Maintaining Dynamic State: Deep, Shallow, and Parallel

Christopher T. Haynes[1]
Indiana University

Richard M. Salter[2]
Oberlin College

Abstract

In the presence of first-class continuations, shallow maintenance of dynamic bindings requires more than the traditional stack-based techniques. This paper provides correctness criteria for such dynamic environments, along with contrasting implementations. A store semantics provides the framework for our correctness criteria and presentation of deep- and shallow-binding implementations. The latter implementation is a new state-space algorithm, which is proved correct. A variation of the algorithm implements Scheme's dynamic-wind operation. Finally, a technique for maintaining dynamic state called *semi-shallow binding* is presented. This compromise between deep- and shallow-binding appears suitable for parallel systems. Applications include fluid binding of lexical variables and logic programming with first-class continuations.

1 Introduction

Dynamic state-management is required in many situations. Usually dynamic contexts are associated with control contexts, as with dynamic variables, "fluid" assignment to lexical variables [3], and the dynamic-wind operation [2]. Logic variable binding is an example of dynamic state-maintenance that is not strictly associated with control (return with success does not undo bindings).

In this paper we address the problem of maintaining multiple dynamic contexts when their extent is not limited by a first-in-last-out discipline. Examples include OR-parallel logic programming systems and systems with first-class continuations that capture their dynamic context. The latter is implied by the combination of the call-with-current-continuation and dynamic-wind operations in Scheme. To be specific, we focus on the problem of maintaining dynamic variable bindings in the presence of first-class continuations, though our analysis applies with minor modification to a variety of other dynamic state-management problems.

We first present a store-semantics that provides a framework for dynamic-binding correctness criteria. A simple deep-binding implementation is presented in this context. Shallow binding yields more efficient lookup, but in the presence of first-class continuations, stack-based techniques cannot be used. State-spaces have traditionally been used to maintain shallow bindings when control is abruptly transferred to prior contexts by the invocation of first-class continuations [1, 4]. A state-space algorithm, simpler than those previously reported, is presented and proved correct in our formal context. A generalization of the state-space algorithm that implements the dynamic-wind operation of Scheme is also provided. Finally, we propose a *semi-shallow binding* technique that represents a cross between the deep- and shallow-binding with promise for use in parallel systems.

Figure 1 presents a state semantics for a simple language with constants, variable references, functions that dynamically bind their arguments, applications, and a call-with-current-continuation

[1]Computer Science Department, Lindley Hall, Bloomington IN 47401. e-mail: chaynes@cs.indiana.edu.
[2]Computer Science Program, Oberlin OH 44074. e-mail: rms@cs.oberlin.edu. Research supported by NSF Grant CCR-9002132.

Syntactic domains:

$$c \in Const, \quad x, y \in Id, \quad e \in Exp$$
$$e ::= c \mid x \mid \mathsf{dlambda}\,(x)\,e \mid e_1\ e_2 \mid \mathsf{cwcc}\ e$$

Semantic domains:

$$\begin{array}{rll} b \in Bas & & \text{unspecified constant values} \\ l \in Loc & & \text{locations} \\ s \in State & & \text{states appropriate for lookup and update} \\ v \in Value & = Bas + (Value \to VCont \to LCont) & \\ k_v \in VCont & = Value \to LCont & \\ k_l \in LCont & = Loc \to State \to Ans & \\ Ans & = Value + Loc + State + (Value \times State) + \ldots & \end{array}$$

Semantic equations:

$$\begin{aligned} \mathcal{B} &: Const \to Bas \\ \mathcal{E} &: Exp \to VCont \to LCont \end{aligned}$$

$$\begin{aligned} \mathcal{E}[\![c]\!]k_v &= k_v(\mathcal{B}[\![c]\!]) \\ \mathcal{E}[\![x]\!]k_v &= \mathsf{lookup}\,x\,k_v \\ \mathcal{E}[\![\mathsf{dlambda}\,(x)\,e]\!]k_v &= k_v(\lambda v k_v' \,.\, \mathsf{update}\,x\,(\mathcal{E}[\![e]\!]k_v')\,v) \\ \mathcal{E}[\![e_1\ e_2]\!]k_v &= \lambda l \,.\, \mathcal{E}[\![e_1]\!](\lambda f l' \,.\, \mathcal{E}[\![e_2]\!](\lambda v \,.\, f v k_v)l)l \\ \mathcal{E}[\![\mathsf{cwcc}\ e]\!]k_v &= \lambda l \,.\, \mathcal{E}[\![e]\!](\lambda f \,.\, f(\lambda v k_v' l' \,.\, k_v v l)k_v)l \end{aligned}$$

lookup and update may be defined as in Figures 2 or 4, or as discussed in Section 5.

