LISP AND SYMBOLIC COMPUTATION: An International *Journal*, 5, 343—375, *1993*

First-Class Extents*

SHINN-DER LEE *(sdlee@cs.indiana.edu)*

DANIEL P. FRIEDMAN *(dfried@cs.indiana.edu)*

Computer Science Department
Indiana University
Bloomington, IN 47405, USA

Keywords: Identifier, Variable, Location, Scope, Extent, State

Abstract. Adding environments as first-class entities to a language can greatly enhance its expressiveness. But first-class environments rely on identifiers, the syntax of variables, and thus do not mesh well with lexically-scoped languages. We present first-class extents as an alternative. First-class extents are founded upon lexical variables with dynamic extent. They are defined directly on the variables themselves rather than on their syntax. They therefore do not cause variable name capturing problems that plague first-class environments. Moreover, distinguishing variables from locations allows first-class extents to be orthogonal to imperative and control features.

1. Introduction

Environments are maps from identifiers, variable names, to locations that hold the values of the variables. They can be used to model various programming mechanisms and concepts such as closures, structures, abstract data types, classes, and inheritance. Adding environments as first-class entities to a language therefore can greatly enhance its expressiveness, as exemplified by MIT Scheme [1, 13], Symmetric Lisp [7], Rascal [9], and reflective languages such as 3-Lisp [15], I_R [10], and Refci [14].

First-class environments, however, can easily lead to context anomalies due to inadvertent variable name capturing. In particular, they do not go well with lexical scoping where variable renaming is a crucial meaning-preserving program transformation. Indeed, lexical variables are nameless in nature, which is why they can be renamed. Such environments also cause problems for macros and optimization techniques such as unfolding. In fact, any program transformation that involves the renaming, introduction, or elimination of variables could potentially alter the domains of environment maps and therefore would interfere with first-class environments.

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

First-class environments are founded on identifiers, the *syntax* of variables. They are the source of their expressiveness, but are also the cause of their context anomalies. Our alternative is to trade some flexibility of first-class environments for the security and modularity of lexical variables. We opt to rely on the variables themselves rather than rely on their names. That is, our first-class entities, first-class *extents,* are maps whose domains are lexical variables instead of identifiers. The define-use dependency of lexical variables is a static property. Renaming, introduction, and elimination of lexical variables thus do not cause inadvertent variable capturing problems.

First-class extents allow a variable to be defined in more than one extent, which means that a variable can denote multiple values. The traditional view of equating a variable with a location, which is the *implementation* of the variable, is inappropriate, because there would then be a one-to-many relation between locations and values. We are therefore prompted to distinguish variables from locations. Thus, a variable can denote multiple locations, each holding one of the several values of the variable. The distinction between variables and locations also makes first-class extents orthogonal to side effects. We can therefore treat the two mechanisms independently.

With the distinctions among identifiers, variables, and locations, there are three stages in determining the value of an identifier. They are modeled by the following three maps *p, e,* and *B:*

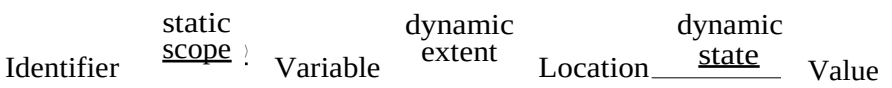
Syntax of Variable —p→ Variable —E→ Implementation of Variable —B→ Contents of Variable

The map p translates an identifier into the variable it denotes. Next, the map E transforms the variable into a location. Then, the map 0 retrieves the contents of the location, which is the value of the variable. Thus, the composition of e and 0 defines the denotation of a variable. The map p serves primarily to help visualize the variable; it should not play a dominant role in determining its semantics.

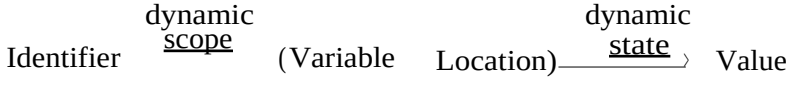
The map B is traditionally called *a store.* If a semantic entity always uses the active store at the point of its definition to determine the contents of its locations, it is said to have a *static* store. A semantic entity with a static store is not sensitive to side effects on its locations by other semantic entities. Conversely, a semantic entity that uses the active store at the point of its invocation to determine the contents of its locations is said to have a *dynamic* store. An entity with dynamic store is sensitive to side effects. Thus, stores model the *state* of locations. Hence we also call them

state maps. The map ρ is usually called an *environment.* Analogous to states, a semantic entity can be associated with either a static environment or a dynamic environment. With a static environment, an identifier in a semantic entity always denotes the same variable. With a dynamic environment, however, an identifier in an entity can denote different variables depending on the different active environments at the points the entity is invoked. Thus, environments model the *scope* of identifiers. Hence, we also call them scope maps. The new map e is called an *extent.* It models the duration during which its locations are accessible. Again, analogous to states and scopes, a semantic entity can be associated with either a static extent or a dynamic extent. With a static extent, a variable is always associated with the same location. With a dynamic extent, however, a variable can denote different locations in the different active extents at the points the entity is invoked.

So, our first-class extents can be characterized as having dynamic extent and state maps, but the scope map is static:



That is, they are based on statically-scoped variables with dynamic extent. They enjoy the security and modularity of static scoping and allow dynamic association of variables with different locations. On the other hand, first-class environments have dynamic scope and state maps:



They allow identifiers to be bound to different variables dynamically. There is no extent map here since variables are equated with locations.

In the next three sections we add first-class extents to Scheme [2]. Scheme is our choice as the base language to demonstrate the expressiveness of first-class extents because it is a lexically-scoped language with imperative features. In addition, it is small, which simplifies our presentation. In Section 5 we give a formal description of the extension, in the form of a denotational semantics. Then, based on the formal semantics, we relate extents to states. In particular, we demonstrate that first-class extents can be simulated with assignment and shallow binding. On the other hand, we also show that extents can provide an alternative characterization of side effects. Section 6 concludes the work. A detailed description of the simulation of extents, including a complete source listing, can be found in the appendix.

2. Simple Extents

To turn extents into a useful programming mechanism, there must be adequate user-level operations. We start with two of them: **make-extent** and **with-extent.** The first creates new extents and the second evaluates an expression within the effect of a given extent. To be flexible, the second operation can be nested so that an expression can be evaluated within the effect of multiple nested extents.

