First-Class Extents

Shinn-Der Lee and Daniel P. Friedman
Computer Science Department
Indiana University
Bloomington, Indiana 47405
sdlee@cs.indiana.edu *dfried@cs.indiana.edu*

August 4, 1992

Abstract

Adding environments as first-class values to a language can greatly enhance its expressiveness. But first-class environments do not mesh well into a lexically scoped language since they rely on identifiers (variable names). By distinguishing variables from identifiers and therefore extents from environments, we present an alternative: first-class extents. First-class extents are defined on variables rather than identifiers and are therefore immune to name capturing problems that plague first-class environments. Then by distinguishing variables from locations and therefore extents from stores, our first-class extents can coexist with imperative features and still allow tail-recursion to be properly implemented as iteration.

To test our claims, we extend Scheme with a collection of features that are essential to first-class extents, give a denotational semantics for the extension, and demonstrate that it can be fully embedded into Scheme albeit losing tail-recursiveness. Then we show how first-class extents lead to a way of extending Scheme with object-oriented programming features.

1 Introduction

In traditional languages a variable with the name i is mapped to a value v by an *environment* map ρ:

$$i \overset{\rho}{\mapsto} v.$$

Adding environments as first-class values to a language can greatly enhance its expressiveness. MIT Scheme [1, 15], T [18], and Symmetric Lisp [8] have shown that records, structures, closures, modules, classes, abstract data types, and inheritance can all be expressed with first-class environments. But since first-class environments rely on identifiers (variable names), α-conversion (renaming) of variables could lead to context anomalies due to inadvertent name capturing.

To illustrate, consider the following expression in MIT Scheme:

```
(let ((x 1))
  (let ((env (let ((y 3)) (make-environment))))
    (let ((p (eval '(lambda () x) env)))
      (p))))
```

The environment *env* constructed by **make-environment** inherits from its enclosing environment the bindings of x and y whose values are 1 and 3, respectively. The code '(lambda () x) is then evaluated within *env*. Hence invoking the procedure p returns 1, the value of the inherited binding. To show that α-conversion does not work with first-class environments we α-convert y to x in the above expression:

```
(let ((x 1))
  (let ((env (let ((x 3)) (make-environment))))
    (let ((p (eval '(lambda () x) env)))
      (p))))
```

The renaming should cause no problem since x does not occur free in the body expression (**make-environment**). But *env* now inherits two bindings of x and the inner one shadows the outer one. The result of invoking p is therefore 3, not 1. So changing a variable's name does change the expression's meaning.

The problem with first-class environments is that they rely on a variable's name, its identifier, rather than on the variable itself. In a lexically scoped language α-conversion implies that a variable can have many different names. Hence, relying on a variable's name is problematic. The obvious alternative is to rely on the semantics of an identifier, the variable. So a variable with the name i should be mapped to a value v using the composition of two maps: a map ρ that binds i to a variable a and a map ϵ that associates a to v:

$$i \overset{\rho}{\mapsto} a \overset{\epsilon}{\mapsto} v.$$

We continue to call the map ρ an environment and use it to express static/dynamic scope. The newly introduced map ϵ is called an *extent*, it allows us to explicitly characterize static/dynamic extent of a language. Now we can make the extent map a first-class value and still retain static scope for the variables. That is, first-class extents allow us to define variables with static scope and dynamic extent. They are therefore immune to name capturing and α-conversion anomalies. With first-class extents we retain most of the expressiveness of first-class environments and still enjoy the security and modularity of static scope.

In this report we use Scheme [4] to test our claims. We choose Scheme for three reasons. First, Scheme has only a few basic constructs, greatly simplifying the presentation. Second, first-class extents are compatible with Scheme's static scope; first-class environments, however, are not. Third, a specialized use of first-class extents already exists in most Scheme implementations, it is the fluid binding operation **fluid-let** [1].

Extending an imperative language such as Scheme with first-class extents raises an immediate concern: the interaction between dynamic extent and side effects. The two notions must be separated. Thus the mapping of a variable a to a value v should be further refined into the composition of two maps: an extent map ϵ that takes a to a store location l and a *store* map θ that binds l to v:

$$i \overset{\rho}{\mapsto} a \overset{\epsilon}{\mapsto} l \overset{\theta}{\mapsto} v.$$

That is, we distinguish variables from locations. One might be tempted not to make this distinction, but to use the original ϵ to model both extent and side effects. That, however, would mean that the language is not properly tail-recursive, since extent must be simulated by side effects with shallow binding [7].

The rest of this report is organized as follows. In section 2 we describe FCE, an extension of Scheme with a collection of operations that is essential to first-class extents. We then show that FCE can be embedded into Scheme. In section 3 we demonstrate how first-class extents lead to a natural way of extending Scheme with object-oriented programming features. In section 4 we compare our first-class extents to related work on first-class environments and first-class stores. Finally we state our conclusion in section 5.

2 FCE

This section extends Scheme [4] with a set of essential first-class extent features. The extension is called FCE and a complete formal description of it, in the form of a denotational semantics, is deferred to the appendix.

We first introduce two basic operations: **make-empty-extent** and **with-extent**. **Make-empty-extent** is a zero argument procedure that produces a distinct empty extent with each invocation. The special form (**with-extent** *ext body*) evaluates the expression *body* within the extent denoted by the expression *ext*. It returns the value of *body*.

At any point during a computation there can be multiple extents in effect at the same time. The one that is most recently in effect is the *most recent extent* (MRE) and the one that is least recently in effect is the *least recent extent* (LRE). For example, with respect to the expression *body* in

```
(with-extent ext1
  (with-extent ext2
    body))
```

the MRE is *ext2* and the *previous* most recent extent is *ext1*. Similarly in

```
(with-extent ext1
  (with-extent (with-extent ext2 body1)
    body2))
```

the MRE and the previous most recent extent of *body1* are *ext2* and *ext1*, respectively. But the MRE of *body2* is the extent denoted by (**with-extent** *ext2 body1*), not *ext2*. The previous most recent extent, however, is still *ext1*.

We assume the existence of a *base* extent that is always the LRE. All programs are evaluated within this base extent. Thus an ordinary Scheme program can be seen as a program that uses no other first-class extent features. In order to treat all extents equally, the base extent is made available by invoking the thunk **get-base-extent**.

We next define variable reference, assignment, new binding creation, and binding unshadowing (from now on the term binding means *extent binding* if not stated otherwise). In the process we also take into consideration the meanings of the relevant Scheme operations: variable reference, **lambda**, **set!**, **call-with-current-continuation** (**call/cc**), and top-level **define**.

2.1 Variable Reference and Assignment

With multiple extents in effect, a variable reference denotes the value defined in the most recently installed extent that has a binding for the variable. Such a binding is the variable's *current binding*. Metaphorically, the multiple extents in effect form a stack whose top element is the MRE. A reference searches the stack from top to bottom. The first binding found is its current binding.

An assignment (**set!** *id exp*) always updates the current binding of the variable denoted by the identifier *id*. To update the binding in any extent we can make that extent the MRE

```
(define id exp) => (with-extent (get-base-extent)
                                (def id exp))
```

Figure 1: **Define.**

```
(make-extent (id exp) ...) => (let ((val exp) ...)
                                 (let ((ext (make-empty-extent)))
                                   (with-extent ext
                                     (begin (def id val) ... ext))))
```

Figure 2: **Make-extent.**

and then use **set!**.

2.2 New Binding Creation

New bindings established by a procedure invocation are added to the LRE. To add bindings to the MRE we employ the new special form (**def** *id exp*). To add bindings to any other extent we can make that extent the MRE and then use **def**.

There are three cases for the operation (**def** *id exp*). First, if *id* is an unbound identifier, it becomes a globally bound identifier and is associated with a new variable. In the base extent this new variable is bound to a fresh location whose value is unspecified. In addition, in the MRE the new variable is bound to another fresh location whose value is that of the expression *exp*. Second, if *id* is bound and the MRE has no binding for it, a new binding associating *id* to a fresh location whose value is *exp* is added to the MRE. Third, if *id* is bound and the MRE has a binding for it, the **def** becomes an assignment operation and the value of the current binding in the MRE is updated.

The **def** special form is a generalization of Scheme's top-level **define**: a top-level **define** expression in Scheme is equivalent to a **def** expression when the base extent is the MRE; see Figure 1 (we assume that the new variables introduced by syntactic extensions are hygienic). When the identifier *id* is not bound **define** allocates a new location for it and adds a new binding to the store. Similarly **def** allocates a new variable for *id* and adds a binding to the base extent. When *id* is already bound **define** becomes an assignment, as does **def**.

With **def** an extent consisting of bindings of a number of variables can be incrementally constructed using the **make-extent** syntax defined in Figure 2. Finally the definition of **fluid-let** [1] in Figure 3 in terms of first-class extents shows why we claim that **fluid-let** is a specialized use of first-class extents.

```
(fluid-let ((id exp)) body) => (with-extent (make-extent (id exp)) body)
```

Figure 3: **Fluid-let.**

2.3 An Example

Next, as an example we develop a simple register machine similar to that described in [15]. The machine has three registers, namely *r0*, *r1*, and *r2*. The abstract data type of registers is shown in Figure 4. In it each register created by *make-reg* is a separate extent and the exact same variable *contents* is being used in each of the registers. Since the variable *contents* is visible only to the *fetch* and *assign* instructions, the security of registers is maintained. We can use the same technique to define record structures in general: a record is an extent with each of its fields being a binding. In particular a register is represented as a record of a single field called *contents*.

In addition to *fetch* and *assign* the machine has three other instructions: the unconditional *goto*, the conditional *branch*, and *halt*. The machine is defined as an extent; see Figure 5.