Figure 1: State semantics with dynamic binding

operator, cwcc, that captures the current continuation (control context) as a first-class functional value. The lookup and update operations abstract the maintenance of dynamic environments. Locations refer to dynamic environments maintained in the store. Since the current location is distinct from the store, both first-class continuations (created in the cwcc equation by $\lambda v k_v' l' \,.\, k_v v l$), and the semantic continuation for argument evaluation in applications ($\lambda f l' \,.\, \mathcal{E}[\![e_2]\!](\lambda v \,.\, f v k_v)l$) close only over the current location, l, not the current store. This preserves the single-threadedness [8] of the store, and reflects efficient implementation.

The procedure update takes an identifier, x, and a location continuation and returns a value continuation. The value continuation is invoked with a value, v, the current location, l, and the current state, s. A new location is then created and passed to the location continuation along with a new state that updates s in a manner that associates l with an environment that binds v to x.

The procedure lookup takes an identifier, x, and a value continuation and returns a location continuation. When the location continuation is passed the current location, l, and state, s, the value continuation is first passed the binding of x in the environment associated with l in s. The

Domain:

$$s \in State = Loc \to (Id \times Value \times Loc) + \{\underline{unbound}\}$$

Exported functions:

$$\begin{aligned}
\mathsf{lookup} &: Id \to VCont \to LCont \\
\mathsf{lookup}\, x\, k_v &= \lambda l\,.\,\mathsf{lookup1}\, x\, k_v\, l\, l \text{ wherec} \\
&\quad \mathsf{lookup1}\, x\, k_v\, l_o\, l = \\
&\quad\quad \lambda s\,.\,\text{case } sl \text{ of} \\
&\quad\quad\quad \langle x', v, l' \rangle \to \text{if } x = x' \text{ then } k_v v l_o s \text{ else } \mathsf{lookup1}\, x\, k_v\, l_o\, l'\, s \\
&\quad\quad\quad \text{else } \to k_v\, \underline{unbound}\, l_o s
\end{aligned}$$

$$\begin{aligned}
\mathsf{update} &: Id \to LCont \to VCont \\
\mathsf{update}\, x\, k_l &= \lambda v l s\,.\,\text{let } l' = \mathsf{new}\, s \text{ in } k_l l' s[l' \mapsto \langle x, v, l \rangle]
\end{aligned}$$

Figure 2: Deep binding

current location is then passed on to the value continuation along with a state that may be either s or a modification of s that reflects a change in the current location.

A correct implementation of lookup and update (the environment abstract data type) must reflect the customary rules of functional extension associated with environments. These rules are formalized before Theorem 3.2.2 in section 3 as the *environment-state invariant.*

In the absence of the cwcc operation, all environment information may be maintained on a stack, since the update and lookup operations then maintain a last-in-first-out discipline: the current location either remains unchanged, is changed to correspond to a new location created by an update, or reverts to the location that was current before the last update.

On the other hand, first-class continuations created by cwcc restore their creation-time environment when they are invoked. Thus when a first-class continuation is created sufficient information must be maintained to allow restoration of the creation-time environment e. This may be at any future time, including contexts in which the current environment has reverted to an environment created prior to the creation of e and then been extended with alternate bindings of some of e's variables. Since context changes can occur in an arbitrary fashion, a stack discipline no longer suffices to maintain the information necessary to restore e when the continuation is invoked. It must be replaced by a tree structure in which old context information persists with indefinite extent.

An implementation reflecting the traditional *deep-binding* approach to maintaining a dynamic binding environment is presented in Figure 2. Environments are represented as association lists, which are space-efficient because all extensions of a given environment share the same tail. Moreover, this structure makes deep binding a natural choice for implementing dynamic binding with first-class continuations. Though extending a deep-bound environment or returning to a previous one are time-efficient operations, deep-bound dynamic environment lookup is inefficient. A

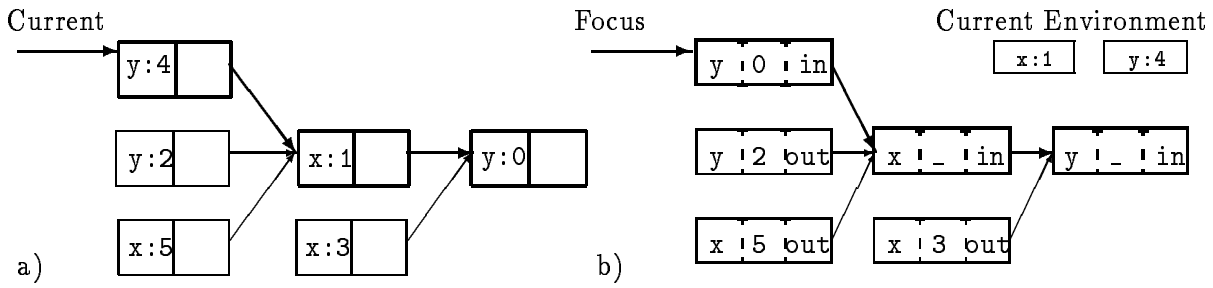

Figure 3: Environment models: a) Linked list b) State Space

deep-bound environment is illustrated in Figure 3a. It is not difficult to prove this implementation satisfies the environment-state invariant.

