the local, rather than the dynamic, environment.

4.1.2 Context capturing

In the realm of macros, variable capturing commands so much attention that one might be tempted to think of *variable capturing* and *bad hygiene* as synonyms. However, Kernel combiners prominently exhibit some misbehaviors that, while clearly not variable capturing *per se*, nevertheless fit rather well with our notion of "bad hygiene". We will call this second group *context capturing*. Context capturing occurs, in general, when a called combiner accesses information usually meant to belong to the interpretation state of the caller.[30]

In Kernel, combiners can access three kinds of contextual information (not counting the arguments to an applicative, which can't be "captured" because they are presumably always meant to be accessible, or they wouldn't exist); hence there are three kinds of context capturing.

(1) Because Kernel supports first-class operatives —"fexprs"— it is subject to *operand capturing*, whereby a combiner that was thought to be applicative may actually be operative and so unexpectedly access its operands. Consider the following minimalist example[31]:

```
($define! call ($lambda (f x) (f x)))
```

[29] Accidental variable capturing, of either form, would be much more likely if ***$quote*** were included in the Kernel standard library — which, as already noted, it isn't. The propensity of ***$quote*** for bad hygiene also extends to operand capturing (§4.1.2); cf. the note at the end of §4.2.2 on use of ***$quote*** in an alternative implementation of ***$set!***.

[30] In principle, both forms of variable capturing are special cases of context capturing. We won't pursue that connection in this report, though.

[31] Exactly this example (modulo trivialities of Kernel syntax) was used in [Baw88] as a criticism of the behavior of first-class operatives in general — quite accurately, since operand capturing is just exactly the form of context capturing enabled by language support for first-class operatives.

Applicative ***call*** is apparently intended to take a function f and an arbitrary object x, and call f with argument x; we expect equivalence

(*call* f x) $=$ (f x)

This equivalence holds for a tame argument f, such as

(*call* *cos* 0) $\Longrightarrow$ 1

but (recalling the implementation of ***$quote*** from §3.2),

(*call* *$quote* 0) $\Longrightarrow$ x

This behavior violates the encapsulation of ***call***, by capturing and publicizing an operand, `x`, that we'd intended to remain strictly private.

The misbehavior is, in essence, a poorly managed consequence of a type error. Combiner ***call*** expected an applicative as its first argument, but got an operative instead; the intended non-operativity of the argument is suggested (though not, alas, guaranteed) by the absence of a `$` prefix on the parameter name, `f`.[32]

We could replace the misbehavior with an error report by using ***apply*** to call ***f***:

```
($define! call ($lambda (f x) (apply f x)))
```

Now (*call* *$quote* 0) explicitly fails, when ***apply*** attempts to unwrap operative ***$quote***. This is a slightly less than pleasing resolution, because it seems to defy our principle that dangerous things be difficult to do by accident; as another partially mitigating measure, it may be appropriate for Kernel compilers to generate a warning message when a parameter without a `$` prefix occurs directly as an operator.

How serious is this misbehavior? Most any software construct (such as ***call***) is apt to behave in unexpected and even bizarre ways when given an input outside its design specs. That said, certain aspects of the situation are particularly troublesome.

- There is no way in general to distinguish statically (i.e., prior to evaluation) between operands that must be treated as syntax, and operands that will affect the computation only through the results of evaluating them. This was one of the most important objections to fexprs in [Pi80], because it sabotages programs that manipulate other programs as data.[33] Such higher-order programs prominently include compilers, as well as forming an entire genre of custom programs within the Lisp tradition. An important mitigation against this problem in Kernel is the use of environment stabilization techniques, as discussed below in §4.2.

[32] The type constraint is not only "not guaranteed" in the sense that it isn't enforced, but also in the sense that it might not be intended: The naming convention distinguishes variables that should *always* be operative, but does not distinguish between those meant to be strictly non-operative and those meant to range over both operative and non-operative values.

[33] In particular, it sabotages α-conversion, i.e., the semantics-invariant renaming of local variables in a compound combiner; note for example that renaming the local variable in the first implementation of ***call*** would alter the result of (*call* *$quote* 0).

- The possibility of operand capture leads to the prospect that any vulnerable, non-atomic operand might be *mutated.* Especially messy would be operand mutation within the body of a compound operative, where it could alter the behavior of the operative on subsequent calls. Kernel precludes most common opportunites for accidental operand mutation by imposing immutability on selected data structures.[34]

(2) Because Kernel combiners, applicative as well as operative, have the option of accessing their dynamic environments, Kernel is also subject to *environment capturing.* To continue the above example of (the first, unsafe implementation of) ***call***,

(***call*** (***$vau*** #ignore e e) 0) $\Longrightarrow$ [environment]

Here, the value returned is actually *the local environment created for the call,* and thus a child of the static environment of combiner ***call***. This is especially alarming because Scheme programmers commonly rely for encapsulation on the inability of clients to directly access the static environment of an exported applicative (as below in §4.2.2). Its destructive potential is considerably reduced, on a relative scale, by the measures to be described in §4.2; but on an absolute scale it remains a serious threat and, for this reason, proactive measures are under consideration for possible addition to Kernel.[35]