A program of the machine is also an extent. Each of its bindings associates a label with a thunk that represents a non-empty sequence of instructions. By invoking such a thunk control is transferred to the first instruction of the sequence. Each sequence of instructions consists of any number of *assign* or *fetch* instructions followed by one of the other three instructions: *halt*, *goto*, or *branch*. This ensures that the thunks are in iterative form [14]. The entry point of a program is by convention always the label *start*. For instance, Figure 6 shows a program that computes the greatest common divisor (GCD) of the two numbers found in registers *r0* and *r1*. The answer is returned through *r0*. Therefore if *gcd-regs* is an extent consisting of the three machine registers, the following expression computes the GCD:

```
(let ((contents "unspecified"))
  (def make-reg (lambda (val) (make-extent (contents val))))
  (def fetch (lambda (reg) (with-extent reg contents)))
  (def assign (lambda (reg val) (with-extent reg (def contents val)))))
```

Figure 4: Register.

```
(def machine
  (make-extent
    (fetch fetch)
    (assign assign)
    (halt (lambda () 'done))
    (goto (lambda (label) (label)))
    (branch (lambda (test then-lab else-lab) (goto (if test then-lab else-lab))))))
```

Figure 5: Register machine.

```
(def gcd-prog
  (make-extent
    (start (lambda () (goto loop)))
    (loop (lambda () (branch (zero? (fetch r1)) done next)))
    (next (lambda () (assign r2 (remainder (fetch r0) (fetch r1)))
                     (assign r0 (fetch r1))
                     (assign r1 (fetch r2))
                     (goto loop)))
    (done (lambda () (halt)))))
```

Figure 6: GCD program.

```
(with-extent machine
  (with-extent gcd-prog
    (with-extent gcd-regs
      (begin (goto start) (fetch r0)))))
```

2.4 Binding Unshadowing

The **with-extent** operation imposes a shadowing semantics on bindings: a more recent extent's bindings shadow those of a less recent extent. The dual is an operation that undoes the work done by some **with-extent** on a variable. It unshadows the variable's current binding and exposes its most recently shadowed one. We name such an operation **with-shadowed** and define its syntax as (**with-shadowed** *id body*). During the evaluation of the expression *body*, the *previous* current binding of *id* temporarily becomes the current binding, any reference to *id* in the evaluation of *body* gets its value from the previus current binding. Inductively, when an unshadowing operation is within another unshadowing operation of the same variable, the *previous* previous current binding becomes the current binding, and so forth. Given this semantics it is an error when there is no shadowed binding. Hence we also provide a predicate to tell whether that is the case. The expression (**shadowed?** *id*) is true if the current binding of the variable denoted by the identifier *id* is shadowing another binding of *id*; it is false otherwise.

To illustrate, the following *sum-x* computes the sum of all bindings of x, except the last one, in all the extents currently in effect:

```
(def sum-x
  (lambda ()
    (if (shadowed? x)
        (+ x (with-shadowed x (sum-x)))
        0)))
```

As an example, assuming that extents *ext1*, *ext2*, and *ext3* bind x to 1, 2, and 3, respectively, and that the LRE has the only other binding for x. Then the following expression returns 6:

```
(with-extent ext1
  (with-extent ext2
    (with-extent ext3
      (sum-x))))
```

The following derivation depicts the different values the same x denotes within different extents:

```
   (sum-of-x)
→  (+ 3 (with-shadowed x (sum-x)))
→  (+ 3 (with-shadowed x
            (+ 2 (with-shadowed x (sum-x)))))
→  (+ 3 (with-shadowed x
            (+ 2 (with-shadowed x
                     (+ 1 (with-shadowed x (sum-x)))))))
→  (+ 3 (with-shadowed x
            (+ 2 (with-shadowed x
                     (+ 1 (with-shadowed x 0))))))
→  (+ 3 (+ 2 (+ 1 0)))
```

With-shadowed has many applications. Let variables x and y be defined in both extents *ext1* and *ext2*. Then consider combining the two extents in such a way that the current bindings of x and y are those in *ext1* and *ext2*, respectively. Installing the extents in either order does not work. Forming a new extent *ext3* out of the desired bindings will not work, since any side effects made to x or y of *ext3* do not affect the corresponding bindings in *ext1* or *ext2*. With **with-shadowed**, however, we can devise the following solution:

```
(def extended-assign
  (make-extent
    (assign (lambda (reg val)
              (display (fetch reg))
              (with-shadowed assign (assign reg val))
              (display (fetch reg))))))
```

Figure 7: *Extended-assign.*

```
(with-extent ext1
  (with-extent ext2
    (with-shadowed x
      exp)))
```

During the evaluation of the expression *exp* the current binding of *y* is the one in *ext2*. As for *x* the current binding is the one in *ext1*, since the one in *ext2* has been unshadowed by the **with-shadowed** operation.

Another major use of unshadowing is to extend the definition of a procedure. For instance, continuing the register machine example of section 2.3, the *extended-assign* of Figure 7 extends the *assign* instruction of Figure 4 so that each time a register is updated its old and new contents are displayed. Hence in addition to computing the GCD of two numbers, the following also displays the contents of each register when it is updated:

```
(with-extent machine
  (with-extent gcd-prog
    (with-extent gcd-regs
      (with-extent extended-assign
        (begin (goto start) (fetch r0))))))
```

So far we have extended Scheme with two major first-class extent features: a **def** operation to incrementally add bindings to an extent and a **with-shadowed** operation to unshadow bindings that are shadowed by nested extents. At any point during a computation the extents installed, including the always present base extent, together with the established unshadowings constitute the (*current*) *effective extent* of the computation at that point. Effective extent is an important notion since it completely determines the current binding of every variable reference and assignment. In the following we make effective extents first-class values as well.

```
(lambda/e args body) => (let ((eff-ext (get-effective-extent)))
                           (lambda args
                             (with-effective-extent eff-ext body)))
```

Figure 8: **Lambda/e.**

```
(def call/cc/e
  (lambda (f)
    (let ((eff-ext (get-effective-extent)))
      (call/cc
        (lambda (k)
          (f (lambda (v) (with-effective-extent eff-ext (k v))))))))))
```

Figure 9: **Call/cc/e.**

2.5 First-Class Effective Extents

Procedures and continuations take on different meanings depending on whether or not they are closed over the definition-time effective extent. We do not impose an arbitrary decision on this issue. Instead, in FCE, by the principle of orthogonality the forming of procedures and continuations, and the decision whether to close them over the definition-time effective extent are independent of each other. Hence we provide two new operations: **get-effective-extent** and **with-effective-extent**. **Get-effective-extent** is a zero argument procedure that returns as a first-class value the current effective extent. The special form (**with-effective-extent** *eff-ext body*) evaluates the expression *eff-ext* to an effective extent, installs it as the new current effective extent, evaluates the expression *body* within the new current effective extent, then the old current effective extent is restored and the value of *body* is returned.

We let **lambda** construct procedures that do not close over the definition-time effective extent. Then we can construct procedures that close over the definition-time effective extent using the syntactic extension **lambda/e** defined in Figure 8. Briefly, when such a procedure is built *eff-ext* is bound to the definition-time effective extent. Subsequently when the procedure is invoked, the procedure's body expressions are evaluated within *eff-ext* rather than the invocation-time effective extent.

Similarly we let **call/cc** obtain continuations that do not close over the definition-time effective extent and then use it to define **call/cc/e**, which obtains continuations that close over the definition-time effective extent; see Figure 9. That is, *eff-ext* is bound to

```
(def debugger-assign
  (make-extent
    (assign (lambda (reg val)
              (with-shadowed assign (assign reg val))
              (call/cc/e debugger)))))

(def debugger
  (with-extent machine
    (with-extent debugger-regs
      (lambda/e (break-point)
        (display-gcd-regs gcd-regs)
        (assign r0 (+ 1 (fetch r0)))
        (break-point "unspecified")))))

(def display-gcd-regs
  (lambda (gcd-regs)
    (display (fetch r0))
    (display (fetch r1))
    (display (fetch r2))))
```

Figure 10: Break point facilities.

the definition-time effective extent and k is bound to the current continuation. Then a procedure acting as a continuation is built out of *eff-ext* and k, and passed to f. Later, invoking such a procedure installs *eff-ext* as the new current effective extent before it passes the value v to k.

Continuing the register machine example of section 2.3 we use **lambda/e**, **call/cc/e**, and **with-shadowed** to implement a break point mechanism so that each time a register is assigned a new value, the machine transfers the control to a debugger. The debugger receives such a break point, increments its own *r0* register, and resumes the stopped program. Figure 10 shows the needed additional facilities. Using **with-shadowed**, the *debugger-assign* instruction invokes the original *assign* to update a register's contents. A break point, which is the current continuation and effective extent, is then obtained with **call/cc/e** and passed to the debugger. Later, the debugger resumes the program by invoking the break point. Since *debugger* is defined with **lambda/e**, it is closed over its definition-time effective extent. Therefore the *r0* in *debugger* denotes the debugger's own *r0* register.