When *shallow binding* is used the current value of each variable is stored in a location that is permanently assigned to that variable. The contents of these locations are changed as necessary to reflect environment updates and returns to prior environments. Thus environment bindings are maintained in a singly-threaded store, while a separate data structure records whatever information is necessary to restore environment bindings when control returns to a prior environment. Again, in the absence of cwcc a simple stack regimen can be used to implement singly-threaded, shallow-bound environments. The requirements of first-class continuations, however, compel us to employ a more general mechanism for environment maintenance, referred to as a *state-space algorithm.*

Section 2 presents an implementation of update and lookup using a state-space algorithm that supports shallow binding. Section 3 proves the state-space algorithm correct. Section 4 presents a variation on the state-space algorithm that implements the dynamic-wind function of Scheme. Section 5 presents an alternate implementation of update and lookup that represents a compromise between deep and shallow binding, and appears suitable for parallel implementations. Section 6 concludes with a discussion of related work and other applications of the dynamic state maintenance techniques presented here, including logic programming.

2 A state-space algorithm

In the absence of the cwcc operation, shallow binding is easily implemented using a stack. When an environment is extended at the point of a procedure call with a new value v for some identifier x, it suffices to push the old value of x and the symbol x onto a stack. On procedure return the stack is popped to restore the prior value of x as the current location reverts to the location that was current before the environment extension.

When a first-class continuation is created with cwcc, however, the stack must be allowed to branch into a tree, called a "state space." Each tree node represents a different environment, storing one variable and one binding for that variable. The node corresponding to the current environment is called the *focus*. If a node q managing variable x is either the focus or its ancestor,

Domains:

$$
\begin{aligned}
l, f, p &\in Loc \\
m &\in Mode = \{\underline{in}, \underline{out}\} \\
n &\in Node = Mode \times Id \times Value \times Loc \\
e &\in Env = Id \to Value + \{\underline{unbound}\} \\
t &\in Tree = Loc \to Node + \{\underline{unused}\} \\
s &\in State = Env \times Tree \times Loc
\end{aligned}
$$

Exported functions:

$$
\begin{aligned}
\mathsf{lookup} &: Id \to VCont \to LCont \\
\mathsf{lookup}\, x\, k_v &= \lambda l s\, .\, \text{let } s' = \mathsf{refocus}\, l\, s \text{ in } k_v(e_{s'} x) l s'
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{update} &: Id \to LCont \to VCont \\
\mathsf{update}\, x\, k_l &= \lambda v l s\, .\, \text{let } s' = \mathsf{refocus}\, l\, s \\
&\qquad \text{in let } l' = \mathsf{new}\, s' \\
&\qquad\quad \text{in } k_l l' \langle e_{s'}[x \mapsto v],\, t_{s'}[l' \mapsto \langle \underline{in}, x, e_{s'} x, l \rangle],\, l' \rangle
\end{aligned}
$$

Local functions:

$$
\begin{aligned}
\mathsf{refocus} &: Loc \to State \to State \\
\mathsf{refocus}\, l\, s &= \text{if } l = f_s \text{ then } s \\
&\qquad \text{else case } t_s l \text{ of} \\
&\qquad\quad \langle \underline{out}, x, v, p \rangle \to \mathsf{enter}\, l\, (\mathsf{refocus}\, p\, s) \\
&\qquad\quad \langle \underline{in}, x, v, p \rangle \to \mathsf{refocus}\, l\, (\mathsf{leave}\, s)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{enter} &: Loc \to State \to State \\
\mathsf{enter}\, l\, s &= \text{let } \langle \_, x, v, p \rangle = t_s l \text{ in } \langle e_s[x \mapsto v],\, t_s[l \mapsto \langle \underline{in}, x, e_s x, p \rangle],\, l \rangle
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{leave} &: State \to State \\
\mathsf{leave}\, s &= \text{let } \langle \_, x, v, p \rangle = t_s f_s \text{ in } \langle e_s[x \mapsto v],\, t_s[f_s \mapsto \langle \underline{out}, x, e_s x, p \rangle],\, p \rangle
\end{aligned}
$$

Figure 4: Shallow-binding state-space algorithm

then q is *in* with respect to the current environment. In this case q stores a previous value for x, which is restored if the focus shifts above q. Otherwise, we say that q is *out*, and q contains the binding for x in q's associated environment. This must be restored should the focus shift back within q. Figure 3b illustrates the state space that manages the shallow equivalent of Figure 3a.

When the current environment shifts to a prior environment, it is necessary to walk the path from the node corresponding to the current environment to that corresponding to the prior one. If a node is in with respect to the current environment and is out with respect to the prior one, we say we are *leaving* the node; if it currently out and will become in, we are *entering* the node. When either entering or leaving a node, the current and stored values are exchanged.