The expression (**make-extent** *(id exp) ...)* evaluates to a new extent in which each pair *(id exp)* associates the lexical variable denoted by the identifier *id* with a fresh location, which contains the value of the expression *exp.* We call the extents created by **make-extent** *simple* in order to distinguish them from the *composite* extents to be defined in the next section. The expression **(with-extent** *ext exp)* evaluates the expression *exp* with the simple extent denoted by the expression *ext* in effect. It returns the result of the evaluation of *exp.* For instance, the following **with-extent** expression evaluates to the list (1 44), not (33 44) or (22 44).

```
(let ((x 1) (y 2))
   (let ((foo (let ((x 22))
                (make-extent (x 33) (y 44)))))
      (with-extent foo (list x y))))
```

Since variables are lexically scoped, *foo* is equivalent to

```
(let ((z 22))
   (make-extent (z 33) (y 44)))
```

where x is renamed to z. The expression *(list x y)* thus refers to the *y* defined in *foo,* but not the *x.* Instead, it refers to the outer *x.* Indeed, due to the stringent scoping constraint on lexical variables, the location of x defined in *foo* is inaccessible.

An expression must be evaluated within the effect of an extent that defines all the variables referenced in that expression. For instance, in the evaluation of the above **with-extent** expression, the variable *foo* must be defined in an extent that is currently in effect. To make the extension of Scheme with the first-class extent features transparent, we assume that every program is evaluated within the effect of a *base* (simple) extent that has a definition for every variable (either global or local) needed by the program. Thus a pure Scheme program, one that uses none of the first-class extent features, is evaluated only with respect to the base extent.

The next example (Figure 1) defines an abstract data type of registers. It takes full advantage of lexical scoping to hide the implementation of

```
(define make-reg '*)
(define fetch '*)
(define assign '*)

(let ((contents '*))
   (set! make-reg
      (lambda (val)
         (make-extent (contents val))))
   (set! fetch
      (lambda (reg)
         (with-extent reg contents)))
   (set! assign
      (lambda (reg val)
         (with-extent reg (set! contents val)))) )
```

Figure 1: Registers — Extent Implementation.

```
(define make-reg
   (lambda (contents)
      (lambda (msg)
         (case msg
            ((fetch) (lambda () contents))
            ((assign) (lambda (val) (set! contents val)))
            (else (error 'register "Unknown message ~s" msg))))) )

(define fetch
   (lambda (reg)
      ((reg 'fetch))))

(define assign
   (lambda (reg val)
      ((reg 'assign) val)))
```

Figure 2: Registers — Closure Implementation.

```
(define rO '*)
(define rl '*)
(define r2 '*)
(define r3'*)
(define continue '*)

(define registers
  (make-extent
    (rO (make-reg 0))
    (rl (make-reg 0) )
    (r2 (make-reg 0))
    (r3 (make-reg 0))
    (continue (make-reg'()))))
```

Figure 3: Machine Registers.

registers. The three operators are *make-reg, fetch,* and *assign. Make-reg* creates registers, *fetch* reads from registers, and *assign* writes to registers. Each register created by *make-reg is* a new simple extent that associates a new location with the lexical variable *contents.* Since *contents is* accessible only to the other two operators, it is free from any inadvertent access. Such a degree of security would not be available if the simple extents were founded on the identifier *contents,* reminiscent of first-class environments. Moreover, renaming *contents* would not preserve the behavior of registers. The security of *contents* can also be achieved using closures. Figure 2 shows a typical closure implementation. The extent implementation is intuitively more appealing because the same pair of operators *fetch* and *assign* can apply to all registers directly without resorting to message passing. There is no need for a dispatch (i.e., case).

Simple extents can be nested. Nested simple extents are expressed with nested **with-extent** expressions. For each variable, the location defined in an inner simple extent shadows the locations in the outer simple extents. The innermost location, the one that shadows all the others, is the variable's *effective location.* It is the location to which references and assignments of that variable refer. Operationally, nested simple extents form a stack with the base extent at the bottom; locations defined in simple extents closer to the top shadow those in simple extents closer to the bottom. The effective location of a variable is the location that is closest to the top of the stack.

As an example we develop a simple register machine similar to that described by Miller and Rozas [131. The machine has five registers: four general purpose registers *rO, r1, r2, r3,* and *a continue* register recording

```
(define goto '*)
(define branch '*)
(define call '*)
(define return '*)

(define instructions
  (make-extent
    (fetch fetch)
    (assign assign)
    (goto (lambda (label) (label)))
    (branch (lambda (test then-label else-label)
              (goto (if test then-label else-label))))
    (call (lambda (entry-label return-label)
            (assign continue
              (cons return-label (fetch continue)) )
            (goto entry-label)))
    (return (lambda ()
              (let ((return-label (car (fetch continue))) )
                (assign continue (cdr (fetch continue)) )
                (goto return-label))))))
```

Figure 4: Machine Instructions.

the return points of subroutine calls. The registers form a simple extent *registers* as shown in Figure 3. In addition to the standard arithmetic and logic operators, there are six instructions used by the machine: the two register operators *fetch* and *assign,* the unconditional *goto,* the conditional *branch,* the subroutine call instruction *call,* and the subroutine return instruction *return.* The instructions also form a simple extent *instructions;* see Figure 4.

A routine of the machine is also a simple extent. Each of its variables is associated with a label, in the form of a thunk, that is the entry point of a non-empty sequence of instructions. By invoking such a label, control is transferred to the first instruction in the sequence. In order to ensure that routines are in iterative form [12], each sequence of instructions consists of a number of *assign* and *fetch* instructions followed by one of the other four control transferring instructions: *goto, branch, call,* or *return.* For instance, Figure 5 shows a routine *gcd2* that computes the greatest common divisor (GCD) of the two numbers found in registers *rO* and *rl.* It returns the answer through register *rO. So,* to compute the greatest common divisor of two numbers, the routine is first set up by nesting the three simple extents

```
(define gcd2 '*)

(define gcd2
   (let ((next '*) (done '*))
      (make-extent
         (gcd2 (lambda ()
                  (branch (zero? (fetch rl)) done next)) )
         (next (lambda ( )
                  (assign r3 (remainder (fetch rO) (fetch rl)) )
                  (assign rO (fetch rl) )
                  (assign r1 (fetch r3) )
                  (goto gcd2)))
         (done (lambda ()
                  (return))))))
```

Figure 5: GCD Subroutine Of Two Numbers.

```
(define gcd3 '*)

(define gcd3
   (let ((next '*) (done '*))
      (make-extent
         (gcd3 (lambda ()
                  (call gcd2 next)))
         (next (lambda ()
                  (assign rl (fetch r2) )
                  (call gcd2 done)) )
         (done (lambda ()
                  (return))))))
```