(3) Because Kernel, and Scheme, include the standard applicative ***call-with-current-continuation***, both languages are subject to *continuation capturing.* That this effect satisfies our definition of context capturing should be self-evident. That it also satisfies our intuitive notion of bad hygiene follows from the fact that first-class continuations subsume the behavior of GOTOs, defying the ability of a caller to know whether he'll ever get control back (even in a terminating computation). Far from being decried as "bad hygiene", however, continuation capturing is commonly presented as a major selling point for the Scheme language. Our interest in mentioning it here (besides making a clean sweep of the possible forms of context capturing) is that it demonstrates the ambivalent nature of bad hygiene, which can sometimes be deliberately exploited to achieve useful effects.[36]

4.2 Stability

A source expression has, by definition, an explicit input representation. This means

[34] In particular, data structures ***load***ed from an external file are immutable.

[35] One such measure would be to introduce a facility that moderates environment capturing, analogously to the way Scheme's ***dynamic-wind*** moderates continuation switching. The precise details of such a facility are under intense scrutiny, lest the design miss an opportunity for elegant exception handling.

[36] A telling remark in this regard occurs in [ClRe91], the key paper on Scheme's hygienic macro facility: "We here ignore the occasional need to escape from hygiene; we have implemented a compatible low-level macro system in which non-hygienic macros can be written."

that all of its atoms are either symbols or literal constants; and since there are no literals that denote combiners, all source-expression atomic operators have to be symbols. The upshot is that a Kernel source expression has substantially *no meaning* independent of the environment where it is evaluated.

This phenomenon differs Kernel from Scheme only in degree. Consider Scheme source expression

```
(define square (lambda (x) (* x x)))
```

The resulting compound applicative ***square*** ceases to have its original meaning if the binding of symbol `*` in its static environment is later changed. In the general case, a Scheme processor would have to look up symbol `*` every single time ***square*** is called. These repeated lookups could be eliminated, and other optimizations might also become possible, if it could be proven that the relevant binding of `*` is *stable* (i.e., will never change).

The corresponding Kernel source expression,

```
($define! square ($lambda (x) (* x x)))
```

is potentially even worse off, since the definition itself will misfire unless variables `$define!` and `$lambda` have their standard bindings when the definition is evaluated. If the definition occurs at the top level of the program, at least it will only be evaluated once; but local definitions won't even have that reassurance. It is therefore of great importance in Kernel to cultivate circumstances under which binding stability can be guaranteed.

If an environment is only accessible from a fixed finite set of source code regions, it will usually be straightforward (if tedious) to prove that most of its bindings are stable. The key to stability is therefore to avoid open-ended access to environments. Open-ended access can occur in two ways: *unbounded lexical extent*, and *environment capturing*. Kernel is designed with defenses against both vectors of instability.

4.2.1 Isolating environments

The lexical extent of an environment is the set of all source expressions that are evaluated in it. Whether it is possible for such an extent to be unbounded depends on how the language processor is arranged; the usual case is a global environment used by a read-eval-print loop to evaluate the entire (unbounded) sequence of input expressions. If virtually all environments are descended from the global environment, as is commonly the case in Scheme systems, then mutating standard bindings in the global environment will cause most compound combiners to malfunction.

Fortunately, the flexibility with which Kernel handles environments can be brought to bear on this problem, by making it easy to isolate software subcomponents in non-descendants of the global environment. This is done by introducing a variant of the standard ***$let*** operative, in which the parent of the local environment for the body of the construct is not a child of the surrounding environment, but instead is explicitly specified.

Ordinary ***$let*** is defined by equivalence

$$
\begin{array}{ll}
 & (\texttt{\$let}\ \ ((sym_1\ \ exp_1)\ \ \ldots\ \ (sym_n\ \ exp_n))\ \ .\ \ body) \\
= & ((\texttt{\$lambda}\ \ (sym_1\ \ \ldots\ \ sym_n)\ \ .\ \ body)\ \ exp_1\ \ \ldots\ \ exp_n)
\end{array}
$$

The following implements ***$let*** as a compound operative:

```
($define! $let
   ($vau (bindings . body) dynamic
      ($define! ptree    (map car bindings))
      ($define! operator (list* $lambda ptree body))
      ($define! operands (map cadr bindings))
      (eval (cons operator operands) dynamic)))
```

Our instability-resistant version, which for the sake of discussion we call ***$let-redirect***, takes the intended parent environment as its first operand (evaluated in the dynamic environment, as are the exp_k). The constructed ***$lambda*** expression is then evaluated in the specified parent, so that

$$
\begin{array}{ll}
 & (\texttt{\$let-redirect}\ \ env\ \ ((sym_1\ \ exp_1)\ \ \ldots\ \ (sym_n\ \ exp_n))\ \ .\ \ body) \\
= & ((\texttt{eval}\ \ (\texttt{\$lambda}\ \ (sym_1\ \ \ldots\ \ sym_n)\ \ .\ \ body)\ \ env)\ \ exp_1\ \ \ldots\ \ exp_n)
\end{array}
$$