The following code shows how to invoke the GCD program with the break point mechanism:

```
(with-extent machine
  (with-extent gcd-prog
    (with-extent gcd-regs
      (with-extent debugger-assign
        (begin (goto start) (fetch r0))))))
```

And if the debugger's *r0* register is defined as

```
(def debugger-regs (make-extent (r0 (make-reg 0))))
```

the number of times a program is stopped can be obtained with

```
(with-extent machine
  (with-extent debugger-regs
    (fetch r0)))
```

2.6 Tail-Recursiveness

The three **with-** operations are not tail-recursive. Each one of them sets up a new effective extent, evaluates its body expression within the new effective extent, and then resets the old effective extent before the body expression's value is returned. Although they are not tail-recursive they do have counterparts that are: **use-extent**, **use-shadowed**, and **use-effective-extent**. The expression (**use-effective-extent** *eff-ext*) installs the effective extent denoted by the expression *eff-ext* as the new current effective extent in the subsequent computation. Similarly the expression (**use-extent** *ext*) extends the current effective extent so that the extent denoted by the expression *ext* becomes the MRE in the subsequent computation. The expression (**use-shadowed** *id*) updates the current effective extent so that the previous current binding of *id* becomes the new current binding in the subsequent computation.

Figure 11 defines the **with-** operations in terms of their **use-** counterparts. In the auxiliary procedure *use*, *eff-ext* is bound to the current effective extent and the new current effective extent is installed by invoking the *install-new-eff-ext* thunk. Then by calling the *action* thunk the *body* expression is evaluated within the new current effective extent. Afterwards the old current effective extent *eff-ext* is reinstated and the value of *body* is returned.

2.7 Predicates

FCE has four more predicates, namely **extent?**, **effective-extent?**, **empty-extent?**, and **def?**. The first two differentiate extents and effective extents from other kinds of values. The expression (**extent?** *exp*) is true if the expression *exp* denotes an extent. Similarly the expression (**effective-extent?** *exp*) is true if the expression *exp* denotes an effective

```
(with-effective-extent eff-ext body) => (use (lambda ()
                                               (use-effective-extent eff-ext))
                                             (lambda () body))

(with-extent ext body) => (use (lambda () (use-extent ext))
                               (lambda () body))

(with-shadowed id body) => (use (lambda () (use-shadowed id))
                                (lambda () body))

(def use
  (lambda (install-new-eff-ext action)
    (let ((eff-ext (get-effective-extent)))
      (install-new-eff-ext)
      (let ((val (action)))
        (use-effective-extent eff-ext)
        val))))
```

Figure 11: **With-effective-extent, with-extent,** and **with-shadowed**.

extent. The third predicate (**empty-extent?** *exp*) is true if *exp* is an extent with no variable defined in it. The fourth predicate (**def?** *id*) is true if the MRE has a binding for the variable denoted by the identifier *id*.

In summary, we have introduced twelve basic FCE operations: eight procedures **get-base-extent**, **make-empty-extent**, **use-extent**, **extent?**, **empty-extent?**, **get-effective-extent**, **use-effective-extent**, and **effective-extent?**, and four special forms **def**, **def?**, **use-shadowed**, and **shadowed?**.

2.8 Embedding

FCE can be fully embedded [9] into Scheme. That is, every basic FCE expression can be defined by an equivalent Scheme expression that gives its meaning. Such an embedding shows that Scheme is as expressive as FCE. It also enables us to study the interactions between first-class extents and Scheme more directly.

The embedding is a derivation of the denotational semantics. There are two major tasks involved in constructing such an embedding. First, extents and effective extents must be made first-class values. Second, the characterization of FCE presented above implies a deep binding semantics: for each variable, its current binding is determined based on the current effective extent. Yet Scheme identifies extents with stores since it identifies variables with locations. Consequently there is *only* one extent in Scheme, which is the base extent, and

```
(def extend-object
  (lambda (own-attrs . optional-superobject)
    (cons own-attrs
      (if (null? optional-superobject) '() (car optional-superobject)))))
```

Figure 12: *Extend-object.*

a variable's current binding in Scheme is therefore always the one in the base extent. As a result the embedding's success is governed by the simulation of the deep binding semantics by a shallow binding one.

With respect to the shallow binding semantics the basic Scheme operations' meanings remain unchanged, whereas the new FCE features' interpretations require further explanation. They are described in detail in the appendix.

3 Object-Oriented Programming

In this section we demonstrate how the addition of first-class extents leads to an unusual approach to object-oriented programming. Our object system is almost a reconstruction of Object Scheme [6]. Instead of spelling out an object system in detail we simply sketch how some of the most fundamental object-oriented mechanisms can be programmed with FCE. First, we consider the characterization and representation of an object, and the construction of an object's inheritance hierarchy. Next we consider writing object-oriented code using method inheritance, method invocation, and self-reference mechanisms. Finally we compare our system with other Scheme-based object systems.

3.1 Objects

In our system, we do not distinguish between classes and instances. Rather, they are both, by convention, specialized objects. We view an object as a collection of attributes that form the state of the object. An attribute can be either a class variable, an instance variable, a method, or any other information that is relevant to an object.

We represent an attribute by an extent binding. Thus a method is a Scheme procedure bound to an ordinary Scheme variable. Our representation of an object O is a list of not necessarily distinct extents $(E_1\ E_2 \ldots E_n)$, $(n \geq 0)$. The tail of the list, namely $(E_2 \ldots E_n)$, is the superobject S of O. Attributes that O inherits from S are the collection of bindings in S. Attributes owned by O are those in the head of the list, namely E_1. They add new attributes to S in order to extend S. The extent E_1 is called the *own extent* of O.

In such a setting we can construct an object with the *extend-object* procedure defined in Figure 12. The space-delimited period preceding the formal paramter *superobject* indicates that, when *extend-object* is called, *own-attrs* matches the first actual argument and *optional-superobject* matches the list of the rest of the actual arguments. So an object without a superobject can be constructed by (*extend-object own-attrs*) where *own-attrs* is the own extent of the object. On the other hand, an object with a superobject *superobject* can be constructed by (*extend-object own-attrs superobject*).

In a system with multiple inheritance, an object can have many superobjects. Object systems like Object Scheme [6] and CLOS [11] use topological sorting algorithms to produce a total ordering of the inheritance graph. We enforce no specific multiple inheritance mechanism. Instead we require that the superobjects, represented by lists of extents, are combined into a single superobject that is in the form of a list of extents.

Knowing that an object is made up of extents we can use **def** to incrementally add attributes to an object's own extent. Thus as in CLOS, it is possible to construct an object by forming its inheritance hierarchy first and adding in attributes later. In other words, the development of an object's inheritance hierarchy can be separated from that of its attributes.

3.2 Object-Oriented Code

Here, a message is a piece of program and sending a message to an object is equivalent to evaluating the program within the receiving object's state, which is its extents. The message will always refer to the latest bindings defined in the receiving object. In this way we achieve self-reference in object inheritance [5, 12, 17] without resorting to any explicit mechanism such as the pseudo variable **self**.

We use the syntactic extension **with-object** defined in Figure 13 to pass the message *msg* to the object *obj*. Briefly, the procedure *with-exts* freezes the establishment of the extents of *obj* by forming thunks along the way. In the meantime it also reverses the order in which the extents appeared in the list so that the object's own extent is established last. Eventually the final thunk formed by *with-exts* is invoked within *init-eff-ext*. Since the definition-time effective extent of *init-eff-ext* consists of only the base extent, the effective extent of the procedure (**lambda** () *msg*) is the base extent. Thus the invocation-time effective extent of **with-object** is disabled and the message *msg* is evaluated in the extents of *obj* only. Hence a message has no access to the state of any object except that of the receiving one.

So far, we only have access to the latest bindings of an object. But we often need to access the superobject's bindings as well. **With-shadowed** thus comes into play. It provides a very general mechanism that enables a message sent to an object to reference

```
(with-object obj msg) => (letrec ((with-exts
                                  (lambda (exts thunk)
                                    (if (null? exts)
                                        (thunk)
                                        (with-exts (cdr exts)
                                          (lambda ()
                                            (with-extent (car exts)
                                              (thunk))))))))
                              (with-effective-extent
                                (with-extent (get-base-extent) init-eff-ext)
                                (with-exts obj (lambda () msg))))

(def init-eff-ext (get-effective-extent))
```

Figure 13: **With-object.**

the superceded bindings defined in the object's superobject. As a result we can use it to express the *around*, *before*, and *after* method inheritance mechanisms of CLOS.

3.3 An Example

As an example, Figure 14 shows a class, *2dcp*, of two-dimensional Cartesian points as an object. The class has four attributes. Three of them, namely *new-instance*, *distance*, and *closer?*, are methods. The fourth one, called *this-class*, is a class variable denoting the class *2dcp* itself.

The *new-instance* method illustrates how something like the *around* method inheritance mechanism might work. It also shows how the inheritance hierarchy of an object can be defined independently of the attachment of attributes to the object. The job of *new-instance* is to instantiate a Cartesian point (a, b) when given the two coordinates a and b. But the *new-instance* method defined in the *2dcp* class does not create an object by itself. Instead, using **with-shadowed**, it calls the *new-instance* method that is defined in the base extent to produce an instance of the class:

(**with-shadowed** *new-instance* (*new-instance*))

Then using **def**, it incrementally adds the coordinates to the returned object to produce the desired instance:

(**with-object** *inst*
  (**begin** (**def** *x* *init-x*) (**def** *y* *init-y*)))

```
(def 2dcp
  (let ((own-attrs
          (let ((x 'any) (y 'any))
            (make-extent
              (new-instance
                (lambda (init-x init-y)
                  (let ((inst (with-shadowed new-instance (new-instance))))
                    (with-object inst
                      (begin (def x init-x) (def y init-y)))
                    inst)))
              (distance
                (lambda () (sqrt (+ (square x) (square y)))))
              (closer?
                (lambda (pt)
                  (< (distance) (with-object pt (distance)))))))))
    (let ((obj (extend-object own-attrs)))
      (with-object obj (def this-class obj))
      obj)))
```

Figure 14: Two-dimensional Cartesian points class.