Figure 6: GCD Subroutine Of Three Numbers.

instructions, registers, and *gcd2.* Next, the two numbers are loaded into registers *rO* and *r1* and then a subroutine call to the label *gcd2 is* issued with a returning label that reads the answer out of register *r0:*

```
(define gcd2-proc
  (lambda (a b)
    (with-extent instructions
      (with-extent registers
        (with-extent gcd2
          (begin
            (assign rO a)
            (assign r1 b)
            (call gcd2 (lambda () (fetch rO)))))))))
```

Continuing the example, we define on top of *gcd2 a* subroutine *gcd3* that computes the greatest common divisor of the three numbers in registers *rO, rl,* and *r2;* see Figure 6. Its procedural abstraction is

```
(define gcd3-proc
  (lambda (a b c)
    (with-extent instructions
      (with-extent registers
        (with-extent gcd2
          (with-extent gcd3
            (begin
              (assign rO a)
              (assign rl b)
              (assign r2 c)
              (call gcd3 (lambda () (fetch r0)))))))))))
```

The same identifiers *next* and *done* are used in both subroutines *gcd2* and *gcd3.* But because they are local lexical variables, they associate different labels in different subroutines and are therefore inaccessible to the other. There is thus no accidental entry due to inadvertent name capturing, which would happen if the labels were based on identifiers, again, reminiscent of first-class environments.

3. Composite Extents

In the previous section we defined a mechanism to build and use simple extents. In this section we generalize it to composite extents. *A composite* extent is a sequence of nested simple extents.

During the evaluation of an expression, the nested simple extents in effect, including the base extent, constitute the *effective extent* at that point.

It completely determines the effective locations of every free variable reference in the expression. In the following we provide a mechanism to reify effective extents into first-class entities as the means of constructing composite extents. With first-class effective extents available, we can specify the effective extent in which semantic entities like procedures and continuations are evaluated. Thus, we can define procedures and continuations with static variable-location bindings, as well as with dynamic variable-location bindings.

We introduce three operations on effective extents. The first two are *get-extent* and **with-extent.** Invoking the zero argument procedure *get-extent* during a computation returns the current effective extent, *excluding* the base extent, as a reified composite extent value. The exclusion of the base extent allows for the "compositionality" of the reified extents, which we will demonstrate later in this section. The expression *(***with-extent** *ext exp)* evaluates the expression *ext* to a composite extent (a reified effective extent); nests its simple extents within the effective extent of the with-extent expression; evaluates the expression *exp* within the new effective extent to a value *v;* and then returns *v* after reinstalling the original effective extent. Simply put, we have generalized **with-extent** to work on sequences of nested simple extents rather than individual simple extents. To simplify the system, we also generalize **make-extent** accordingly to construct composite extents that happen to consist *only* of a single simple extent. With these generalizations, every extent is considered to be a composite extent from now on unless stated otherwise.

For instance, the following is another way of defining a procedural abstraction of the register machine running the *gcd2* routine presented at the end of the previous section.

```
(define gcd2-machine
  (with-extent instructions
    (with-extent registers
      (with-extent gcd2
        (get-extent)))))

(define gcd2-proc
  (lambda (a b)
    (with-extent gcd2-machine
      (begin
        (assign rO a)
        (assign r1 b)
        (call gcd2 (lambda () (fetch r0)))))))
```

The invocation of *get-extent* within the three required extents returns the

composition of the three extents as the composite extent *gcd2-machine.* This composite extent is then used in the procedure *gcd2-proc.*

Like simple extents, composite extents returned by *get-extent* can be composed, through nesting, to form new composite extents. For instance, a machine running the *gcd3* routine (cf. Section 2) can be defined as a composite extent as follows:

```
(define gcd3-machine
  (with-extent gcd2-machine
    (with-extent gcd3
      (get-extent))))
```

The extent *gcd3* is established within *gcd2-machine* and the composition is obtained by a call to *get-extent.* The same behavior can also be obtained by the following definition:

```
(define gcd3-machine
  (with-extent gcd3
    (with-extent gcd2-machine
      (get-extent))))
```

That is, the nesting orders have no effect on the outcome because the extents *gcd2-machine* and ***gcd3*** are mutually exclusive with respect to their defined variables. This example demonstrates why the reified effective extent obtained by *get-extent, gcd2-machine* in this case, does not include the base extent. If it did, since the base extent has a location for every variable used in a program, its locations would shadow those of ***gcd3*** and therefore the alternate definition of *gcd3-machine* would behave the same as *gcd2-machine.*

The third operation on effective extents provides a means to "escape" the effect of the effective extent. The expression **(abort-extent** *exp)* temporarily disables the effect of the effective extent of the expression; evaluates the expression *exp* only within the base extent to some value *v;* reinstalls the disabled effective extent; and then returns v as the result. For instance, the expression

```
(with-extent gcd3
  (abort-extent
    (with-extent gcd2-machine
      (get-extent))))
```

evaluates to a composite extent that is equivalent to *gcd2-machine,* not *gcd3-machine.* The effect of the extent *gcd3* is temporarily ignored by the

```
(extend-syntax (lambda/composable-extent)
  ((lambda/composable-extent args exp)
   (with ((ext (gensym)))
     (let ((ext (get-extent)))
       (lambda args
         (with-extent ext exp))))))

(extend-syntax (lambda/non-composable-extent)
  ((lambda/non-composable-extent args exp)
   (with ((ext (gensym)))
     (let ((ext (get-extent)))
       (lambda args
         (abort-extent
           (with-extent ext exp)))))))
```

Figure 7: Procedures With Static Extents.

```
(define call/cc/composable-extent
  (lambda (f)
    (let ((ext (get-extent)))
      (call/cc
        (lambda (k)
          (f (lambda (v)
               (with-extent ext (k v)))))))))

(define call/cc/non-composable-extent
  (lambda (f)
    (let ((ext (get-extent)))
      (call/cc
        (lambda (k)
          (f (lambda (v)
               (abort-extent
                 (with-extent ext (k v)))))))))
```

Figure 8: Continuations With Static Extents.

abort-extent operation. Thus only *gcd2-machine is* in effect during the invocation of *get-extent.*

The effect of the extent *ext is fluid* with respect to the expression *exp* in the evaluation of (**with-extent** *ext exp).* That is, *ext is* in effect only during the evaluation of *exp. As* soon as *exp* returns a value *v,* the effect of *ext is* removed. Free variables in *v* thus no longer refer to the locations defined in *ext.* For instance, the expression

```
(let ((x 1))
  (let ((bar (with-extent (make-extent (x 3))
                          (lambda () x))))
    (bar)))
```

evaluates to 1, not 3. Once the with-extent expression returns the procedure **(lambda () *x),*** the location of *x* whose value is 3 is no longer in effect. Thus, when *bar is* called, it refers to the location whose value is 1.