The following implements ***$let-redirect***:

```
($define! $let-redirect
   ($vau (parent bindings . body) dynamic
      ($define! static   (eval parent dynamic))
      ($define! ptree    (map car bindings))
      ($define! operator (list* $lambda ptree body))
      ($define! operands (map cadr bindings))
      (eval (cons (eval operator static) operands) dynamic)))
```

The combination

```
($let-redirect (make-kernel-standard-environment)
   ((foo foo)
    (quux quux))
   ...)
```

would then evaluate the body, "...", in a local environment with all the Kernel standard bindings, and also bindings of symbols `foo` and `quux` to whatever those symbols were bound to in the surrounding environment when the local environment was set up.[37] Once the local environment has been set up, mutations to the outside environment are not (directly) visible in it.

The current Kernel draft includes an operative ***$let-safe*** as shorthand for ***$let-redirect*** with the standard environment; that is,

```
   ($let-safe bindings . body)
 = ($let-redirect (make-kernel-standard-environment)
                  bindings . body)
```

4.2.2 Restricting environment mutation

Measures to prevent environment capturing were discussed in §4.1.2. Given that environment capturing occurs at all, however, Kernel curtails its destructive potential by requiring, in effect, that ***$define!*** be the only primitive mutator of environments.[38] (In addition to mere damage control, this feature supports the principle that dangerous things should be difficult to do by accident. More on that toward the end of the subsection.) To illustrate how this works, consider the following Scheme code for an encapsulated counter.

```
(define count
  (let ((counter 0))
    (lambda ()
      (set! counter (+ counter 1))
      counter)))
```

The first time ***count*** is called it returns 1, the second time 2, and so on. The internal counter can't be accessed except through ***count*** because it's stored in an environment reachable only through the static-environment reference from ***count*** — and Scheme provides no general way to extract the static environment of an applicative.

Scheme's ***set!*** operative, although it seems innocuous in this example, is an indiscriminately overpowered tool in general. Whereas ***$define!*** creates or modifies a binding in the immediate dynamic environment, ***set!*** finds the visible binding for the specified symbol, and mutates *that binding*, even if the binding is non-local. Consequently, there is no way in Scheme to make a binding visible without also granting the observer the right to mutate it.

[37] Actually, the Kernel standard bindings are not in the local environment, but rather in an ancestor. The local environment binds symbols foo and quux, and has as its parent the environment returned by ***make-kernel-standard-environment***; and *that* environment actually contains no bindings at all, but has a parent containing all the Kernel standard bindings. The significance of this distinction has to do with mutation of bindings, which will be addressed in §4.2.2.

[38] We present ***$define!*** as the primitive here because its behavior should be more familiar to Scheme programmers; but, as we'll see, ***$define!*** is equi-powerful with standard combiner ***$set!***, and from a formal perspective it would actually be slightly simpler to make ***$set!*** the primitive and derive ***$define!*** from it.

Kernel's ***$define!***, like Scheme's ***define***, only mutates its immediate dynamic environment; non-local bindings are unaffected (although they might be locally overridden by a local binding for the same symbol). Given an explicit reference to any environment e, one could mutate e by assembling and evaluating a ***$define!*** combination in e; but because all environment mutators in Kernel must be constructible from ***$define!***, there is no way to mutate e without an explicit reference to it. And, just as there is no generic operation for extracting the static environment of a given combiner, there is also no generic operation for extracting the parent of a given environment. So bindings in Kernel can be made visible without granting the right to mutate them.

The following implements operative ***$set!***[39]:

```
($define! $set!
   ($vau (e s v) dynamic-env
      ($let ((target-env (eval e dynamic-env)))
         (eval (list $define! s
                          (list (unwrap eval) v dynamic-env))
               target-env))))
```

The key to this implementation is that ***$define!*** will always evaluate its second operand in ***target-env***, but we need ***v*** to be evaluated in ***dynamic-env***; so we have to make the second operand to ***$define!*** an operative combination, whose operands therefore *won't* be evaluated before calling it. Here we've used the underlying operative of ***eval***, which will take its unevaluated first operand (***v***, which is the unevaluated third operand of ***$set!***) and evaluate it in the environment that is its unevaluated second operand (***dynamic-env***, the dynamic environment from which ***$set!*** was called).[40]

[39]The implementation of ***$define!*** using ***$set!*** is simpler than that of ***set!*** using ***$define!***:

```
($set! (get-current-environment)
       $define!
       ($vau (s v) e (eval (list $set! e s v) e)))
```

[40]Alternatively, we could have evaluated ***v*** in ***dynamic-env*** before we constructed the ***$define!*** expression, and embedded the result in the constructed expression as the operand to ***$quote*** (§3.2). Following is an implementation using this approach, for illustrative purposes; but note that the use of ***$quote*** introduces a gratuitous additional step to the computation.

```
($define! $set!
   ($vau (e s v) dynamic-env
      ($let ((target-env (eval e dynamic-env))
             (result     (eval v dynamic-env)))
         (eval (list $define! s
                          (list $quote result))
               target-env))))
```