This, however, requires that the *new-instance* method be defined in the base extent as shown in Figure 15. It attaches an empty extent in front of the class object denoted by the variable *this-class*.

An immediate consequence of this is that we can define a generic function [11] to instantiate an arbitrary class, provided that the class has a properly defined *new-instance* method and a variable named *this-class* that denotes the class itself. We call this generic function *new* and define it in Figure 15. Now we can express the instantiation of a two-dimensional Cartesian point (3, 4) in a more readable style: (*new 2dcp* 3 4).

The other two class methods, namely *distance* and *closer?*, show how to accomplish self-reference without resorting to any special mechanism. Consider invoking *closer?* on the argument *pt1* within the state of some point *pt0*. It compares the distances of both points *pt0* and *pt1* from the origin in order to determine which point is closer to the origin. Once within the state of *pt0* the binding of *distance* is already established. Thus a simple variable reference is all it takes to access that method. Similarly in the body of *distance* the bindings of *x* and *y* are already in place, ensuring that the references to the coordinates will access the correct values.

```
(def new-instance
  (lambda args
    (extend-object (make-empty-extent) this-class)))
(def new
  (lambda (obj . args)
    (with-object obj (apply new-instance args))))
```

Figure 15: *New-instance* and *new*.

3.4 Other Scheme-based Object Systems

Finally we compare our object system with other Scheme-based ones. Our system is derived from Object Scheme [6]. The difference between the two systems is in the way unshadowing is accomplished. The counterpart of **with-shadowed** in Object Scheme is the **shadowed** operation. The expression (**shadowed** *id obj*) denotes the binding of *id* that is immediately shadowed by the binding of *id* in the own extent of object *obj*. There are two restrictions on the use of **shadowed**. First, (**shadowed** *id obj*) must be evaluated within the own extent of *obj*. Second, the own extent of *obj* must have a binding for *id*. It seems to us that the restrictions are too artificial just to reference a variable defined in the superobject. Besides, **with-shadowed** allows for more flexibility than **shadowed**. We can express **shadowed** in terms of **with-shadowed** as follows:

(**shadowed** *id obj*) => (**with-shadowed** *id id*)

Furthermore, suppose that O is the object (E_1 E_2 E_3) where the attribute a is defined in all three extents. One subobject (E_0 E_1 E_2 E_3) of O may inherit the a's in all three extents E_1, E_2, and E_3, while a second subobject (E_0' E_1 E_2 E_3) may inherit the a's in E_2 and E_3 only. In Object Scheme these possibilities must be programmed into the a's in the extents of O before the subobjects are constructed. This is not only cumbersome but sometimes impossible, since the object O may not know in advance how many subobjects it will have and what their behavior will be.

Next we compare our system to the other Scheme-based object systems. In our system an object is treated as a true state and a message sent to an object is a piece of program to be evaluated within the state of the receiving object. As a result when evaluating a message we can access any attribute of the receiving object by just an ordinary Scheme variable reference. This treatment of objects has several important consequences. First, it permits us to take full advantage of the simplicity, modularity, and security of lexically bound variables. Moreover, the mechanism we use to control the privacy of an object's attributes is the same one we use to control the visibility of ordinary Scheme variables. For instance in the *2dcp* class of Figure 14, the variables x and y are private within the instances of the class, only the methods defined in the class have access to them.

Another advantage of treating an object as a true state is that we can achieve self-reference by simple ordinary Scheme variable references. This enables us to write object-oriented code in the same way as we write ordinary Scheme code. For instance the piece of code $(+\ x\ y)$ computes the sum of x and y in ordinary Scheme as well as in any object.

Also, we can view Scheme as an object whose list of extents is (). In fact this is exactly what we did when we defined **with-object** in section 3.2. There we also made the assumption that the Scheme object is always the implicit least specific superobject of all other objects. This view is critical in that it allows us to say that ordinary Scheme programming is a special case of object-oriented programming where computation is always carried out within the state of the Scheme object. As a result when combined with the fact that writing object-oriented code is the same as writing ordinary Scheme code, we can say that our system is a *natural* extension of Scheme.

In other Scheme-based object systems [2, 16], an object is a dispatcher simulating an environment. But unlike an ordinary environment that associates identifiers with values, a dispatcher maps values (symbols for instance) to values. Thus an object's attributes are accessed through procedure invocations rather than by ordinary Scheme variable references. As a result the state represented by a dispatcher cannot be established so that a message can be evaluated within it. This is because a dispatcher is not a true environment.

Another major drawback of implementing an object as a dispatcher is that the resulting object-oriented code is not Scheme-like. For instance, assume that the object *obj* has two attributes x and y that are denoted by the values 'x-cor and 'y-cor, respectively. Then to compute the sum of x and y one must write (+ (**send** *obj* 'x-cor) (**send** *obj* 'y-cor)), or (+ (**send self** 'x-cor) (**send self** 'y-cor)) if the pseudo variable **self** is equated to *obj*.

Another problem with simulating an object as a dispatcher is that it cannot use lexical scoping to control the accessibility of an object's attributes. Instead it must rely on the uniqueness of a dispatcher's domain, *e.g.* symbols like 'x-attr and 'y-attr, to hide its attributes from the rest of the program.

Yet the most critical deficiency is that ordinary Scheme is not an object since it cannot be made into a dispatcher. This is another reason why the object-oriented code of [2, 16] is not Scheme-like. Clearly the object-oriented programming style induced by implementing an object as a dispatcher does not integrate well with Scheme's programming style.

4 Related Work

Consider the register abstract data type of Figure 4. Since the contents of a register is denoted by a lexical variable *contents*, we have total control over its scope — only *fetch*

```
(define make-reg (lambda (val) (make-environment (define contents val))))
(define fetch (lambda (reg) (lexical-reference reg 'contents)))
(define assign (lambda (reg val) (lexical-assignment reg 'contents val)))
```

Figure 16: Registers as first-class environments.

and *assign* can operate on it. Thus

```
(let ((contents "unspecified"))
  (with-extent reg (def contents contents)))
```

does no harm to the contents of the register *reg*, since this *contents* is a different variable from the one used by *fetch* and *assign*.

In contrast, consider the same abstract data type defined in [15] using MIT Scheme's first-class environments; see Figure 16. Because a register's contents is associated with the symbol 'contents, anyone can read and write a register. There is no security to the abstract data type. Also, renaming the identifier *contents* changes the semantics of the data type, unless the symbols in *fetch* and *assign* are changed accordingly. Furthermore, given a register *reg* as a first-class environment, its contents will be clobbered when a new definition of *contents* is added to it:

```
(eval '(define contents "unspecified") reg)
```

Clearly first-class environments lack the security and modularity of first-class extents.

There are, however, situations where first-class extents cannot replace first-class environments. They are when the system is incapable of figuring out the variable associated with an identifier. For instance, program module interfaces, the interconnections between separately compilable program units, must rely on some protocol that ultimately must be expressed at the symbolic level. The identifiers of first-class environments best serve this purpose.

Stores have also been made first-class values. The language GL [10] uses first-class stores in testing and debugging programs. With first-class stores, a location can denote multiple "versions" of values. These versions must be stored somewhere. In a multiple memory module architecture, such as SIMD, they could be in the same location of different memory modules. In a conventional architecture, they could be in different core dump files resident on disks. In any event, the mapping from locations to values needs an extra level of indirection to model the dynamic behavior of locations.

Since extents are independent of stores, multiple extents can exist in a single store. It is therefore possible to simulate first-class stores with first-class extents. The dynamic behavior of locations can be promoted to the earlier stage of variables. That is, a location that is associated with multiple versions of values can be simulated by associating the variable that denotes the location to multiple locations holding the values.

5 Conclusions

When it is not necessary we do not want our programs to depend on identifiers (variable names). To avoid the dependency we have distinguished variables from identifiers and locations. In the process we have created an intermediate level, extents, between environments and stores:

$$\text{Ide} \xrightarrow{\text{environment}} \text{Var} \xrightarrow{\text{extent}} \text{Loc} \xrightarrow{\text{store}} \text{Val}.$$

We have shown the advantages of first-class extents over first-class environments and stores, as well as their limitations. They should coexist. The interesting question is: "Which to use when?" We have presented a set of features whose behavior is essential to every language with first-class extents. The three major features discussed are an incremental definition operation, an unshadowing operation, and an effective extents mechanism. In particular we have incorporated these features into Scheme.

Based on first-class extents we have reconstructed Drescher's Object Scheme [6] in which object-oriented programming is a natural extension of Scheme's fluid binding. Object Scheme is significantly different from other Scheme object systems. In particular an object's attributes, including instance variables, class variables, and methods, are all denoted by lexical variables rather than symbols. No special mechanisms, such as **send** and **self**, are therefore needed for their definition and access.

The last issue concerns implementations. It is possible to embed the first-class extent features directly into a language. In particular we have embedded them into Scheme.

Acknowledgements

We owe much to Gary Drescher, whose Object Scheme inspired this work. We are grateful to Julia Lawall and Gary Drescher for many discussions on the design and implementation of FCE. We thank Matthias Felleisen and John Simmons for their comments on earlier drafts of this paper. The programs were typeset using the SLaTeX system provided by Dorai Sitaram.

This research was partially supported by the National Science Foundation under grants CCR 87-02117, CCR 89-01919, and CCR 90-00597.

References

[1] H. Abelson and G. J. Sussman with J. Sussman. *Structure and Interpretation of Computer Programs.* MIT, 1985.

[2] N. Adams and J. Rees. Object-oriented programming in Scheme. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 277–288, 1988.

[3] Cadence Research Systems. *Chez Scheme System Manual Revision 2.1*, September 1991.