With the effective extent mechanism being orthogonal to procedures and continuations, we can define procedures and continuations with static extents, i.e., with static variable-location bindings. Figure 7 shows two ways to define procedures with static extents.[1] The **lambda/composable-extent** form constructs a procedure that captures the composable effective extent *ext* when it is defined. This captured effective extent is established on top of the effective extent when the procedure is invoked. That is, the effective extent during the evaluation of the procedure's body expression is the composition of *ext* and the procedure's invocation-time effective extent. The **lambda/non-composable-extent** form is similar except that it ignores the procedure's invocation-time effective extent. In other words, it is equivalent to capturing a non-composable definition-time effective extent. Similarly, we can define two kinds of continuations with composable and non-composable static extents. They are shown in Figure 8 without comment.

Continuing the register machine example of Section 2 we implement a break point mechanism to keep track of the number of subroutine calls occurring in the execution of a program. Figure 9 shows the needed facility. The *call* instruction is extended with a *call/cc/non-composable-extent* jump to a debugger before transferring control to the entry label of the called subroutine. So, each time a subroutine is called, the machine transfers control to the debugger. The procedural abstraction *gcd2-proc is* then defined with **lambda/non-composable-extent** to capture the composition of the extent *gcd2-machine* and the *debugger-call-instruction* extent that defines the extended *call* instruction. The debugger is a procedure

[1] Here we use the extend-syntax syntactic extension system of *Chez* Scheme [6].

```
(define debugger-call-instruction
  (make-extent
    (call (lambda (entry-label return-label)
            (call/cc/non-composable-extent debugger)
            ((with-extent gcd2-machine call)
             entry-label return-label)))) )

(define gcd2-proc
  (with-extent gcd2-machine
    (with-extent debugger-call-instruction
      (lambda/non-composable-extent (a b)
        (begin
          (assign rO a)
          (assign rl b)
          (call gcd2 (lambda () (fetch r0))))))))

(define debugger
  (with-extent instructions
    (with-extent debugger-registers
      (lambda/non-composable-extent (break-point)
        (begin
          (assign rO (+ (fetch rO) 1))
          ( break-point "unspecified"))))) )
```

Figure 9: Break Point Facility.

with a non-composable static extent consisting of the machine instructions and a separate set of registers *debugger-registers.* It takes a break point, in the form of a continuation, increments its own version of the *rO* register, and resumes the break point.

4. Unshadowing Locations

Nesting of extents induces a shadowing semantics on locations. When a variable is defined in nested extents, its location defined in the inner (more recent) extent shadows the locations in the outer (less recent) extents. The outer locations are therefore inaccessible. Such a constraint limits the flexibility of nesting as a general means of composing extents. For instance, let variables x and y be defined in both extents *foo* and *bar.* Then there is no way to compose the two extents so that the effective locations of x and y are those in *foo* and *bar,* respectively. Neither order of nesting the extents

works. We need a mechanism to reverse the effect of shadowing. Moreover, the way *call is* extended in the previous section (cf. Figure 9) requires that the to-be-extended *call* instruction, the one in *gcd2-machine,* be identified explicitly. The extension thus does not apply to any other extent that also defines *call.*

The dual of shadowing is an operation that undoes the effect of some **with-extent** on a variable. It makes a variable's most recently shadowed effective location its new effective location. We call such an operation *unshadowing* and define its syntax as (**with-shadowed** *id exp).* During the evaluation of the expression *exp,* the most recently shadowed effective location of *id* temporarily becomes the effective location. Inductively, when two unshadowing operations of the same variable are nested, the *next* most recently shadowed effective location becomes the new effective location, and so forth. Given this semantics, it is an error when there is no shadowed location. Hence we also provide a predicate to decide whether that is the case. The expression **(shadowed?** *id) is* true if the effective location of the variable denoted by the identifier *id* shadows another location of *id;* it is false otherwise.

Using **with-shadowed,** we can devise the following solution to combine the *foo* and *bar* extents mentioned above:

```
(with-extent foo
  (with-extent bar
    (with-shadowed x
      (get-extent))))
```

In the resulting extent, the effective location of y is the location in *bar. As* for *x,* its effective location is the location in *foo,* since the location in *bar* has been unshadowed by the **with-shadowed** operation. We can accomplish the extension of the *call* instruction in Figure 9 using **with-shadowed** as follows:

```
(define debugger-call-instruction
  (make-extent
    (call (lambda (entry-label return-label)
            (call/cc/non-composable-extent debugger)
            (with-shadowed call
              (call entry-label return-label))))))
```

The unshadowing uncovers whatever *call* instruction that is shadowed by the one defined in the *debugger-call-instruction* extent. This solution is more appealing because *debugger-call-instruction is* defined without the knowledge of *gcd2-machine* and therefore can be used to extend the *call* instruction of any extent.

The use of unshadowing to extend the behavior of a procedure like *call* is reminiscent of a method extension in ob ject-oriented programming. Indeed, based on first-class extents, we can build an object-oriented programming paradigm on top of a lexically-scoped language. An extent is thought of as an object. Its variables are the object's methods and instance variables. Extent nesting is object inheritance and **with-shadowed** corresponds to the *super* pseudo-variable. Evaluating an expression within the effect of an extent is equivalent to sending a message to an object. The message will always refer to the object's most specific method and instance variable definitions because of the shadowing semantics of nested extents. Thus self-reference in object inheritance is achieved without resorting to any explicit mechanism like the *self* pseudo-variable. Drescher's Object Scheme [3], which we reconstructed with first-class extents [11], is such a Scheme-based object system.

5. Interpreting First-Class Extents

In this section we provide a denotational description [16] of the first-class extent features introduced in the last three sections. In addition, using shallow binding, we show that these features can be interpreted with assignment.