For our state-space algorithm, we define a state $s = \langle e_s, t_s, f_s \rangle$ containing both the current environment and the current system of alternate environments represented skeletally by a state space. The environment of the state is modeled as the map e_s, taking identifiers to values. The state space is comprised of the map t_s and the distinguished location f_s. t_s models a store taking locations to nodes and is called the *node tree* of the state. A node $n = \langle m_n, x_n, v_n, p_n \rangle$ consists of a *mode* m_n, with value *in* or *out*, a variable x_n *bound* by the node, the bound variable's *stored value* v_n, and the location p_n of the node's *parent* in the node tree. Location f_s, the focus of the state, references the current focus in the node tree.

> Note on terminology: Traditionally the term "state space" refers to structures that maintain a "space" consisting of a number of "states" representing distinct execution environments. Thus a state space corresponds in our terminology to a node tree and the states of a state space correspond to our environments. The phrase "state space" corresponds well with the programmer's view, but not with the semantic use of the term state as a domain of values that are singly-threaded.

The focus of a state indicates the node tree element corresponding to the environment of the state. Each update operation extends the node tree from the current focus with a new node that becomes the new focus. The mode of the current focus and each of its ancestors is *in*, while all other nodes are *out*.

The fundamental state-space operation is *refocusing*: moving its focus to a given location. This is accomplished by the refocus procedure of Figure 4. If the focus f_s of the current state is the same as the given location l, there is nothing to do, and the current state is returned. Otherwise, if the node corresponding to the given location is not the current focus but is *in*, then the procedure leave is first called to obtain a state in which the focus has been moved out of the current state to its parent. The procedure refocus is then called again with the same location l and the new state. If the node corresponding to the given location is *out*, then refocus is called first to obtain a state in which the focus has been moved to the parent of the given location. The procedure enter is then called to move the focus into the given location.

The update operation accepts an identifier, a location continuation, a value, and the current location and state. It first calls refocus to obtain a state whose focus is the current location. A new

location is then generated by calling the procedure new, and this location is passed to the location continuation along with a new state in which the environment has been appropriately extended, and an extended node tree with an added node corresponding to the new location.

The lookup operation accepts an identifier, a value continuation, and the current location and state. The focus is first moved to the current location, producing a new state. The value continuation is then invoked with the value corresponding to the identifier in the environment of the new state, plus the current location and the new state.

3 Correctness of the state-space algorithm

The correctness of this model is presented in three steps. First we show that only a certain subset of possible states can arise; we refer to these as constructible states. Next we show that for constructible states the refocus algorithm always terminates, and that the set of $\underline{in}$-nodes is precisely the path from the focus to the root of the node tree. Finally, we show that when s is constructible, refocus $l\, s$ produces an environment that depends only on l, and that this environment is the one created when l was allocated by update.

3.1 Constructibility

The update and refocus functions are well defined only for locations that have meaning in the given state. Specifically, we say a location l is *bound* in state s if $t_s l \neq \underline{unused}$.

Assuming the initial environment is empty, the algorithm requires an *initial state,* ϵ, containing a single node with mode $\underline{in}$ and arbitrary contents in its other fields:

$$\epsilon = \langle \{\}, \{f_\epsilon \mapsto \langle \underline{in}, \_, \_, \_ \rangle\}, f_\epsilon \rangle$$

Definition 3.1.1 *A* constructible state sequence $\{s_0, \ldots, s_n\}$ *is one in which* $s_0 = \epsilon$ *and, for* $0 \leq i \leq n-1$, s_{i+1} *is obtained from* s_i *by either a* lookup *or* update *operation. A state is* constructible *if it is an element of some constructible state sequence.*

In a constructible state, each bound location has been allocated by some invocation of update. The node initially referenced by the location contains a bound variable and parent that remain unchanged by subsequent refocusing. The states s_i in a constructible sequence contain structurally compatible node trees, rooted at $t_{s_i} f_\epsilon$. The current node tree is extended by updates and retains its structure during refocusing, although modes and stored values change.

Definition 3.1.2 *A state is* mode-proper *if the set of* $\underline{in}$*-nodes is exactly the set of nodes along the path from* f_s *to* f_ϵ *in the node tree.*

Lemma 3.1.1 *If* s *is mode-proper and* l *refers to a node of* t_s*, then*

1. refocus $l\,s$ *terminates without error;*

2. *If* $s' = \mathsf{refocus}\,l\,s$, *then* $l = f_{s'}$.

3. $s' = \mathsf{refocus}\,l\,s$ *is mode-proper.*

Proof. To prove that $\mathsf{refocus}\,l\,s$ terminates, we show that the number of recursive calls to refocus is equal to the length of the path in the node tree between l and f_s. Since the algorithm terminates when $l = f_s$, it suffices to show that with each recursive call the length of this path is reduced. Consider first the case where l is $\underline{out}$. Let p be the parent of l. Since s is mode-proper, l cannot be an ancestor of f_s, and so p lies on the path between l and f_s. Therefore, the path length is reduced by one in the recursive call $\mathsf{refocus}\,p\,s$. Now suppose l is $\underline{in}$. Then l must be an ancestor of f_s, so that f_s cannot be f_ϵ and therefore has a parent. Since the recursive call is made on a state whose focus is the parent of f_s, the path is again reduced by one.