[4] W. Clinger and J. Rees (editors). Revised^4 report on the algorithmic language Scheme. *Lisp Pointers*, 4(3):1–55, 1991.

[5] W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages and Applications*, pages 433–443, 1989.

[6] G. L. Drescher. Object Scheme: Object inheritance as fluid binding. Technical report, Thinking Machines Corporation, 1990.

[7] B. F. Duba, M. Felleisen, and D. P. Friedman. Dynamic identifiers can be neat. Technical Report 220, Computer Science Department, Indiana University, April 1987.

[8] D. Gelernter, S. Jagannathan, and T. London. Environments as first class objects. In *Proceedings of the Fourteenth ACM Symposium on Principles of Programming Languages*, pages 98–110, 1987.

[9] C. T. Haynes and D. P. Friedman. Embedding continuations in procedural objects. *ACM Transaction on Programming Languages and Systems*, 9(4):582–598, October 1987.

[10] G. F. Johnson and D. Duggan. Stores and partial continuations as first-class objects in a language and its environment. In *Proceedings of the Fifteenth ACM Symposium on Principles of Programming Languages*, pages 158–168, 1988.

[11] G. L. Steele Jr. *Common Lisp: The Language.* Digital Press, second edition, 1990.

[12] S. N. Kamin. Inheritance in Smalltalk-80: A denotational definition. In *Proceedings of the Fifteenth ACM Symposium on Principles of Programming Languages*, pages 80–87, 1988.

[13] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 151–161, 1986.

[14] J. McCarthy. Towards a mathematical science of computation. In *Proceedings of IFIP Congress 63*, pages 21–28. North-Holland, 1963.

[15] J. S. Miller and G. J. Rozas. Free variables and first-class environments. *Lisp and Symbolic Computation*, 4(2):107–141, 1991.

[16] K. Nørmark. Simulation of object-oriented concepts and mechanisms in Scheme. Technical Report R 90-01, Institute of Electronic Systems, Aalborg University, January 1990.

[17] U. S. Reddy. Objects as closures: Abstract semantics of object-oriented languages. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 289–297, 1988.

[18] J. A. Rees, N. I. Adams, and J. R. Meehan. *The T Manual.* Computer Science Department, Yale University, 1984.

[19] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT, 1981.

Appendix

This appendix consists of two sections. The first is a denotational description of FCE. The second is a Scheme interpretation of FCE in the form of an embedding.

A Formal Semantics of FCE

In this section, we provide a denotational description [7, 19] of FCE. It is presented in three parts: the abstract syntax of the basic FCE expressions, the semantic domains on which the basic expressions are defined, and the valuation function that maps the basic expressions to their meanings drawn from the semantic domains.

A.1 Abstract Syntax

$$
\begin{array}{ccll}
c & : & C & \text{(Constants)} \\
i & : & I & \text{(Identifiers)} \\
e & : & E & \text{(Expressions)}
\end{array}
$$

$$
\begin{array}{rcl}
e & ::= & c \mid (\textbf{if}\ e\ e\ e) \mid i \mid (\textbf{set!}\ i\ e) \mid (\textbf{lambda}\ (i)\ e) \mid (e\ e) \mid (\textbf{call/cc}\ e) \\
 & & \mid (\textbf{get-base-extent}) \mid (\textbf{make-empty-extent}) \mid (\textbf{use-extent}\ e) \\
 & & \mid (\textbf{get-effective-extent}) \mid (\textbf{use-effective-extent}\ e) \\
 & & \mid (\textbf{def}\ i\ e) \mid (\textbf{use-shadowed}\ i) \\
 & & \mid (\textbf{extent?}\ e) \mid (\textbf{empty-extent?}\ e) \mid (\textbf{effective-extent?}\ e) \\
 & & \mid (\textbf{def?}\ i) \mid (\textbf{shadowed?}\ i)
\end{array}
$$

Figure 17: Abstract syntax.

Figure 17 defines the abstract syntax of FCE. FCE has, in addition to Scheme's basic expressions[1], twelve new basic constructs: the procedures **get-base-extent**, **make-empty-extent**, **use-extent**, **extent?**, **empty-extent?**, **get-effective-extent**, **use-effective-extent**, and **effective-extent?**, and the special forms **def**, **def?**, **use-shadowed**, and **shadowed?**.

A.2 Semantic Domains

[1] In order to simplify the presentation, we omit multiple argument procedures.

1.	a	:	Var	(Variables)
2.	l	:	Loc	(Locations)
3.	ρ	:	$\mathrm{Env} = \mathrm{Ide} \xrightarrow{\circ} \mathrm{Var}$	(Environments)
4.	ε	:	$\mathrm{Ext} = \mathrm{Var} \xrightarrow{\circ} \mathrm{Loc}$	(Extents)
5.	θ	:	$\mathrm{Sto} = \mathrm{Loc} \xrightarrow{\circ} \mathrm{Val}$	(Stores)
6.	$\hat{l}$	:	$\mathrm{ELoc} \subset \mathrm{Loc}$	(Extent-Locations)
7.	n	:	$\mathrm{Ind} = \{0, 1, 2, \ldots\}$	(Stack Indices)
8.	τ	:	$\mathrm{Stk} = \mathrm{Ind} \xrightarrow{\circ} \mathrm{ELoc}$	(Stack)
9.	u	:	$\mathrm{Reg} = \mathrm{Ind} \times \mathrm{Ind}$	(Unshadowed Regions)
10.	β	:	$\mathrm{Bli} = \mathrm{Var} \xrightarrow{\circ} \mathrm{Reg}^*$	(Blinders)
11.	ϵ	:	$\mathrm{EExt} = \mathrm{Ind} \times \mathrm{Stk} \times \mathrm{Bli}$	(Effective Extents)
12.	κ	:	$\mathrm{Con} = \mathrm{Val} \longrightarrow \mathrm{EExt} \longrightarrow \mathrm{Sto} \longrightarrow \mathrm{Val}$	(Continuations)
13.	p	:	$\mathrm{Pro} = \mathrm{Val} \longrightarrow \mathrm{EExt} \longrightarrow \mathrm{Sto} \longrightarrow \mathrm{Con} \longrightarrow \mathrm{Val}$	(Procedures)
14.	v	:	$\mathrm{Val} = \mathrm{Ext} + \mathrm{ELoc} + \mathrm{EExt} + \cdots$	(Values)

Figure 18: Semantic domains.

To model extents and effective extents as first-class values FCE must separate the notion of extent from those of environment and store. Once we have the semantic model for FCE we must generalize the basic Scheme expressions' semantics accordingly. Then in order to show that FCE is a conservative extension of Scheme we must demonstrate that, when its added features are removed, the FCE semantic model degenerates into that of Scheme.

Equations 1–5 of Figure 18 define the semantic domains of variables, locations, environments, extents, and stores. Environments, extents, and stores are finite functions. Since extents are first-class values, they should be included in the domain of values. But when a binding is incrementally added to an extent by **def**, its effect must be shared by all the program units that have access to the extent. Hence we introduce the domain of *extent-locations*, which is a sub-domain of locations, as a level of indirection to model this sharing; see equation 6 of Figure 18. Therefore the value domain is expanded with both extents and extent-locations (cf. equation 14 of Figure 18).

We model the nesting of extents by a stack of *not* necessarily distinct extent-locations, with the extent associated with the top extent-location being the most recent extent (MRE). The stack's elements are extent-locations rather than extents. The indirection is used to model the sharing semantics mentioned above. The extent-locations are not necessarily distinct because an extent can be installed more than once. The semantic domains related to extent-location stacks are defined by equations 7–8 of Figure 18.

We define a stack to be a finite function that maps from stack indices (non-negative

$$
\begin{array}{l}
\psi : \mathrm{Var} \longrightarrow \mathrm{Ind} \longrightarrow \mathrm{Stk} \longrightarrow \mathrm{Reg}^* \longrightarrow \mathrm{Sto} \longrightarrow \mathrm{Ind} \\
\quad \psi\, a\, n\, \tau\, []\, \theta = a \in \mathit{Dom}(\theta(\tau\, n)) \;\rightarrow\; n,\; \psi\, a\, (n{-}1)\, \tau\, []\, \theta \\
\quad \psi\, a\, n\, \tau\, [(n_1, n_2), u^*]\, \theta = \\
\qquad\qquad\quad n = n_1 \;\;\rightarrow\;\; n_2{-}1 \\
\qquad\qquad \mid\; n > n_1 \;\;\rightarrow\;\; a \in \mathit{Dom}(\theta(\tau\, n)) \;\rightarrow\; n,\; \psi\, a\, (n{-}1)\, \tau\, [(n_1, n_2), u^*]\, \theta \\
\qquad\qquad \mid\; n < n_1 \;\;\rightarrow\;\; \psi\, a\, n\, \tau\, u^*\, \theta \\
\\
\phi : \mathrm{Var} \longrightarrow \mathrm{EExt} \longrightarrow \mathrm{Sto} \longrightarrow \mathrm{Ind} \\
\quad \phi\, a\, (n, \tau, \beta)\, \theta = \psi\, a\, n\, \tau\, (\beta\, a)\, \theta \\
\\
\varphi : \mathrm{Var} \longrightarrow \mathrm{EExt} \longrightarrow \mathrm{Sto} \longrightarrow \mathrm{Ind} \\
\quad \varphi\, a\, (n, \tau, \beta)\, \theta = \psi\, a\, ((\phi\, a\, (n, \tau, \beta)\, \theta){-}1)\, \tau\, (\beta\, a)\, \theta
\end{array}
$$

Figure 19: Semantic functions.

integers) to extent-locations, with the bottom of the stack being indexed by 0. The advantage of doing so will become clear later. Within such a setting the first binding found for a variable, starting from the top of the stack, is the *current binding* of the variable and the extent in which the current binding resides is the *current extent* of the variable. But this is correct only if the variable reference is not within any **use-shadowed** expression of the variable in question.