Before presenting the formal semantics, we replace the non-tail-recursive **with-extent, with-shadowed,** and **abort-extent** with their tail-recursive counterparts: *use-extent,* **use-shadowed,** and *remove-extent.* The expression *(use-extent ext)* installs the composite extent denoted by the expression *ext* on top of the current effective extent in the subsequent computation. Similarly, the expression **(use-shadowed** *id)* updates the effective extent so that the most recently shadowed location of *id* becomes the new effective location in the subsequent computation. The expression *(remove-extent)* replaces the effective extent with the base extent only.

The non-tail-recursive operations can be defined as syntactic extensions in terms of their tail-recursive counterparts (Figure 10). For **with-extent,** the effective extent is saved in *old-eff.* It is then extended with the given composite extent *ext.* Next, the body expression *exp is* evaluated within the new effective extent to obtain the result *val.* Then the saved extent *old-eff* is reinstalled as the effective extent before the value *val* is returned as the result of the entire **with-extent** expression. The other two operations are defined similarly.

So, the core first-class extent constructs are the procedures *get-extent, remove-extent,* and *use-extent,* and the special forms **make-extent, use-shadowed,** and **shadowed?.** Figure 11 defines the abstract syntax of the language whose formal semantics is given in this section. In order to

```
(define non-tr
  (lambda (setup-new-eff body)
    (let ((old-eff (get-extent)))
      (setup-new-eff )
      (let ((val (body)))
        (remove-extent)
        (use-extent old-eff )
        val)))))

(extend-syntax (with-extent)
  ((with-extent ext exp)
   (non-tr
     (lambda () (use-extent ext))
     (lambda () exp))))

(extend-syntax (with-shadowed)
  ((with-shadowed id exp)
   (non-tr
     (lambda () (use-shadowed id))
     (lambda () exp))))

(extend-syntax (abort-extent)
  ((abort-extent exp)
   (non-tr
     (lambda () (remove-extent))
     (lambda O exp))))
```

Figure 10: Non-Tail-Recursive Extent Operations.

```
i  E Ide (Identifier)
e  E  Exp (Expression)

e  ::= i | (set! i e) | (lambda (i) e) | (e e) | (call/cc e)
       | (make-extent i e) | (get-extent)
       (use-extent e) | (remove-extent)
       (use-shadowed i) | (shadowed? i)
```

Figure 11: Abstract Syntax.

a	E	Var	(Variable)
1	E	Loc	(Location)
p	E	Env = Ide Var	(Environment)
E	E	SExt = Var --e4 Loc	(Simple Extent)
8	E	Sta = Loc -.94 Val	(State)
n	E	Nat = {0,1,2,...}	(Natural Number)
U	E	Uns=Var —*Nat	(Unshadowing)
E	E	CExt = (SExt x Uns)*	(Composite Extent)
	E	Con=Val-*CExt ~Sta-*Val	(Continuation)
p	E	Pro = Val —> CExt —> Sta -4 Con —~ Val	(Procedure)
v	E	Val = CExt + **{true, false}** +{?}+•••	(Value)

Figure 12: Semantic Domains.

simplify the presentation, only constructs whose semantics are relevant to extents are included. They are variable references, assignments, procedures, procedure invocations, and first-class continuations.[2]

5.1. Formal Semantics

Without unshadowing, a composite extent would merely be a sequence of nested simple extents. With unshadowing, however, a composite extent must also keep track of the unshadowings of every variable. Moreover, when two composite extents are composed, their unshadowings must also be composed coherently. The detailed formal description of the first-class extent features is the subject of this section.

Figure 12 defines the necessary semantic domains. A simple extent E is a finite function $\{a_1 \quad {}^{l}1i \ldots, an\ H\ I_n\}$, where $n > 0$, that associates variable a_i with location l_i. Without unshadowing, a composite extent E would simply be a sequence of concatenated simple extents $E_n \circledR \cdots \circledR El$, where $n > 0$, with E_n being the most recently established simple extent. The inclusion of unshadowing, however, requires a composite extent to acquire an additional map u to record for each variable the number of unshadowings in effect. Such a map u is thus called an unshadowing map. Since all composite extents are initially created by **make-extent,** they start out as a pair (E, u) of a simple extent e and an unshadowing map u. Later, their compositions can be obtained by *get-extent.* Thus, a composite extent cis a sequence of (E, u)-pairs: $(en, un)\ e \cdot \cdot \circledR(el, u_1)$. Each u_i serves as the unshadowing map of the sub-extent $(E_i, u_i) \quad \cdots \circledR(El, u_1)$. It associates with

[2] For simplicity, the semantics only describes single argument procedures and single variable **make-extent.**

```
: Var —* CExt — (Loc + {undefined})
  Oac = (,oa0E

cp : Var —* Nat —* CExt --). (Loc + {undefined})
  cpan((E,u)®c) = ba(n+(ua))EE
  (jcan() = undefined

: Var —* Nat —* SExt — CExt —* (Loc + {undefined})
  OaOEE = a E Dom(E) --> Ea,cpa0c
  7|' a(n+ 1)E€ = a E Dom(E) -~ cpanc, cpa(n+ 1)€
```

Figure 13: Effective Location Lookup.

each variable the number of unshadowing operations issued while E_i is the most recently established simple extent. With this setting, the composition of composite extents q and E_2 is simply their concatenation $e_1 \circledR E2$. An effective extent is a composite extent whose least recently established simple extent is the base extent. That is, if $c \circledR (E, u)$ is an effective extent, E is the base extent.

The effective location of a variable a in a composite extent $(E_n, u_n) \circledR (E1, u_1)$ is defined by the function shown in Figure 13. Intuitively, the variable's number of unshadowings is kept in an accumulator n that is initially set to zero. Then before each simple extent E_i is examined for a definition of the variable, the number of unshadowings recorded in u_i is added to the accumulator. If the variable is not defined in the simple extent, the process is repeated for the sub-extent $(E_{i-1}, u_{i-1}) \circledR \cdots \circledR (E1, u_1)$. Otherwise, the location found in E_i is the effective location of the variable provided the accumulator reads zero, which means there is no more unshadowing. If not, the accumulator is decremented by one and the process continues with the sub-extent $(E_{i-1}, u_{i-1}) \circledR \ldots \circledR (E_1, u_1)$. In the abnormal case, if there are no more simple extents, that is i = 0, the variable does not have an effective location in the composite extent.