We use induction on the number of recursive calls to show that $l = f_{s'}$ and s' is mode-proper. In the base case, if $l = f_s$, then $s' = s$. Now assume the induction assumption that $l = f_{s'}$ and s' is mode-proper whenever l and f_s are n nodes apart. Suppose that l and f_s are $n+1$ nodes apart. If l is $\underline{out}$ ($m_{t_{s'}\,l} = \underline{out}$), then by the induction assumption the recursive call $\mathsf{refocus}\,p\,s$ produces a mode-proper state, s_p, whose $\underline{in}$-locations are on the path from the root to focus p. In s', this path is extended to the new focus l. Now suppose that l is $\underline{in}$. To complete the proof we must show that the state to which the recursive call is applied is mode-proper. But this is clear since its node tree is identical to that of s except for having been refocused one node toward the root of the node tree, with the mode of the node associated with the old focus changed to $\underline{out}$. □

That states remain mode-proper is a key invariant of the state-space algorithm. To reason about lookup and update we introduce the pairing location continuation $\pi_l{=}\lambda l s\,.\,\langle l, s\rangle$ and the pairing value continuation $\pi_v{=}\lambda v l s\,.\,\langle l, s\rangle$.

Lemma 3.1.2 *Every constructible state is mode-proper.*

Proof. If $\langle l', s'\rangle = \mathsf{update}\,x\,\pi_l\,v\,l\,s$ and s is mode-proper, then s' is mode-proper and $l' = f_{s'}$, since update extends a refocused node tree at its focus with an $\underline{in}$-node, which becomes the new focus. Also, if $\langle l', s'\rangle = \mathsf{lookup}\,x\,\pi_v\,l\,s$, then $s' = \mathsf{refocus}\,l\,s$ is mode-proper by part 3 of lemma 3.1.1. The lemma follows by induction, given that ϵ is mode-proper. □

Finally, we require the following technical lemma:

Lemma 3.1.3 *If* s *is a mode-proper state, then* $t_s f_\epsilon$ *is* $\underline{in}$.

Proof. In $\mathsf{refocus}\,l\,s$, leave is applied only when l is $\underline{in}$. But if s is mode-proper, l must be an ancestor of f_s, so that $f_s \neq f_\epsilon$. □

3.2 Proper States

We now consider the link between the node tree and environment. In a constructible state a new location is added to the node tree by an update operation. We pay special attention to the environment created by that update, since subsequent calls to refocus with that location must re-create this environment.

Inspecting the algorithm for update, we see that if $\langle l, s\rangle = \mathsf{update}\, x\, \pi_l\, v\, l'\, s'$, the environment e_s is produced by extending with the binding $x \mapsto v$ the environment $e_{s''}$, where $s'' = \mathsf{refocus}\, l'\, s'$. We call x the *variable managed by* l, denoted x_l, e_s the *environment managed by* l, denoted e_l, $e_{s''}$ the *base environment for* l, denoted e_l^{b}, v the *new binding at* l, denoted v_l^{n}, and $e_l^{\mathrm{b}} x_l$ the *old binding at* l, denoted v_l^{o}.

This notation is well-defined, since each location is allocated by a unique update in a given proper state sequence, and thus we have:

Lemma 3.2.1 *If s_i and s_j are elements of the same state space, and l is active in both, then x_l, e_l, e_l^{b}, v_l^{n}, and v_l^{o} are the same in both s_i and s_j, and $e_l = e_l^{\mathrm{b}}[x_l \mapsto v_l^{\mathrm{n}}]$.* □

We now define the key property whose invariance under update and refocus will prove the correctness of the state space model:

Definition 3.2.1 *A constructible state, s, is* proper *if*

1. $e_s = e_{f_s}$.
2. *For each location $l \neq f_\epsilon$ such that $n = t_s l \neq \underline{\mathit{unused}}$,*
   (a) $e_l^{\mathrm{b}} = e_{p_n}$
   (b) *If l is $\underline{\mathit{in}}$, then the stored value at l is v_l^{o}*
   (c) *If l is $\underline{\mathit{out}}$, then the stored value at l is v_l^{n}*

Thus in a proper state the environment is always the one managed by the focus, and the stored values are either the new or old bindings, depending on whether the node is $\underline{\mathit{out}}$ or $\underline{\mathit{in}}$, respectively.

To show that refocusing a proper state s produces a proper state, we first prove that this property is preserved by $\mathsf{leave}\, s$, and by $\mathsf{enter}\, l\, s$ whenever l is a child of f_s.