Next we model unshadowing with unshadowed regions. An *unshadowed region* of a variable is a pair of stack indices $(n,n'{+}1)$. It indicates that n is the stack-top index when the unshadowing happens and that n' is the index of the new current extent. Hence the variable's current binding is not in any of the extents whose extent-locations are indexed by $n,\ldots,n'{+}1$. Next we define a *blinder* to be a finite function from variables to sequences of unshadowed regions. A variable is not unshadowed by any **use-shadowed** expression when the sequence of unshadowed regions associated with it is empty. Otherwise it is mapped to the non-empty sequence [*head*,*tail*] where *head* is the unshadowed region encoding the most recent unshadowing and *tail* is a sequence of unshadowed regions encoding the previous unshadowings, inductively. The semantic domains of unshadowed regions and blinders are defined by equations 9–10 of Figure 18.

A stack-top index, an extent-location stack, and a blinder together form an *effective extent*. The semantic domain of effective extents is defined by equation 11 of Figure 18. Since effective extents are first-class values, the value domain must include the domain of effective extents as a summand as well (cf. equation 14 of Figure 18).

Next, with respect to the effective extents, we define two semantic functions to determine the stack indices of the current extent and the *previous* current extent of a given variable. These functions are called ϕ and φ, respectively, and are shown in Figure 19. They are defined in terms of an auxiliary function ψ. The function ψ takes five arguments: (1) a variable a whose current extent is requested, (2) a stack index n indicating the extent to look for a binding of a, (3) a stack τ of extent-locations, (4) a sequence of unshadowed regions u^* of a, and (5) a store θ. It returns the stack index of the current extent of a. According to the sequence of unshadowed regions u^*, ψ is split into two cases. If the sequence is empty, there is no more unshadowing. Therefore the extent indicated by the index n is consulted to determine if it has a binding for a. If so, n is the desired index. Otherwise, n is decremented (*i.e.*, stack is popped) and the search continues inductively. On the other hand, if the unshadowed sequence is $[(n_1, n_2),u^*]$ where (n_1, n_2) is the most recent unshadowed region, n is compared against n_1. There are three subcases. First, if $n = n_1$, $n_2{-}1$ is the desired index because (n_1, n_2) is the most recent unshadowed region and therefore $n_2{-}1$ is the index of the current extent. Second, if $n > n_1$, there are some extents installed after the most recent unshadowing and therefore they should be consulted. Third, if $n < n_1$, all extents in the region (n_1,n_2) are skipped and the search continues with the next unshadowed region.

We now have an effective extent model for FCE. We must consider how the new model interacts with continuations and procedures. Since FCE procedures and continuations do not close over the definition-time effective extent, they are passed the invocation-time effective extent. The semantic domains of continuations and procedures are defined by equations 12–13 of Figure 18.

Finally when FCE degenerates to Scheme, the new effective extent model can be reduced to a simple extent that associates each variable with exactly one location. Consequently the domain of effective extents can be absorbed by that of the stores. First, the blinder of an effective extent always maps a variable to an empty sequence of unshadowed regions since there is no **use-shadowed** expression in Scheme. We therefore can drop the blinder from the effective extent. We can also drop the extent-location stack from the effective extent since Scheme has no **use-extent** expression. Consequently we do not need the effective extent and therefore FCE is a conservative extension of Scheme.

A.3 Valuation Function

The valuation function $[\![\ ]\!]$ is defined in Figure 20. The constant and **if** clauses are straightforward and are thus omitted. Also left out are the clauses for the predicates **extent?**, **empty-extent?**, **def?**, **shadowed?**, and **effective-extent?**. In the clauses for variable reference, **set!**, **def**, and **use-shadowed**, we show only the cases when the identifier involved is bound. We also omit the case when the variable to be unshadowed by **use-shadowed** is

$$\llbracket\ \rrbracket : \mathrm{Exp} \longrightarrow \mathrm{Env} \longrightarrow \mathrm{EExt} \longrightarrow \mathrm{Sto} \longrightarrow \mathrm{Con} \longrightarrow \mathrm{Val}$$

$$\llbracket i \rrbracket\ \rho\ (n,\tau,\beta)\ \theta\ \kappa = \textbf{let}\ v = \theta(\theta(\tau(\phi\,(\rho\, i)\,(n,\tau,\beta)\,\theta))(\rho\, i))\ \textbf{in}\ \kappa\ v\ (n,\tau,\beta)\ \theta$$

$$\begin{array}{l}
\llbracket(\textbf{set!}\ i\ e)\rrbracket\ \rho\ \epsilon\ \theta\ \kappa = \\
\quad \textbf{klet}\ (v,(n,\tau,\beta),\theta) = \llbracket e \rrbracket\ \rho\ \epsilon\ \theta\ \textbf{in} \\
\quad\quad \kappa\ ?\ (n,\tau,\beta)\ \theta[\theta(\tau(\phi\,(\rho\, i)\,(n,\tau,\beta)\,\theta))(\rho\, i) \mapsto v]
\end{array}$$

$$\begin{array}{l}
\llbracket(\textbf{lambda}\ (i)\ e)\rrbracket\ \rho\ \epsilon\ \theta\ \kappa = \\
\quad \textbf{let}\ p = \lambda v \epsilon \theta \kappa\ .\ \llbracket e \rrbracket\ \rho[i \mapsto a_f]\ \epsilon\ \theta[\hat{l}_0 \mapsto (\theta\,\hat{l}_0)[a_f \mapsto l_f]][l_f \mapsto v]\ \kappa\ \textbf{in} \\
\quad\quad \kappa\ p\ \epsilon\ \theta
\end{array}$$

$$\begin{array}{l}
\llbracket(e_p\ e_v)\rrbracket\ \rho\ \epsilon\ \theta\ \kappa = \\
\quad \textbf{klet}\ (p,\epsilon,\theta) = \llbracket e_p \rrbracket\ \rho\ \epsilon\ \theta\ \textbf{in} \\
\quad\quad \textbf{klet}\ (v,\epsilon,\theta) = \llbracket e_v \rrbracket\ \rho\ \epsilon\ \theta\ \textbf{in} \\
\quad\quad\quad p\ v\ \epsilon\ \theta\ \kappa
\end{array}$$

$$\begin{array}{l}
\llbracket(\textbf{call/cc}\ e)\rrbracket\ \rho\ \epsilon\ \theta\ \kappa = \\
\quad \textbf{let}\ v = \lambda v \epsilon \theta \kappa'\ .\ \kappa\ v\ \epsilon\ \theta\ \textbf{in} \\
\quad\quad \textbf{klet}\ (p,\epsilon,\theta) = \llbracket e \rrbracket\ \rho\ \epsilon\ \theta\ \textbf{in} \\
\quad\quad\quad p\ v\ \epsilon\ \theta\ \kappa
\end{array}$$

$$\llbracket(\textbf{get-scheme-extent})\rrbracket\ \rho\ \epsilon\ \theta\ \kappa = \kappa\ \hat{l}_0\ \epsilon\ \theta$$

$$\llbracket(\textbf{make-empty-extent})\rrbracket\ \rho\ \epsilon\ \theta\ \kappa = \kappa\ \hat{l}_f\ \epsilon\ \theta[\hat{l}_f \mapsto \emptyset_\varepsilon]$$

$$\begin{array}{l}
\llbracket(\textbf{use-extent}\ e)\rrbracket\ \rho\ \epsilon\ \theta\ \kappa = \\
\quad \textbf{klet}\ (\hat{l},(n,\tau,\beta),\theta) = \llbracket e \rrbracket\ \rho\ \epsilon\ \theta\ \textbf{in}\ \kappa\ ?\ (n{+}1,\tau[n{+}1 \mapsto \hat{l}],\beta)\ \theta
\end{array}$$

$$\llbracket(\textbf{get-effective-extent})\rrbracket\ \rho\ \epsilon\ \theta\ \kappa = \kappa\ \epsilon\ \epsilon\ \theta$$

$$\llbracket(\textbf{use-effective-extent}\ e)\rrbracket\ \rho\ \epsilon\ \theta\ \kappa = \textbf{klet}\ (\epsilon,\epsilon',\theta) = \llbracket e \rrbracket\ \rho\ \epsilon\ \theta\ \textbf{in}\ \kappa\ ?\ \epsilon\ \theta$$

$$\begin{array}{l}
\llbracket(\textbf{def}\ i\ e)\rrbracket\ \rho\ \epsilon\ \theta\ \kappa = \\
\quad \textbf{klet}\ (v,(n,\tau,\beta),\theta) = \llbracket e \rrbracket\ \rho\ \epsilon\ \theta\ \textbf{in} \\
\quad\quad (\rho\, i) \in Dom(\theta(\tau\, n))\ \rightarrow \\
\quad\quad\quad \kappa\ ?\ (n,\tau,\beta)\ \theta[\theta(\tau\, n)(\rho\, i) \mapsto v], \\
\quad\quad\quad \kappa\ ?\ (n,\tau,\beta)\ \theta[\tau\, n \mapsto (\theta(\tau\, n))[\rho\, i \mapsto l_f]][l_f \mapsto v]
\end{array}$$

$$\begin{array}{l}
\llbracket(\textbf{use-shadowed}\ i)\rrbracket\ \rho\ (n,\tau,\beta)\ \theta\ \kappa = \\
\quad \kappa\ ?\ (n,\tau,\beta[\rho\, i \mapsto [(n,(\varphi\,(\rho\, i)\,(n,\tau,\beta)\,\theta){+}1),\beta(\rho\, i)]])\ \theta
\end{array}$$

Figure 20: Valuation function $\llbracket\ \rrbracket$.

not shadowing another binding.