Let $(E_n, u_n) \circledR \cdots \circledR (E_2, u_2) \circledR (E_1, u_1)$ be the effective extent during the computation of an expression. The first-class extent constructs other than **make-extent** are interpreted as follows. The expression *(get-extent)* returns the prefix $(E_n, u_n) \circledR \cdots \circledR (E_2, u_2)$ as a reified value. The expression *(remove-extent)* passes (E_1, u_1) as the new effective extent to the subsequent computation. The expression *(use-extent e),* where e denotes a composite extent e', forms the new effective extent c'®€ and passes it to the subsequent computation. For the expression **(shadowed?** i), where i denotes the vari-

able *a,* the unshadowing count of *a* in u_m is incremented by one to form the new unshadowing map $un = un[a \mapsto (u_m\ a) + 1]$ and an attempt to look up the variable a within the new effective extent (en, u_n')®• •®(E_1, u_1) is made. If the result indicates that the search is successful, the variable is shadowed; otherwise it is not. The denotation of the expression **(use-shadowed** i) is similar except that it passes the new effective extent to the subsequent computation.

The interpretation of the expression (**make-extent** *i* e), where *i* denotes the variable a, is as follows. The expression e is first evaluated to a value *v.* Next, a new location *l* is allocated and *v* is stored in it. Then, a simple extent $\{a \mapsto l\}$ and a constant unshadowing map $\lambda a . 0$ that associates every variable with a count of zero unshadowings are formed. The result of the **make-extent** expression is then a composite extent of a single element, the pair $(\{a \mapsto l\}, \lambda a . 0)$.

The valuation function [ ] is given in Figure 14. The notation used is summarized in the following. The notation ? denotes an unspecified value. It is the return value of constructs that are used only for their side-effect behavior. The notation $f[d \mapsto r]$ denotes the function g, an extension of the function f, with

$$g(d') \quad \begin{cases} r & \text{if } d' = d \\ f(d') & \text{otherwise.} \end{cases}$$

In order to make the continuation-passing-style semantics easier to read,

$$[e]\rho\kappa\sigma(\lambda v\kappa\sigma . E)$$

is written as

klet $(v, \varepsilon, \sigma) = [e]\rho\varepsilon\sigma$ **in**
E

5.2. Simulating Extents With States

We demonstrate that via an embedding [8] states can simulate first-class extents. In particular, we show that every basic extent construct can be defined by an equivalent Scheme expression that gives meaning to the construct. The embedding is derived from the denotational semantics given above. The semantics employs deep binding: each reference to a variable recomputes its effective location based on the effective extent. Yet each variable in Scheme is associated with exactly one location, its *shallow location.* As a result the success of the embedding is governed by the simulation of a variable's multiple locations with a single shallow location, i.e., the simulation of deep binding with shallow binding [4]. A complete description of the embedding is deferred to the appendix.

```
[ : Exp→ Env-~CExtt-~Sta- Con — Val

[2]~ p E 9 K = ~C (9(0 (p i) E)) E 0

[(set! i e)l pE9ic =
    klet (v, 9) _ [e]l p E 0 in
      K?E' B[c(pi)E v]

[(lambda (i) e)l p e e K, = K p E 9
    where p is Av(E (E, u))0K.
                [el (p[i a]) (E ® (E[a ']], u)) (0[11-4 v]) Ic
      with a being a fresh variable and 1 being a fresh location

Rep e z)l p E 9 K =
    klet (p, E, 9) = [epI p E 9 in
      klet (v, €, 9) = Pp]] pEB in
        pvEOK

[(call/cc e)l p E 0 K =
    klet (p, E, 9) = [el p E 9 in
      p(AVE0K f . KVE0)E0K

[(make-extent i e)l p E 9 Ic=
    klet (v, E, 0) = [e] p E 9 in
      ~c ({pi g--> 1}, A a . 0) E (0[11--÷ v])
        where 1 is a fresh location

[(get-extent)] p (E ®(e, u)) 0 IC = K E (E ® (E, u)) 9

[(use-extent e)l p E 0 K =
    klet (E', E, 9) = [e1 p E 0 in
      K?(E' E)9

(remove-extent)Jf p (E ® (E, u)) 9 K = K ? (E, u) 9

[(use-shadowed i)] p ((E, u) E) 0 K =
    K?((E,u[pi1-'u(pi)+1])®E)9

[(shadowed? i)l p ((E, u) €) 0 IC =
    (0 (pi) ((E, u[p i H u (pi) + 1]) ® E)) = undefined -~
      #cfalse ((E,u) E)9,
      lctrue ((E,u)®E)9
```

Figure 14: Valuation Function.

Briefly, the embedding is a generalization of the following state-based implementation of the fluid binding operation **fluid-let** [1, pages 324–325] that is found in many Scheme dialects.

```
(extend-syntax (fluid-let)
  ((fluid-let ((id exp)) body)
   (with ((old-val (gensym)) (ans (gensym)))
     (let ((old-val id))
       (set! id exp)
       (let ((ans body))
         (set! id old-val)
         ans)))))
```

The variable *id* assumes the value of the expression *exp* during the evaluation of the expression *body*. After that, *id* resumes its original value *old-val* before the result of *body is* returned. In other words, fluid-let creates a variable-location binding like **make-extent** and uses it fluidly like with-**extent** to affect the evaluation of an expression. But the binding is not a first-class value; it therefore cannot be saved and used later. Indeed, the behavior of **fluid-let** can be expressed with first-class extents as follows:

```
(extend-syntax (fluid-let)
  ((fluid-let ((id exp)) body)
   (with-extent (make-extent (id exp))
     body)))
```

A sound implementation of the fluid binding operation should wrap *a dynamic-wind* operation [5] around the **fluid-let** expression to guard against transfers of control in and out of the expression. Since extents are independent of control features and effective extents are available as first-class values, the mechanism can be easily implemented with first-class extents. Thus, the decision to implement such a mechanism becomes a user-level issue. In summary, the distinction that makes first-class extents a more appealing programming mechanism than fluid binding is that variable-location bindings are first-class values and there are flexible means to manipulate them.

We have shown that states can simulate extents, as indicated by the assignments used in the implementation of **fluid-let.** On the other hand, extents also provide side effects, although the effects are performed on variables rather than locations. For instance, *(use-extent* (**make-extent** *(id exp)))* installs a new effective location and therefore a new value for the variable *id.* Thus, in the subsequent computation, references to *id* denote the new value, reminiscent of **(set!** *id exp).* In fact, modeling side effects with extents provides an additional degree of flexibility that is not found

with traditional assignments alone. Besides being able to specify procedures (and continuations) with dynamic extents, *e.g.* **lambda,** we can also define procedures with static extents, *e.g.* **lambda/composable-extent** and **lambda/non-composable-extent.** The former are sensitive to the change of effective locations, but not the latter.