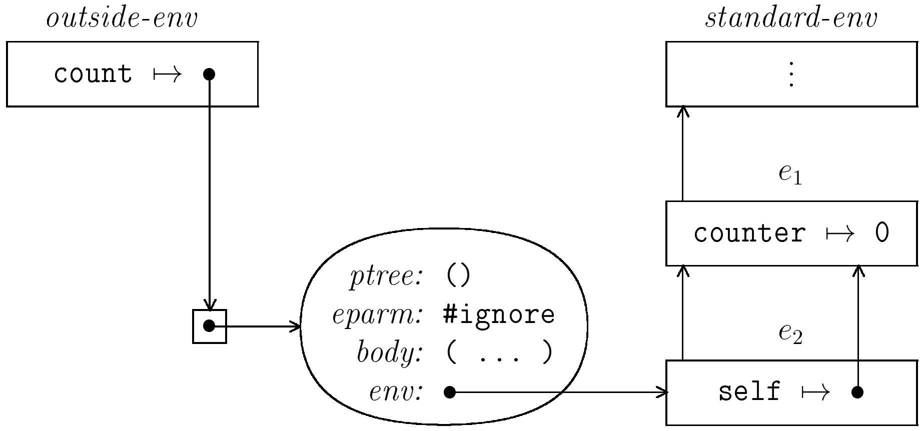

Figure 2: Objects in the first Kernel version of ***count***

Using this device, here is a translation of ***count*** into Kernel:

```
($define! count
   ($let-safe ((counter 0))
      ($let ((self (get-current-environment)))
         ($lambda ()
            ($set! self counter (+ counter 1))
            counter))))
```

The ***\$let-safe*** creates a local environment, call it e_1, with binding `counter` $\mapsto$ `0`, whose parent contains the standard Kernel bindings. The ***\$let*** then creates a local environment e_2 whose parent is e_1, with binding `self` $\mapsto e_1$ in e_2. This last binding arises because ***\$let*** evaluates value expressions for its bindings in the surrounding environment: The surrounding environment is e_1, and the value expression is `(get-current-environment)`, so the value to be bound is e_1. Finally, ***\$lambda*** constructs an applicative with static environment e_2. The arrangement of objects is illustrated by Figure 2.

Note that the need for additional code in the Kernel version of ***count*** —code for binding and accessing variable `self`— is consistent with the language design principles stated at the beginning of the section (§4). Non-local environment mutation (which is a potentially dangerous activity) must have an explicitly specified target (so the programmer has to do it deliberately); and *permission* for non-local environment mutation (dangerous) must be explicitly supported by providing a name for the environment to be mutated (deliberate).

Finally, lest the above Kernel implementation of ***count*** be taken dogmatically, here is an alternative Kernel implementation that more closely parallels the Scheme

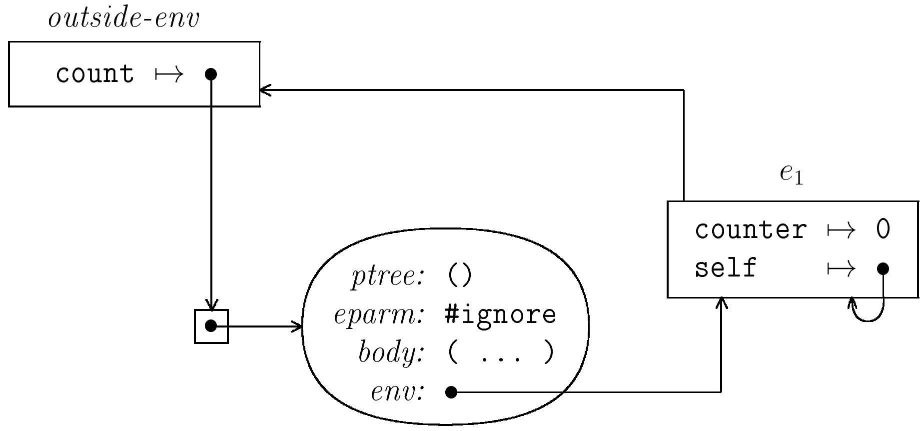

Figure 3: Objects in the second Kernel version of ***count***

version.

```
($define! count
   ($letrec ((counter 0)
             (self    (get-current-environment)))
      ($lambda ()
         ($set! self counter (+ counter 1))
         counter)))
```

The call to ***$let-safe*** has been omitted, reducing the nesting depth to that of the Scheme implementation (and, of course, leaving the local environment vulnerable to mutations of the surrounding environment). The call to ***$let*** has been replaced with ***$letrec***, a variant also available in Scheme that evaluates its binding expressions in the constructed local environment, rather than its surrounding parent environment; thus, the local environment is returned by ***get-current-environment***, and locally bound by variable `self`. The arrangement of objects is illustrated by Figure 3.