The notation used in Figure 20 is summarized as follows. The unspecified value is denoted by ?. The arid extent is denoted by $\emptyset_e$. It binds no variables. The notation ϵ_0 denotes the initial effective extent whose configuration is $(0, \tau_0, \beta_0)$. The initial stack τ_0 of extent-locations has size one. It maps the index 0 to $\hat{l}_0$, which is the extent-location of the base extent. The initial blinder β_0 maps every variable to the empty sequence [] of unshadowed regions. The initial store θ_0 maps the extent-location $\hat{l}_0$ to the base extent. The notation $f[d \mapsto r]$ denotes the function g, an extension of the function f. $(g\,d)$ is r whereas $(g\,d')$ is $(f\,d')$ when $d' \neq d$. The subscript f of a_f, l_f, and $\hat{l}_f$ indicates that they are fresh. The notation

$$\begin{array}{l}\mathbf{klet}\ (v, \epsilon', \theta) = [\![e]\!]\ \rho\ \epsilon\ \theta\ \mathbf{in}\\ \quad E\end{array}$$

is another way of expressing

$$[\![e]\!]\ \rho\ \epsilon\ \theta\ (\lambda v\ \epsilon'\ \theta\ .\ E).$$

B Interpreting FCE

In this section we show that FCE can be fully embedded [9] into Scheme. That is, every basic FCE construct can be defined by an equivalent Scheme expression that gives meaning to the construct. We prefer implementing FCE in terms of an embedding rather than in terms of an interpreter for several reasons. First, an embedding allows us to better study the similarities as well as the differences between the two languages, especially when one is a conservative extension of the other. Second, it demonstrates the inherent power of Scheme to express new concepts and mechanisms.

The embedding is a derivation of the denotational semantics given in the last section. There are two major tasks involved in such an embedding. First, extents and effective extents must be made first-class values. Second, the semantics of FCE developed in the last section is a deep binding one: each time a variable is referenced its location is determined based on the current effective extent. Yet the location of a variable reference in Scheme is always the one in the *only* extent, which is equated to the only store. As a result the success of the embedding is governed by the simulation of the deep binding semantics by a shallow binding one.

B.1 Extents

We simulate the variable associated with the identifier *id* by the following accessor-setter pair of procedures:

```
(cons (lambda () id) (lambda (v) (set! id v)))
```

To compare two variables for equality, we employ the following *var=?* predicate:

```
(define var=?
  (let ((unique "unique"))
    (lambda (var1 var2)
      (let ((saved ((car var1))))
        ((cdr var1) unique)
        (let ((val ((car var2))))
          ((cdr var1) saved)
          (eq? val unique))))))
```

That is, the variable *var1* is temporarily assigned a uniquely identifiable value after its original value is saved. Then the value of the variable *var2* is read. If it is the uniquely identifiable value, the two variables are the same. Otherwise they are not.

We represent a binding by a *cons* cell. The *car* is a variable and the *cdr* serves as its location. An extent is simulated by a list of bindings. Thus each time the **make-empty-extent** procedure is called, it simply returns an empty list. There are two basic extent operations, namely *extend-extent* and *lookup-extent*. The *lookup-extent* operation uses *var=?* to determine if a variable is defined in an extent. Besides a variable and an extent, *lookup-extent* takes two extra procedures as success and failure continuations. If a binding is found the success continuation is called with that binding as its argument. Otherwise the failure continuation is called. The *extend-extent* operation adds a binding to an extent. This is done by a side effect so that the newly added binding is shared by all program units that have access to the extent.

Finally to achieve the effect that the base extent has a binding for every variable used in a program, we denote the base extent by a variable called *base-extent* and define the **get-base-extent** procedure as follows:

```
(set! get-base-extent (lambda () base-extent))
```

This *base-extent* variable is initially bound to an empty extent. Then whenever *base-extent* is asked for the binding of the variable *var* for the first time, we add to *base-extent* a new binding associating *var* with its current value in the store. In this way *base-extent* will always appear to have a binding for every variable.

B.2 Effective Extents

We translate the semantic functions on effective extents employed in the denotational semantics directly into Scheme code. The only difference is that extent-locations are replaced by extents directly, since the indirection furnished by extent-locations has been absorbed into the simulation of extents. Also, we use a variable called *effective-extent* to always denote the current effective extent during a computation. The effective extent held by *effective-extent* is consulted whenever the (previous) current binding of a variable is requested.

B.3 Deep Binding

In embedding FCE into Scheme we must ensure that for every variable its corresponding location in the store, which is also the base extent, always holds its current binding's value. Consequently each time a variable's current binding is changed, its value in the store must be updated accordingly. There are four basic FCE constructs that can change a variable's current binding: **def**, **use-shadowed**, **use-extent**, and **use-effective-extent**. In all cases the old current binding's value, which is in the store, is saved. Then the new current binding' value is written to the store. We name this switching process *switch-bnds*.

With respect to the shallow binding semantics, the basic Scheme expressions' meanings remain unchanged, whereas the new FCE operations' interpretations require further explanation. They are described in detail in the following.

B.4 Get-effective-extent and Use-effective-extent

We define the **get-effective-extent** procedure in Scheme as follows:

(**set! get-effective-extent** (**lambda** () *effective-extent*))

When invoked, **get-effective-extent** simply returns the value of *effective-extent*.

The **use-effective-extent** procedure is defined as follows:

(**set! use-effective-extent**
  (**lambda** (*eff-ext*)
    (*install-extent base-extent*)
    (**set!** *effective-extent eff-ext*)
    (*save-extent base-extent*)))

Operationally, the base extent is installed as the MRE by (*install-extent base-extent*). This is done by switching every new binding of *base-extent* with its corresponding old current binding of the current effective extent. As a result the current bindings of all variables save their values from the store. Next the current effective extent is set to *eff-ext*. Finally the bindings of *base-extent* are saved by (*save-extent base-extent*) so that the current bindings of *eff-ext* become the new current bindings. The *save-extent* procedure is the opposite of *install-extent*. It switches every old binding of *base-extent* with its corresponding new current binding of *eff-ext*. This practically switches the values defined in the current bindings, with respect to *eff-ext*, into the store.

B.5 Use-extent

The **use-extent** procedure is defined in Scheme as follows:

(set! use-extent
  (**lambda** (*ext*)
    (*install-extent ext*)
    (**set!** *effective-extent* <*new-effective-extent*>)))

First, (*install-extent ext*) makes the bindings of *ext* the new current bindings. Then *effective-extent* is updated to the new effective extent denoted by the expression <*new-effective-extent*> in which *ext* is the MRE.

B.6 Use-shadowed

Here is the Scheme code for interpreting the **use-shadowed** syntax when *var* is the variable to be unshadowed:

(**begin** (*switch-bnds* (*current-bnd var*) (*previous-current-bnd var*))
        (**set!** *effective-extent* <*new-effective-extent*>))

Briefly, the new current binding, the previous current binding obtained by (*previous-current-bnd var*), and the old current binding, the current binding obtained by (*current-bnd var*), switch values. Then *effective-extent* is updated accordingly.

B.7 Def

We interpret the addition of a binding associating the variable *var* with the value *val* to the MRE *ext*, whose stack index is n, as follows:

(*lookup-extent var ext*
  (**lambda** (*bnd*)
    (**if** (*eq? bnd* (*current-bnd var*))
      ((*cdr var*) *val*)
      (*set-cdr! bnd val*)))
  (**lambda** ()
    (**let** ((*new-bnd* (*cons var val*))
          (*regs* ((*cdddr effective-extent*) *var*)))
      (**when** (**or** (*null? regs*) (> *n* (*car* (*car regs*))))
        (*switch-bnds* (*current-bnd var*) *new-bnd*))
      (*extend-extent ext new-bnd*))))

When *var* is already defined in the MRE, a **def** is equivalent to a **set!**. So if the binding *bnd* in the MRE is the current binding, the new value *val* is written to the store by ((*cdr var*) *val*). Otherwise the binding *bnd* in the MRE is updated to the new value by (*set-cdr!*

bnd val). On the other hand, if *var* is not defined in the MRE, a new binding *new-bnd* must be added to *ext* by (*extend-extent ext new-bnd*). This newly added binding is the new current binding, if the MRE is not within any unshadowed region. As such, the old and new current bindings must switch values.

B.8 Predicates

To distinguish both kinds of extents from other types of Scheme values, they are each tagged with a uniquely identifiable value. The interpretations of the predicates are then straightforward and the reader is referred to the program listing at the end of this appendix.

B.9 Robustness

The shallow binding embedding described in this section is complete, but *not* robust. There is no guarantee that when an error occurs the values in the current bindings, with respect to the current effective extent, will be the same as those in the store. In fact such inconsistency is potentially ever-present during a computation. To solve this problem we need a mechanism to automatically update the location component of a current binding whenever its corresponding location in the store is updated. Unfortunately this is not possible in standard Scheme [4]. Thus the next best solution is to guarantee that whenever the computation stops, either normally or abnormally, consistency is maintained. There are many reasons a computation can be stopped abnormally. Interrupts, errors, debuggers, to name a few, can all suspend the computation of a program. Since none of them is part of standard Scheme, we cannot provide a portable solution to the problem.