6. Conclusions

We do not want our programs to depend on the name of a variable. But first-class environments are too expressive to be completely written off just because they violate that constraint. To avoid the dependency, we distinguish a variable from its syntax and implementation, and create an intermediate map, extent, between scope and state:

$$\text{Identifier} \xrightarrow{\text{scope}} \text{Variable} \xrightarrow{\text{extent}} \text{Location} \xrightarrow{\text{state}} \text{Value}$$

Subsequently, we define first-class extents that are founded on variables with static scopes but dynamic extents, contrasting the dynamic scopes of first-class environments. Unlike first-class environments, first-class extents do not cause inadvertent name capturing anomalies. They are secure and modular.

There are situations, however, where first-class extents cannot replace first-class environments. Such situations occur 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.

Clearly first-class environments and extents both have their advantages and limitations. They should coexist. The interesting question is: "Which to use when?"

Acknowledgements

We owe much to Gary Drescher, whose Object Scheme inspired this work. We thank Matthias Felleisen, Julia Lawall, Anurag Mendhekar, John Simmons, and an anonymous referee for their insightful comments. The programs were typeset using the system provided by Carl Bruggeman.

References

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

2. W. Clinger and J. Rees (editors). Revised ^{4} report on the algorithmic language Scheme. *Lisp Pointers,* 4(3):1-55, 1991.

3. G. L. Drescher. Object Scheme: Object inheritance as fluid binding. Thinking Machines Corporation, 1990.

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

5. R. K. Dybvig. *The Scheme Programming Language.* Prentice Hall, 1987.

6. R. K. Dybvig. *Chez Scheme System Manual Revision 2.2.* **Cadence Research Systems, 1992.**

7. 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.

8. 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.

9. S. Jagannathan. Reflective building blocks for modular systems. In *Proceedings of the International Workshop on New Models for Software Architecture '92: Reflection and Meta-Level Architecture,* pages 61-68, 1992.

10. S. Jefferson and D. P. Friedman. A simple reflective interpreter. In *Proceedings of the International Workshop on New Models for Software Architecture '92: Reflection and Meta-Level Architecture,* pages 48-55, 1992.

11. S.-D. Lee and D. P. Friedman. First-class extents. Technical Report 350, Computer Science Department, Indiana University, March 1992.

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

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

14. J. W. Simmons and D. P. Friedman. A reflective system is as extensible as its internal representations: An illustration. Technical Report 366, Computer Science Department, Indiana University, October 1992.

15. B. C. Smith. Reflection and semantics in Lisp. In *Proceedings of the Eleventh ACM Symposium on Principles of Programming Languages*, pages 23-35, 1984.

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

Appendix

We demonstrate that first-class extents can be fully embedded [8] into Scheme. That is, every basic extent construct can be defined by an equivalent Scheme expression that gives meaning to the construct. The embedding is derived from the denotational semantics given in Section 5.1. There are two major tasks involved. First, composite extents must be made first-class values. Second, the semantics developed in that section uses deep binding: each variable reference recomputes its effective location based on the current effective extent. Yet each variable in Scheme has only one location, its *shallow location*, associated with it. As a result we need to simulate deep binding with shallow binding, i.e., simulate a variable's multiple locations with a single shallow location.

A complete source listing of the embedding written in *Chez* Scheme [6] is given in Section E. It uses two non-standard features, namely the error-handling procedure parameter *error-handler* and the syntactic extension facility **extend-syntax** with the uninterned-symbol generator *gensym* to support hygiene.

A. Simulating Composite Extents

We simulate the lexical variable associated with an identifier *id* by an accessor-setter pair of procedures generated by the following **id->var** syntactic extension:

```
(extend-syntax (id->var)
  ((id->var id)
   (with ((v (gensym)))
     (cons (lambda () id) (lambda (v) (set! id v))))))
```

Subsequently, to determine if two such pairs *var1* and *var2* denote the same variable, 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))))))
```

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

A variable-location binding is represented with a *cons* cell. The cell's *car* field holds the variable and the *cdr* field serves as its location. A simple extent is simulated by a tagged list of variable-location bindings. An unshadowing map is defined as a procedure that takes a variable and returns a non-negative integer. It uses the *var=?* predicate to compare variables. A composite extent is represented as a list of a simple extent and an unshadowing map pairs. Composite extent composition (nesting) is list append. The semantic function is easily translated directly into Scheme.

The base extent, denoted by the variable *base, is* initially empty. Whenever a variable is mentioned in a **make-extent** operation for the first time, a variable-location binding for the variable is incrementally added to *base.* In this way *base* will always appear to have a binding for every variable that has multiple locations. The simple extent *base is* the only one in the system that is extensible. The effective extent during a computation is a composite extent. It is denoted by a variable called *eff* that is consulted whenever the effective location of a variable is requested. The extent *base* is always the last element of *eff.*

With such representations of composite and effective extents, the operations *get-extent* and **make-extent** have straightforward interpretation. The *get-extent* operation simply makes a copy of the current effective extent *eff* without the base extent. That is, it duplicates the list in *eff* excluding its last element. The **make-extent** operation builds a composite extent consisting of a pair of a simple extent and an unshadowing map. It is a syntactic extension defined as follows:

```
(extend-syntax (make-extent)
  ((make-extent (id exp) ...)
   (cond
     ((and (defined? id) ...)
      (extend-base (id->var id) ...)
      (list
        (cons
          (list 'simple-extent (cons (id->var id) exp) ...)
          (lambda (vac) 0))))
     (else (error 'make-extent "~s" '(id . . .))))))
```

First we check that each of the identifiers *id* . . . mentioned is defined, i.e., it denotes some variable. This is done by the **defined?** syntactic extension of Section D. Next, the *extend-base* procedure adds to *base* variable-location bindings of the variables **(id->var** *id)* ... that are not yet defined in *base.* Then, we build a simple extent that associates the variables to new locations and an unshadowing map that indicates none of the variables is unshadowed. Finally, a composite extent consisting of the simple extent and the unshadowing map is returned.

B. Simulating Deep Binding

Because of shallow binding, every variable's shallow location is equated with its effective location. Thus, each time a variable is associated with a new effective location, its contents in the old effective extent and the shallow location must be updated accordingly. That is, the old effective location's up-to-date value, which is in the shallow location, must be saved. Then the new effective location's value must be written to the shallow location. We call this value switching process *switch-vals.* There are three basic constructs, described below, that can change a variable's effective location: *use-extent,* **use-shadowed,** and *remove-extent.* The **shadowed?** predicate also needs to determine if a variable's effective location can be altered.

The *remove-extent* procedure resets the effective extent *eff* to consist of only the base extent *base,* which is the new effective extent. But before that, it switches values between the new and old effective locations, with respect to the new and old effective extents, of every variable defined in *base.* The behavior of *(use-extent ext) is* similar. It switches values between effective locations of the current effective extent *eff* and the new effective extent that is the composition of the given composite extent *ext* and the current effective extent. Then it sets *eff* to the new effective extent.

The predicate **(shadowed?** *id) is* interpreted as follows. Shadowing exists only if the identifier *id* denotes a variable. This is made certain by the **(defined?** *id)* operation of Section D. Next, a new effective extent with

the unshadowing count of the variable (id->var *id)* incremented by one is obtained. Then an attempt is made to look up the variable's effective location in the new effective extent. If such a location is available, the variable can be unshadowed; otherwise, its effective location does not shadow any other location. The interpretation of **use-shadowed** is similar to that of **shadowed?.** The current and new effective extents are searched for the variable's effective location. If either search fails, the unshadowing cannot take place. Otherwise the values in the two effective locations are switched and the new effective extent is installed.

C. Robustness

The shallow binding embedding described in this appendix is complete, but *not* robust. There is no way to guarantee that at any point during a computation, the values of the effective locations, with respect to the effective extent *eff,* will be the same as those of the shallow locations. 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 stop 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 [2], we cannot provide a portable solution to the problem.

Yet it is possible on a case-by-case basis if an appropriate hook is available. For instance in *Chez* Scheme [6] there is a current error-handling procedure that is invoked when an error is detected. With the provision of such a mechanism, we can maintain the desired consistency by extending the definition of the error-handling procedure as follows:

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

The original error handler is saved and the new error handler is installed. When the new error handler is invoked, the effective extent is obtained and reinstalled immediately. Thus the effective extent is not changed; instead only the effective locations are forced to save their values from the shallow locations. This maintains consistency.

D. Unbound Identifier References

The basic first-class extent constructs having to do with variable reference, namely **make-extent, shadowed?,** and **use-shadowed** all could refer to an undefined identifier. In *Chez* Scheme such a reference causes an error because the identifier is considered "unbound." This is not acceptable in particular for **shadowed?** since a predicate should only evaluate to a boolean value. To remedy the problem in the embedding, we provide the following **defined?** syntax to detect whether an identifier is bound.

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

Operationally, a continuation that restores the old error handler and then returns to the continuation of the **defined?** expression is 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 if the identifier is unbound the new error handler is invoked and therefore the **defined?** expression is false. Otherwise the value associated with the variable is ignored and a true value is returned.

E. Program Listing

```
;;; private declarations

(define base (list 'simple-extent))
(define eff (list (cons base (lambda (var) 0))))

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

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

(define lookup-bnds
  (lambda (var bnds)
    (cond
      ((null? bnds) 'unbound)
      ((var=? (caar bnds) var) (car bnds))
      (else (lookup-bnds var (cdr bnds))))))

(define lookup-ext
  (lambda (var n sext ext)
    (let ((bnd (lookup-bnds var (cdr sext))))
      (cond
        ((eq? bnd 'unbound)
         (if (null? ext)
             'unbound
             (lookup-ext var (+ n ((cdar ext) var))
                         (caar ext) (cdr ext))))
        ((= n 0) bnd)
        ((null? ext) 'unbound)
        (else (lookup-ext var (+ (— n 1) ((cdar ext) var))
                          (caar ext) (cdr ext)))))))
```

```
(define lookup
  (lambda (var eff)
    (let ((n ((cdar eff) var)))
      (lookup-ext var n (caar eff) (cdr eff)))))

(define unshadow
  (lambda (var eff)
    (let ((old-uns (cdar eff)))
      (let ((new-uns
             (lambda (x)
               (if (var=? x var)
                   (+ (old-uns x) 1)
                   (old-uns x)))))
        (cons (cons (caar eff) new-uns) (cdr eff))))))

(define extend-base
  (lambda vars
    (cond
      ((null? vars) 'done)
      ((eq? (lookup-bnds (car vars) (cdr base)) 'unbound)
       (set-cdr! base
         (cons (cons (car vars) ((caar vars))) (cdr base)))
       (apply extend-base (cdr vars)))
      (else (apply extend-base (cdr vars))))))


(extend-syntax (id->var)
  ((id->var id)
   (with ((v (gensym)))
     (cons (lambda () id) (lambda (v) (set! id v))))))

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

```
;;; end of private declarations

(define get-extent
  (letrec ((loop
            (lambda (ext)
              (if (null? (cdr ext))
                  '()
                  (cons (car ext) (loop (cdr ext)))))))
    (lambda () (loop eff ))))

(define remove-extent
  (lambda ()
    (for-each
      (lambda (bnd)
        (switch-vals (lookup (car bnd) eff) bnd) )
      (cdr base))
    (set! eff (list (cons base (lambda (var) 0))))))

(define use-extent
  (lambda (ext)
    (let ((new-eff (append ext eff)) )
      (for-each
        (lambda (bnd)
          (switch-vals
            (lookup (car bnd) eff )
            (lookup (car bnd) new-eff)))
        (cdr base))
      (set! eff new-eff))))

(extend-syntax (make-extent)
  ((make-extent (id exp) ...)
   (andmap symbol? '(id ...))
   (cond
     ((and (defined? id) ...)
      (extend-base (id->var id) ...)
      (list
        (cons
          (list 'simple-extent (cons (id->var id) exp) .)
          (lambda (var) 0))))
     (else (error 'make-extent "~s" '(id ...))))))
```

```
(extend-syntax (shadowed?)
  ((shadowed? id)
   (and (defined? id)
        (let ((var (id->var id)))
          (let ((bnd (lookup var (unshadow var eff))) )
            (not (eq? bnd 'unbound)))))))

(extend-syntax (use-shadowed)
  ((use-shadowed id)
   (if (defined? id)
       (let ((var (id->var id)))
         (let ((new-eff (unshadow var eft)))
           (let ((old (lookup var eff) )
                 (new (lookup var new-eff)) )
             (cond
               ((or (eq? old 'unbound) (eq? new 'unbound))
                (error 'unshadowing "s" 'id))
               (else (switch-vals old new)
                     (set! eff new-eff))))))
       (error 'unshadowing "s" 'id))))

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