5 Conclusions and future work

There are three main perspectives from which a programming language may be studied. The human perspective concerns the feasibility of the language as a medium for programmers to write and read programs; the mathematical perspective concerns the feasibility of stating and proving theorems about programs; and the implementation perspective concerns the feasibility of efficiently interpreting programs — usually, though not necessarily, by compiling them to efficient executable images.

Traditionally, fexprs have been criticized from *all three* perspectives.

The mathematical difficulty with fexprs is rooted in the lambda calculus.

Classical semantics characterizes each programming language $\mathcal{L}$ by its *contextual equivalence* relation: $M =_{\mathcal{L}} N$ for any $\mathcal{L}$-expressions M, N iff for every $\mathcal{L}$-context $C[\Box]$ (an $\mathcal{L}$-context being simply an $\mathcal{L}$-program with a hole in it), $C[M]$ and $C[N]$ are observationally equivalent. If two languages have the same contextual equivalence relation, classically they have the same semantics. For a Lisp language, observational equivalence is generally based on some variant lambda calculus, to which have been added whatever peculiar semantic features are needed for the target language — call-by-value, sequential state, etc. [FeHi92]

Unfortunately, lambda calculus is founded on the assumption that all subterms are subject to rewriting at any time. Thus, *adding* fexprs to lambda calculus means adding a mechanism to actively inhibit subterm rewriting — in other words, a quotation device. But if we have a quotation device, we can create a context $C[\Box]$ such that $C[M]$ and $C[N]$ are observationally equivalent iff M and N are syntactically identical; so, the contextual equivalence relation becomes trivial, and classical semantics goes into meltdown.[41] What we really need is not an extended lambda calculus, but a modified calculus —presumably, *vau calculus*— in which subterms remain passive until their evaluation is initiated by a surrounding context. With no active inhibitor in the calculus, evaluation once initiated proceeds regardless of context. That is the technique we used (without a complete calculus to support it) in §3.3.1: For any expression x, we write $[\text{eval}\ x\ e]$ to designate evaluation of x in environment e, and so on. Then we can define a non-syntactic equivalence $M =_{\mathcal{L}} N$ iff, for all environments e and all contexts C, $C[\text{eval}\ M\ e]$ and $C[\text{eval}\ N\ e]$ are observationally equivalent.

Given the current state of the work, development of a complete vau calculus appears to be a substantial but feasible exercise.

From an implementation perspective, the essential problem with fexprs is that, owing to their potential for operand capturing (§4.1.2), they interfere with static program analysis, and thus, with optimization. This was Pitman's foremost objection to fexprs in [Pi80]. There is no denying that static analysis is more complicated with fexprs than without. However, while the type of combiner designated by an operator is clearly undecidable in general, tedious but straightforward static analysis can decide the type in some —hopefully, common— particular cases by proving that the operator's binding in the local environment will not change. Measures to promote such environment stability are in place in the Kernel design (§4.2); and the primary foil to those measures, environment capturing, is the subject of further language features under consideration (§4.1.2). Moreover, as noted above, the vau calculus should provide a good formal medium for pursuing relevant optimizing transformations. We are therefore cautiously optimistic that, once sufficient groundwork has been laid, Kernel will be compilable with competitive efficiency.

[41] We present the problem at its simplest here. [Wa98] makes a detailed study of contextual equivalence in the presence of various kinds of fexpr facilities.

The preceding sections of this report have considered Kernel mostly from the human perspective. The outlook from that perspective is predominately positive; use of the facilities seems to be straightforward and, for the most part, accident-resistant. Since preliminary assessments of the other perspectives (as described above) are also positive, it appears likely that Kernel will continue to develop into a viable programming medium.

A Kernel meta-circular evaluator (in Scheme)

Here we provide a meta-circular evaluator[42] for a (near) subset of Kernel (current preliminary version, [Sh0x]), written in Scheme. The only omissions from the language are standard combiners that were deemed not to contribute significantly to the illustrative purpose of the exercise. (Besides a few lesser core primitives, we have omitted all non-primitive combiners and most non-core primitives such as arithmetic.)

Because Scheme does not recognize a lexeme `#ignore`, symbol `ignore` is used instead. Consequently, in the interpreted language it isn't possible to have a formal parameter named `ignore`.

Complete code is provided for a working interpreter.

A.1 Evaluator central logic

```
;
; Evaluate an expression in an environment.
;
(define eval
  (lambda (exp env)
    (cond ((pair? exp) (combine (eval (car exp) env) (cdr exp) env))
          ((symbol? exp) (lookup exp env))
          (else exp))))


;
; Evaluate a combination in an environment.
;
(define combine
  (lambda (combiner operands env)
    (if (operative? combiner)
        (operate combiner operands env)
        (combine (unwrap combiner) (map-eval operands env) env))))
```