Yet on a case by case basis, we can provide a solution if a sufficient hook is available. For instance in *Chez* Scheme [3] the current error handling procedure is invoked when an error is detected. With the provision of such a hook, we can maintain the desired consistency when an error occurs by extending the definition of the error handler as follows:

```
(let ((old-handler (error-handler)))
  (error-handler (lambda args
                   (use-effective-extent (get-effective-extent))
                   (apply old-handler args))))
```

That is, the original error handler is obtained by the expression (*error-handler*) and the new error handler is installed by the expression (*error-handler* <*new-error-handler*>). When the new error handler is invoked, the current effective extent is obtained and reinstalled immediately. Thus the current effective extent is not changed; instead only the current bindings are forced to save their values from the store.

B.10 Unbound Identifier References

The basic FCE constructs having to do with variable reference, namely **def**, **def?**, **use-shadowed**, and **shadowed?**, all could reference a "free" identifier. That is, they could reference an identifier that is not bound by any **lambda** or top-level **define**. In *Chez* Scheme such a reference causes an error because the identifier is considered "unbound." In the embedding, to remedy the problem, we provide the following **def?** syntax that can detect whether an identifier is bound.

```
(def? id) => (let ((old-handler (error-handler)))
                (let ((ans (call/cc
                             (lambda (return)
                               (error-handler (lambda args (return #f)))
                               id
                               #t))))
                  (error-handler old-handler)
                  ans))
```

Operationally, the continuation of the **def?** expression is obtained and saved in *return*. The error handling procedure is temporarily replaced by a procedure that returns to the program point *return* with a false value. Thus the **def?** expression is false if the identifier *id* is unbound. Otherwise the value associated with *id* is ignored and a true value is returned.

B.11 Program Listing

Below is the source listing of the complete embedding written in *Chez* Scheme. It uses four non-standard features, namely the *define-top-level-value* procedure that is used in **def** to create a global variable binding if the identifier to be bound is free, the above-mentioned error handling procedure parameter *error-handler*, the syntactic extension facility **extend-syntax**, and the uninterned symbol generator *gensym*.

```
(define get-base-extent "unspecified")
(define make-empty-extent "unspecified")
(define use-extent "unspecified")
(define extent? "unspecified")
(define empty-extent? "unspecified")
(define get-effective-extent "unspecified")
(define use-effective-extent "unspecified")
(define effective-extent? "unspecified")

(let*
  ;;; beginning of private declarations
  ((extent-tag "extent")
   (effective-extent-tag "effective-extent")

    (make-effective-extent (lambda (index stack blinder)
                              (cons effective-extent-tag
                                (cons index (cons stack blinder)))))

    (base-extent (cons extent-tag '()))

    (effective-extent (make-effective-extent 0
                        (lambda (n) base-extent)
                        (lambda (var) '())))

    (var=? (let ((unique "unique"))
             (lambda (var1 var2)
               (let ((saved ((car var1))))
                 ((cdr var1) unique)
                 (let ((val ((car var2))))
                   ((cdr var1) saved)
                   (eq? val unique))))))

    (extend-extent (lambda (ext bnd) (set-cdr! ext (cons bnd (cdr ext)))))

    (extend-stack (lambda (stack n ext)
                    (lambda (n1)
                      (if (= n1 n) ext (stack n1)))))

    (extend-blinder (lambda (blinder var reg)
                      (let ((regs (cons reg (blinder var))))
                        (lambda (var1)
                          (if (var=? var1 var) regs (blinder var1))))))
```

```
(lookup-extent (lambda (var ext sk fk)
                 (letrec ((loop (lambda (bnds)
                                  (cond
                                    ((null? bnds)
                                     (if (eq? ext base-extent)
                                         (let ((bnd (cons var ((car var)))))
                                           (extend-extent base-extent bnd)
                                           (sk bnd))
                                         (fk)))
                                    ((var=? (car (car bnds)) var) (sk (car bnds)))
                                    (else (loop (cdr bnds)))))))
                   (loop (cdr ext)))))
(lookup-index (lambda (var n stack regs)
                (letrec ((loop (lambda (n regs)
                                 (if (null? regs)
                                     (lookup-extent var (stack n)
                                       (lambda (bnd) n)
                                       (lambda () (loop (- n 1) regs)))
                                     (let ((n1 (car (car regs))))
                                       (cond
                                        ((> n n1)
                                         (lookup-extent var (stack n)
                                           (lambda (bnd) n)
                                           (lambda () (loop (- n 1) regs))))
                                        ((< n n1) (loop n (cdr regs)))
                                        (else (- (cdr (car regs)) 1))))))))
                  (loop n regs))))
(current-index (lambda (var)
                 (lookup-index var
                   (cadr effective-extent)
                   (caddr effective-extent)
                   ((cdddr effective-extent) var))))
(previous-current-index (lambda (var)
                          (lookup-index var
                            (- (current-index var) 1)
                            (caddr effective-extent)
                            ((cdddr effective-extent) var))))
```

```
(current-bnd  (lambda (var)
                (lookup-extent var
                  ((caddr effective-extent) (current-index var))
                  (lambda (bnd) bnd)
                  (lambda () #f))))

(previous-current-bnd (lambda (var)
                        (lookup-extent var
                          ((caddr effective-extent) (previous-current-index var))
                          (lambda (bnd) bnd)
                          (lambda () #f))))

(switch-bnds (lambda (old-bnd new-bnd)
               (let ((var (car old-bnd)))
                 (set-cdr! old-bnd ((car var)))
                 ((cdr var) (cdr new-bnd)))))

(install-extent (lambda (ext)
                  (for-each
                    (lambda (new-bnd)
                      (switch-bnds (current-bnd (car new-bnd)) new-bnd))
                    (cdr ext))))

(save-extent (lambda (ext)
               (for-each
                 (lambda (old-bnd)
                   (switch-bnds old-bnd (current-bnd (car old-bnd))))
                 (cdr ext))))

(use-shadowed-proc (lambda (var)
                     (switch-bnds (current-bnd var) (previous-current-bnd var))
                     (set! effective-extent
                       (make-effective-extent (cadr effective-extent)
                         (caddr effective-extent)
                         (extend-blinder (cdddr effective-extent) var
                           (cons (cadr effective-extent)
                             (+ (previous-current-index var) 1)))))))

(shadowed?-proc (lambda (var) (> (current-index var) 0)))
```

```
(def?-proc (lambda (var)
             (lookup-extent var
               ((caddr effective-extent) (cadr effective-extent))
               (lambda (bnd) #t)
               (lambda () #f))))
(def-proc (lambda (var val)
            (let ((n (cadr effective-extent)))
              (let ((ext ((caddr effective-extent) n)))
                (lookup-extent var ext
                  (lambda (bnd)
                    (if (eq? bnd (current-bnd var))
                        ((cdr var) val)
                        (set-cdr! bnd val)))
                  (lambda ()
                    (let ((new-bnd (cons var val))
                          (regs ((cdddr effective-extent) var)))
                      (when (or (null? regs) (> n (car (car regs))))
                        (switch-bnds (current-bnd var) new-bnd))
                      (extend-extent ext new-bnd))))))))
;;; end of private declarations
(set! get-base-extent (lambda () base-extent))
(set! make-empty-extent (lambda () (cons extent-tag '())))
(set! get-effective-extent (lambda () effective-extent))
(set! use-extent (lambda (ext)
                   (unless (extent? ext) (error 'FCE "Not an extent: ~s" ext))
                   (install-extent ext)
                   (set! effective-extent
                     (let ((n (+ (cadr effective-extent) 1)))
                       (make-effective-extent n
                         (extend-stack (caddr effective-extent) n ext)
                         (cdddr effective-extent))))))
(set! use-effective-extent (lambda (eff-ext)
                             (install-extent base-extent)
                             (set! effective-extent eff-ext)
                             (save-extent base-extent)))
```

```
    (set! extent? (lambda (obj) (and (pair? obj) (eq? (car obj) extent-tag))))
    (set! empty-extent? (lambda (obj)
                          (and (not (eq? obj base-extent))
                               (extent? obj)
                               (null? (cdr obj)))))
    (set! effective-extent? (lambda (obj)
                              (and (pair? obj) (eq? (car obj) effective-extent-tag))))
    (extend-syntax (id→var)
      ((id→var id) (with ((val (gensym)))
                     (cons (lambda () id) (lambda (val) (set! id val))))))
    (extend-syntax (def)
      ((def id exp) (with ((def-proc def-proc) (val (gensym)))
                      (let ((val exp))
                        (unless (defined? id) (define-top-level-value 'id "unspecified"))
                        (def-proc (id→var id) val)
                        "unspecified"))))
    (extend-syntax (def?)
      ((def? id) (with ((def?-proc def?-proc))
                   (and (defined? id) (def?-proc (id→var id))))))
    (extend-syntax (use-shadowed)
      ((use-shadowed id) (with ((use-shadowed-proc use-shadowed-proc))
                           (if (shadowed? id)
                               (use-shadowed-proc (id→var id))
                               (error 'FCE "Not shadowed: ~a" 'id)))))
    (extend-syntax (shadowed?)
      ((shadowed? id) (with ((shadowed?-proc shadowed?-proc))
                        (and (defined? id) (shadowed?-proc (id→var id)))))))
  (let ((old-handler (error-handler)))
    (error-handler (lambda args
                     (use-effective-extent (get-effective-extent))
                     (apply old-handler args))))
```

```
(let ((bound (lambda (accessor)
               (let ((old-handler (error-handler)))
                 (let ((ans (call/cc (lambda (return)
                                       (error-handler (lambda x (return #f)))
                                       (accessor)
                                       #t))))
                   (error-handler old-handler)
                   ans)))))
  (extend-syntax (defined?)
    ((defined? id) (with ((bound bound))
                     (bound (lambda () id))))))
```