Lemma 3.2.2 *If s is proper, then* leave s *is proper whenever $f_s \neq f_\epsilon$ and* enter $l\, s$ *is proper whenever $f_s = p_{t_s l}$ (the focus of s is the parent of l).*

Proof. Neither enter nor leave change the base environment of any location, so condition (2a) of Definition 3.2.1 continues to hold. Suppose $l = p_{f_s}$, and let $s' = \mathsf{leave}\, s$, so that l is the focus of s'. Now f_s manages variable x_{f_s}, with old binding $v_{f_s}^{\mathrm{o}}$ and new binding $v_{f_s}^{\mathrm{n}}$. Since s is proper,

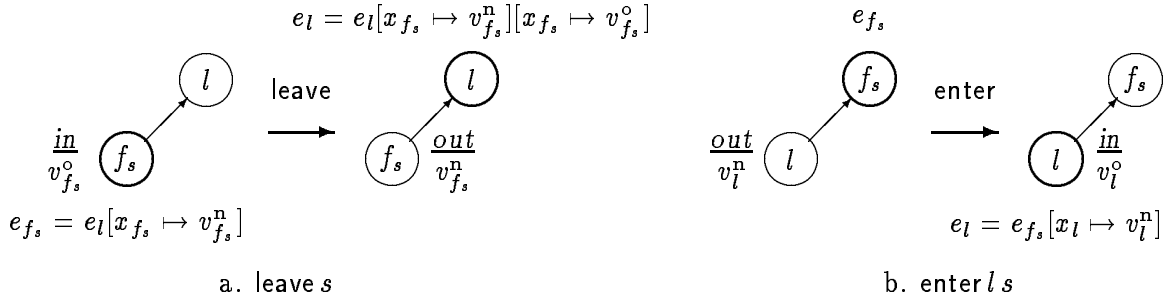

Figure 5: Dynamics of leave and enter.

$e_s = e_{f_s} = e_l[x_{f_s} \mapsto v^{\mathrm{n}}_{f_s}]$, while since f_s is $\underline{in}$ in s (it is the focus), the value stored at f_s in s must be $v^{\circ}_{f_s}$, which is the value of x_{f_s} in e_l. But since this is the value assigned by leave to x_{f_s}, we see that $e_{s'} = e_s[x_{f_s} \mapsto v^{\circ}_{f_s}] = e_l[x_{f_s} \mapsto v^{\mathrm{n}}_{f_s}][x_{f_s} \mapsto v^{\circ}_{f_s}] = e_l$ (see Figure 5a), which gives condition (1). Moreover, f_s is $\underline{out}$ in s' with stored value $e_s x_{f_s} = v^{\mathrm{n}}_{f_s}$, and this is the only node whose mode has changed, which gives us conditions (2b) and (2c). Thus s' is proper.

Now suppose $f_s = p_{t_s l}$, and let $s' = \mathsf{enter}\, l\, s$, so that, again, l is the focus of s'. This time, l manages x_l and since l is $\underline{out}$ in s (it is a child of the focus) the value stored at l is v^{n}_l. Therefore the state change in enter produces $e_{s'} = e_{f_s}[x_l \mapsto v^{\mathrm{n}}_l] = e_l$ (see Figure 5b), giving us condition (2). Since l is $\underline{in}$ in s' with stored value $e_{f_s}\, x_l = v^{\circ}_l$, and this is the only node whose mode has changed, we again get conditions (2b) and (2c). Thus s' is again proper. □

The invariance of proper-ness under refocusing may now be demonstrated:

Theorem 3.2.1 *If s is proper, then* refocus $l\, s$ *is proper.*

Proof. This follows by induction on the length of the path between the old and new foci, using Lemma 3.2.2, Part 2 of Lemma 3.1.1, Corollary 3.1.2 (to guarantee that leave is never applied to the root of the node tree) and the fact that ϵ is proper. □

Finally, to show that every constructible state in proper, we must show that updating a proper state produces a proper state. Using the state-returning location and value continuations $\sigma_l = \lambda l s\,.\,s$ and $\sigma_v = \lambda v l s\,.\,s$ we have:

Corollary 3.2.1 *If s is proper, then $s' =$* update $x\, \sigma_l\, v\, l\, s$ *and $s'' =$* lookup $x\, \sigma_v\, v\, l\, s$ *are proper.*

Proof. From Theorem 3.2.1 we know that $e^{\mathrm{b}}_{s'_f} = e_{s_f}$. The rest follows directly by inspection of the state change described in the definitions of lookup and update. □

Corollary 3.2.2 *Every constructible state is proper.* □

Any state-based implementation of lookup and update must guarantee that in any state following the creation of an environment the customary rules of functional extension associated with environments are satisfied. The following *environment state invariant* formalizes this requirement.

> *If in a proper state sequence* $\langle l', s_i \rangle = \mathit{update}\, x\, \pi_l\, v\, l\, s_{i-1}$ *for some* $i > 0$ *and* $\nu_v = \lambda vls\,.\,v$, *then for any* $j \geq i$,
>
> - $\mathsf{lookup}\, x\, \nu_v\, l'\, s_j = v$; *and*
> - $\mathsf{lookup}\, y\, \nu_v\, l'\, s_j = \mathsf{lookup}\, y\, \nu_v\, l\, s_j$ *for* $y \neq x$.

The following theorem then establishes the correctness of the state-space algorithm.