[42]Re the term *meta-circular*: A meta-circular evaluator for a programming language $\mathcal{L}$ is an interpreter for $\mathcal{L}$ written in another programming language $\mathcal{L}'$. If $\mathcal{L} = \mathcal{L}'$, the definition would be *circular*, meaning that you couldn't understand the definition of $\mathcal{L}$ unless you already knew $\mathcal{L}$; but all such definitions are *meta*-circular, in that no matter how many such evaluators you have, you can't understand any of them unless you already know at least one programming language. [Rey72]

```
;
; Call an operative.
;
(define operate
  (lambda (operative operands env)
    ((get-operative-action operative) operands env)))

;
; Evaluate a list of expressions,
; and return a list of the results.
;
(define map-eval
  (lambda (objects env)
    (map (lambda (x) (eval x env)) objects)))
```

A.2 Interpreter top level

```
;
; The interpreter is simply a read-eval-print loop run on a child
; of the standard environment.  Using a child environment insulates
; the standard environment from mutations of the global environment.
;
(define interpreter
  (lambda ()
    (rep-loop (make-child-environment standard-environment))))

;
; The read-eval-print loop, parameterized by the choice of
; global environment.
;
(define rep-loop
  (lambda (env)
    (display ">>> ")
    (let ((exp (read)))
      (newline)
      (show (eval exp env))
      (newline)
      (newline)
      (rep-loop env))))
```

```
;
; Show an object of the interpreted language.
;
(define show
  (lambda (x)
    (cond ((applicative? x) (display "#[applicative]"))
          ((inert?       x) (display "#inert"))
          ((operative?   x) (display "#[operative]"))
          ((environment? x) (display "#[environment]"))
          ((pair?        x) (display "(")
                            (show (car x))
                            (show-aux (cdr x))
                            (display ")"))
          (else (write x)))))

(define show-aux
  (lambda (x)
    (cond ((null? x))
          ((pair? x) (display " ") (show (car x)) (show-aux (cdr x)))
          (else      (display " . ") (show x)))))
```

A.3 Encapsulated Kernel data types

```
;;;;;;;;;;;;;;;;;;;;;
; encapsulated types ;
;;;;;;;;;;;;;;;;;;;;;
;
; Encapsulated objects in the interpreted language are represented by
; procedures in the meta-language.  The representing meta-procedure
; takes a symbol as argument, and returns one of several fields.
; Internally, it's an alist.
;

(define make-encapsulated-object
  (lambda (type alist)
    (lambda (message)
      (if (eq? message 'type)
          type
          (cdr (assoc message alist))))))

(define make-encapsulated-type-predicate
  (lambda (type)
    (lambda (object)
      (and (procedure? object)
           (eq? (object 'type) type)))))

;;;;;;;;;;;;;
; operative ;
;;;;;;;;;;;;;
;
; An operative has type 'operative, and attribute 'action whose value
; is a meta-procedure that takes the operands and environment as its
; parameters.
;

(define make-operative
  (lambda (action)
    (make-encapsulated-object 'operative
                              (list (cons 'action action)))))

(define operative? (make-encapsulated-type-predicate 'operative))

(define get-operative-action (lambda (x) (x 'action)))
```

```
;;;;;;;;;;;;;;;
; applicative ;
;;;;;;;;;;;;;;;
;
; An applicative has type 'applicative, and attribute 'underlying
; whose value is a combiner (either applicative or operative).
;   The principal constructor is called "wrap" instead of
; "make-applicative", and the accessor is called "unwrap" instead of
; "get-applicative-underlying".  Note the secondary constructor
; metaproc->applicative.
;

(define wrap
  (lambda (combiner)
    (make-encapsulated-object 'applicative
                              (list (cons 'underlying combiner)))))

(define applicative? (make-encapsulated-type-predicate 'applicative))

(define unwrap (lambda (x) (x 'underlying)))

(define metaproc->applicative
  (lambda (metaproc)
    (wrap (make-operative (lambda (operands env)
                            (apply metaproc operands))))))

;;;;;;;;;
; inert ;
;;;;;;;;;
;
; The inert value has type 'inert and no attributes.
;

(define inert (make-encapsulated-object 'inert ()))
(define inert? (make-encapsulated-type-predicate 'inert))
```

```
;;;;;;;;;;;;;;;
; environment ;
;;;;;;;;;;;;;;;
;
; An environment has type 'environment, and attribute 'frames whose
; value is a list of lists of name-value pairs.  Lookup starts with
; the first list of name-value pairs.
;

(define make-empty-environment
  (lambda ()
    (make-encapsulated-object 'environment
                              (list (cons 'frames (list ()))))))

(define make-child-environment
  (lambda (env)
    (make-encapsulated-object 'environment
      (list (cons 'frames (cons () (get-environment-frames env)))))))

(define environment? (make-encapsulated-type-predicate 'environment))

(define get-environment-frames (lambda (x) (x 'frames)))


;
; Returns the value bound to name if there is one, otherwise fails.
;
(define lookup
  (lambda (name env)
    (cdr (get-binding name (get-environment-frames env)))))
```

```
;
; Locally binds name to value.  Mutates an existing local binding,
; or creates one.  Be careful of the parameter order:
; The env is first here, last in match!.
;
(define add-binding!
  (lambda (env name value)
    (let ((frames (get-environment-frames env)))
      (let ((binding (assoc name (car frames))))
        (if (eq? binding #f)
            (set-car! frames (cons (cons name value) (car frames)))
            (set-cdr! binding value))))))

;
; Locally binds a formal-parameter-tree to an object.
; Be careful of the parameter order:
; The env is last here, first in add-binding!.
;
(define match!
  (lambda (ptree object env)
    (cond ((equal?  ptree 'ignore))
          ((symbol? ptree) (add-binding! env ptree object))
          ((pair?   ptree) (match! (car ptree) (car object) env)
                           (match! (cdr ptree) (cdr object) env))
          (else #f))))

;
; Returns the binding for name if there is one, otherwise
; returns #f.  This is the only procedure in the interpreter
; that takes a frames parameter instead of an environment.
;
(define get-binding
  (lambda (name frames)
    (if (null? frames)
        #f
        (let ((binding (assoc name (car frames))))
          (if (pair? binding)
              binding
              (get-binding name (cdr frames)))))))
```

A.4 Kernel standard environment

```
;
; The standard environment contains bindings for all built-in
; combiners.
;
(define standard-environment (make-empty-environment))
(add-binding! standard-environment '+  (metaproc->applicative +))
(add-binding! standard-environment '*  (metaproc->applicative *))
(add-binding! standard-environment '<? (metaproc->applicative <))
(add-binding! standard-environment '>? (metaproc->applicative >))
(add-binding! standard-environment '=? (metaproc->applicative =))
(add-binding! standard-environment
              'wrap   (metaproc->applicative wrap))
(add-binding! standard-environment
              'unwrap (metaproc->applicative unwrap))
(add-binding! standard-environment
              'exit   (metaproc->applicative
                        (lambda () (terminate-the-interpreter))))
                      ; a lambda expression is used here to defer
                      ; lookup of symbol "terminate-the-interpreter"
                      ; because the symbol hasn't been bound yet.

(add-binding! standard-environment '$if
  (make-operative
    (lambda (operands env)
      (if (eval (car operands) env)
          (eval (cadr operands) env)
          (if (null? (cddr operands))
              inert
              (eval (caddr operands) env))))))
```

```
(add-binding! standard-environment '$vau
  (make-operative
     (lambda (operands static-env)
        (let ((parameter-tree (car  operands))
              (env-parameter  (cadr operands))
              (body           (cddr operands)))
          (make-operative
            (lambda (operands dynamic-env)
              (let ((local-env (make-child-environment static-env)))
                (match! parameter-tree operands    local-env)
                (match! env-parameter  dynamic-env local-env)
                (eval-sequence body local-env))))))))

    (define eval-sequence
      (lambda (sequence env)
        (cond ((null?      sequence ) inert)
              ((null? (cdr sequence)) (eval (car sequence) env))
              (else (eval          (car sequence) env)
                    (eval-sequence (cdr sequence) env)))))

(add-binding! standard-environment '$define!
  (make-operative
    (lambda (operands env)
      (add-binding! env (car operands) (eval (cadr operands) env))
      inert)))
```

A.5 Termination

```
;
; The special case of interpreter termination shouldn't intrude
; on the central logic of the evaluator, therefore termination
; should be entirely encapsulated within meta-procedure
; terminate-the-interpreter.  The way to do that is to use a
; first-class continuation.
;

;
; This binding is a stub; lastly we'll set! it to what we want.
;
(define terminate-the-interpreter ())

;
; Capture the continuation here at the end of building the
; interpreter; normal termination passes 'interpreter-terminated
; to the captured continuation.
;
(call-with-current-continuation
  (lambda (c)
    (set! terminate-the-interpreter
          (lambda () (c 'interpreter-terminated)))
    'interpreter-constructed))
```

B References

[AbSu96] Harold Abelson and Gerald Jay Sussman with Julie Sussman, *Structure and Interpretation of Computer Programs*, Second Edition, MIT Press, 1996. Available (as of December 2002) at URL:
`http://mitpress.mit.edu/sicp/sicp.html`

The second edition of the Wizard Book. (See [Ra02, "Wizard Book"].)

[Bar79] "PL/I Preprocessor", Chapter 15 of Robert Arthur Barnes, *PL/I for Programmers*, New York: Elsevier North Holland, Inc., 1979, pp. 405–437.

The macro preprocessor to end all macro preprocessors of the non-Lisp HLL world.

[Baw88] Alan Bawden, *Reification without Evaluation*, memo 946, MIT AI Lab, June 1988. Available (as of December 2002) at URL:
`http://www.ai.mit.edu/publications/pubsDB/pubs.html`

[ClRe91] William Clinger and Jonathan Rees, "Macros that work", *POPL '91*: *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, Orlando, Florida, January 21–23, 1991, pp. 155–162.

Draws together the best of previous work reconciling macros with hygiene. The introduction extensively discusses the various problems that can arise with macros.