Theorem 3.2.2 *The state-space algorithm of Figure 4 satisfies the environment state invariant.*

Proof. The environments used by $\mathsf{lookup}\_\ k_v\, l\, s_j$ and $\mathsf{lookup}\_\ k_v\, l'\, s_j$ are, respectively, $e_{s'}$ and $e_{s''}$, where $s' = \mathsf{refocus}\, l\, s_j$ and $s'' = \mathsf{refocus}\, l'\, s_j$. Since s_j is proper, so are s' and s'', so that $e_{s''} = e_{l'}$ and $e_{s'} = e_l = e_{l'}[x \mapsto v]$ (since l' is the parent of l) $= e_{s''}[x \mapsto v]$. Since $\mathsf{lookup}\, y\, \nu_v\, l\, s_j$ yields $e_{s'}y$, this is v if $y = x$, and $e_{s''}y = \mathsf{lookup}\, y\, \nu_v\, l'\, s_j$ if $y \neq x$. □

4 Dynamic-wind

In a language with any form of programmer-manageable state, there may be a need to guarantee that some operation, called a *postlude*, be performed whenever control leaves a dynamic context. Control may leave either by normal control flow or through an escaping mechanism, such as a throw to a Lisp catch tag, a C long-jump, an ML exception, or a Scheme continuation. Lisp systems often provide an unwind-protect mechanism for this purpose [9].

With the introduction of first-class continuations, such as are provide by Scheme's call-with-current-continuation procedure, there may also be a need to perform another operation, called a *prelude*, whenever control enters a dynamic context either via normal control flow or via invocation of a continuation that had previously been captured within the context.

To fulfill both of these needs, Scheme supports a dynamic-wind procedure that generalizes unwind-protect by allowing both a prelude and a postlude to be associated with the dynamic context in which a given *body* is evaluated [2]. Its arguments are nullary procedures that are invoked to perform the prelude, body, and postlude operations, respectively. Using dynamic-wind it is straightforward to support dynamic binding of lexical variables, as with the fluid-let form [3].

The dynamic-wind procedure may be defined in Scheme using a state space that is similar in concept to that which supports the lookup and update operations of Section 2. The new complication is that the state-space mechanism should be robust in spite of attempts to enter or leave prelude or postlude computations via invocation of first-class continuations. Without some protection mechanism, the state space and/or the user's program could be left in an inconsistent state. To

```
(define (make-node prelude postlude parent) (vector #f prelude postlude parent))
(define (is-out? node) (vector-ref node 0))
(define (set-node-is-out! node bool) (vector-set! node 0 bool))
(define (prelude node) (vector-ref node 1))
(define (postlude node) (vector-ref node 2))
(define (parent node) (vector-ref node 3))

(define (toggle-is-out! node)
  (set-node-is-out! node (not (is-out? node))))

(define focus (make-node 'ignored 'ignored 'ignored))

(define (refocus new-focus)
   (cond
     ((is-out? new-focus)
      (refocus (parent new-focus))
      (set! focus new-focus)
      (toggle-is-out! focus)
      ((prelude focus)))
     ((not (eq? focus new-focus))
      (set! focus (parent focus))
      (toggle-is-out! focus)
      ((postlude focus))
      (refocus new-focus))))

(define orig-call/cc call-with-current-continuation)

(define (call-with-current-continuation f)
  (orig-call/cc
    (lambda (k)
      (f (let ((saved-focus focus))
           (lambda (x)
             (if (not (eq? saved-focus focus)) (refocus saved-focus))
             (k x)))))))

(define (dynamic-wind prelude body postlude)
  (prelude)
  (let ((answer (proto-dynamic-wind prelude body postlude)))
    (postlude)
    answer))

(define (proto-dynamic-wind prelude body postlude)
  (set! focus
    (let* ((sealed #f) (seal! (lambda () (set! sealed #t))))
      (make-node
        (lambda () (if sealed (error) (proto-dynamic-wind error prelude seal!)))
        (lambda () (proto-dynamic-wind error postlude seal!))
        focus)))
  (let ((answer (body)))
    (set! focus (parent focus))
    answer))
```

Figure 6: Dynamic-wind in Scheme

accommodate exception handling, transfer out of a prelude or postlude should be allowed—provided that control never reenters the associated state-space node. See figure 6 for an implementation of `dynamic-wind` that implements this policy. An error is signalled (via the procedure `error`) in a node's prelude if the node-specific variable `sealed` is set, which occurs if control leaves the node's prelude or postlude prematurely. An error is also signaled if control enters a prelude or postlude via a continuation.

5 Distributed semi-shallow binding

The state-space algorithm does not support concurrent instances of multiple, incompatible environments. If state spaces were used in a concurrent system, each process would have to have its own copy, since later state space actions at the parent would render the creation time environment inaccessible. In this section we sketch a distributed form of binding that might be termed *semi-shallow binding*, since it represents a compromise between shallow and deep binding. It may be implemented in the context of Figure 1 by appropriate definition of lookup and update, and its correctness verified using the environment state invariant of Section 3.

This approach maintains a separate *context tree* for each variable. Each context tree is a projection of the global state space maintaining only the bindings a given variable and the path information needed to connect these bindings. The current context is now represented by a *context path*, instead of a state space pointer. Rather than using a location to extract from a single state space an environment in which a variable binding may be found, we now use a variable to identify the context tree from which a context path can extract the variable binding.

The context path is depicted as a numerical sequence, representing a path through a global node tree. Context tree edges are labeled by corresponding context path values. Knowledge of the out-degree of each node tree element is required so that each new context may be given a unique numerical labeling. This may be obtained through a global *context structure* tree that is consulted and extended with each call of update.

Context trees are typically sparse, frequently containing chains of nodes without values. Such chains may be *path compressed* to a single edge labeled by the corresponding path. Pereira's virtual copy arrays are another example of a tree structure dedicated to bindings of a single variable [6]. Our context trees differ in that our trees may have compressed paths and values at interior nodes.

Figure 7 illustrates this approach, representing the same dynamic information as in Figure 3, but restricted to the variable `x`. Each box contains a binding for `x` in the environment represented by its node, while unboxed numbers annotate the path. The current context path is then $\langle 0,0,0\rangle$. Performing `lookup(x)` we follow the path specified by the context path as far as possible, locating the node with a value box containing `1`, the current binding of `x`. The operation `update(x,y)` causes the entire context path to be traversed. Since the final 0-branch does not exist in the context tree, we add a new branch labeled `0`. Having reached the end of the current context path, we extend it with another edge labeled `0` (since the current focus has out-degree 0), but immediately combine

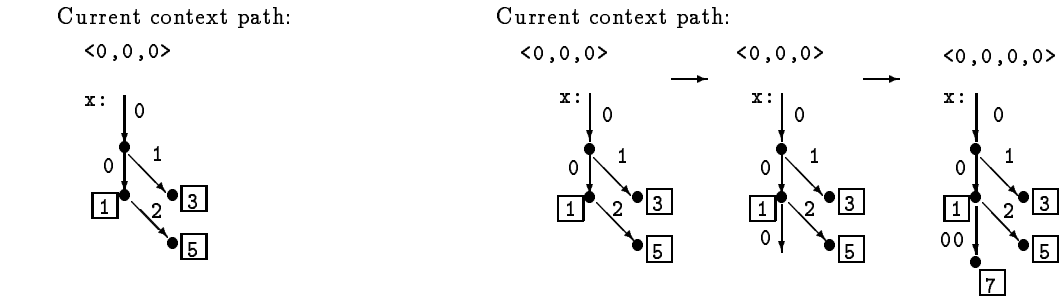

Figure 7: Context path environments

this edge with the last one to form a compressed edge labeled 00. A new node with a box containing 7 is then attached to the new edge.

Semi-shallow binding has been used in the implementation of a graph-rewrite system for logic programming [5]. Preliminary analysis indicates that semi-shallow binding may reasonably be adapted to parallel computation systems without shared memory. Since context trees and the context structure are monotonic (modified only via extension), coherence problems are avoided. Experimental study of this approach is ongoing.

6 Conclusion

We have addressed the problem of efficiently implementing dynamic binding by presenting correctness criteria and alternative implementations in the context of a singly-threaded store semantics. These correctness criteria and implementation techniques also apply to other situations in which dynamic state must be maintained efficiently in the presence of first-class continuations. For example, in implementing a variant of Prolog that supports first-class success and failure continuations we have used the first state-space algorithm presented here to maintain two separate state spaces: one maintaining logic variable bindings and the other providing dynamic protection against database mutation [7]. We believe our second implementation technique, based on context paths, may provide an attractive approach to the maintenance of multiple environments in highly-parallel logic programming systems, though experience is needed to explore the system parameters within which this may be the technique of choice.

Acknowledgement: The authors wish to thank Rhys Price Jones for his contribution to the development of semi-shallow binding.

References

[1] H. G. Baker Jr. Shallow binding in Lisp 1.5. *Comm. ACM*, 21(7):565–569, July 1978.

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

[3] R. K. Dybvig. *The Scheme Programming Language*. Prentice-Hall, 1987.

[4] C. Hanson and J. Lamping. Dynamic binding in Scheme. Unpublished manuscript, 1984.

[5] R. P. Jones and R. M. Salter. Implementing nondeterminism using graph reduction. Technical report, Oberlin College, 1993.

[6] F. C. N. Pereira. A structure-sharing representation for unification-based grammar formalism. In *Proc. of the 23rd Annual Meeting of the Assoc. for Comp. Linguistics*, pages 137–144, 1985.

[7] R. Salter and C. T. Haynes. Continuation-based control operators for logic programming. Technical Report 293, Indiana University, 1989.

[8] D. A. Schmidt. Detecting global variables in denotational definitions. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, April 1985.

[9] G. L. Steele Jr. *Common Lisp: The Language*. Prentice-Hall, 1991.