[CrFe91] Erik Crank and Matthias Felleisen, "Parameter passing and the lambda calculus", *POPL '91*: *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, Orlando, Florida, January 21–23, 1991, pp. 233–244. Available (as of January 2003) at URL:
`http://www.ccs.neu.edu/scheme/pubs/`

[FeHi92] Matthias Felleisen and Robert Hieb, "The Revised Report on the Syntactic Theories of Sequential Control and State", *Theoretical Computer Science* 103 no. 2 (September 1992), pp. 235–271. Available (as of January 2003) at URL:
`http://www.ccs.neu.edu/scheme/pubs/`

[KeClRe98] Richard Kelsey, William Clinger, and Jonathan Rees, editors, "Revised^5 Report on the Algorithmic Language Scheme", 20 February 1998. Available (as of November 2002) at URL:
`http://www-swiss.ai.mit.edu/projects/scheme/index.html`

[Kle52] S.C. Kleene, *Introduction to Metamathematics*, Princeton, N.J.: Van Nostrand, 1952.

This excellent book is (as of January 2002) still in print, by North-Holland, in its thirteenth impression. Corrections were made through the seventh impression in 1971.

[KoFrFeDu86] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba, "Hygienic macro expansion", *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, 1986, pp. 151–159.

[McC60] John McCarthy, "Recursive Functions of Symbolic Expressions and Their Computation by Machine", *Communications of the ACM* 3 no. 4 (April 1960), pp. 184–195.

This is the original reference for Lisp.

[Mi93] John C. Mitchell, "On Abstraction and the Expressive Power of Programming Languages", *Science of Computer Programming* 212 (1993) [Special issue of papers from Symposium on Theoretical Aspects of Computer Software, Sendai, Japan, September 24–27, 1991], pp. 141–163. Also, a version of the paper appeared in: Takayasu Ito and Albert R. Meyer, editors, *Theoretical Aspects of Computer Science: International Conference TACS'91* [Sendai, Japan, September 24–27, 1991] [*Lecture Notes in Computer Science* 526], Springer-Verlag, 1991, pp. 290–310. Available (as of December 2002) at URL:
`http://theory.stanford.edu/people/jcm/publications.htm`

[NEPLS] *New England Programming Languages and Systems Symposium Series* (NEPLS). URL (as of November 2002):
`http://www.nepls.org/`

[Pi80] Kent M. Pitman, "Special Forms in Lisp", *Proceedings of the 1980 ACM Conference on Lisp and Functional Programming*, 1980, pp. 179–187.

This is one of those papers that gets cited a lot, by papers within its particular clique, because it carefully and clearly develops some basic conclusions that everyone later wants to take as given. In a nutshell: Fexprs are badly behaved (second opinion: they're ugly, too), so future Lisps should use macros instead.

[Pi83] Kent M. Pitman, *The revised MacLisp Manual* (Saturday evening edition), MIT Laboratory for Computer Science Technical Report 295, May 21, 1983.

[Ra02] Eric S. Raymond, *The Jargon File*, version 4.3.3, 20 September 2002. Available (as of November 2002) at URL:
`http://www.tuxedo.org/~esr/jargon/`

[Rey72] John C. Reynolds, "Definitional Interpreters for Higher-Order Programming Languages", *Proceedings of the ACM Annual Conference*, 1972, (vol. 1) pp. 717–740.

Unifies several previous attempts to define language semantics under the name "meta-circular evaluator".

He describes static scoping, not by that name, and says —forcefully— that everyone agrees that well-designed languages must work that way.

[Sh0x] John N. Shutt, "Revised^{-1} Report on the Kernel Programming Language". To appear.

[Sta75] Thomas A. Standish, "Extensibility in Programming Language Design", *SIGPLAN Notices* 10 no. 7 (July 1975) [*Special Issue on Programming Language Design*], pp. 18–21.

A retrospective survey of the subject, somewhat in the nature of a postmortem. The essence of Standish's diagnosis is, as I understand it, that the extensibility features required an expert to use them. He notes that when a system is complex, modifying it is complex.

He classifies extensibility into three types: *paraphrase* (defining a new feature by showing how to express it with pre-existing features — includes ordinary procedures as well as macros); *orthophrase* (adding new facilities that are orthogonal to what was there — think of adding a file system to a language that didn't have one); and *metaphrase* (roughly what would later be called "reflection").

[StGa93] Guy L. Steele Jr. and Richard P. Gabriel, "The Evolution of Lisp", *SIGPLAN Notices* 28 no. 3 (March 1993) [Preprints, *ACM SIGPLAN Second History of Programming Languages Conference*, Cambridge, Massachusetts, April 20–23, 1993], pp. 231–270.

[Wa98] Mitchell Wand, "The Theory of Fexprs is Trivial", *Lisp and Symbolic Computation* 10 no. 3 (May 1998), pp. 189–199. Available (as of December 2002) at URL:
`http://www.ccs.neu.edu/home/wand/pubs.html`

[WaFr86] Mitchell Wand and Daniel P. Friedman, "The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower", *Lisp and Symbolic Computation* 1 no. 1 (1988), pp. 11–37.