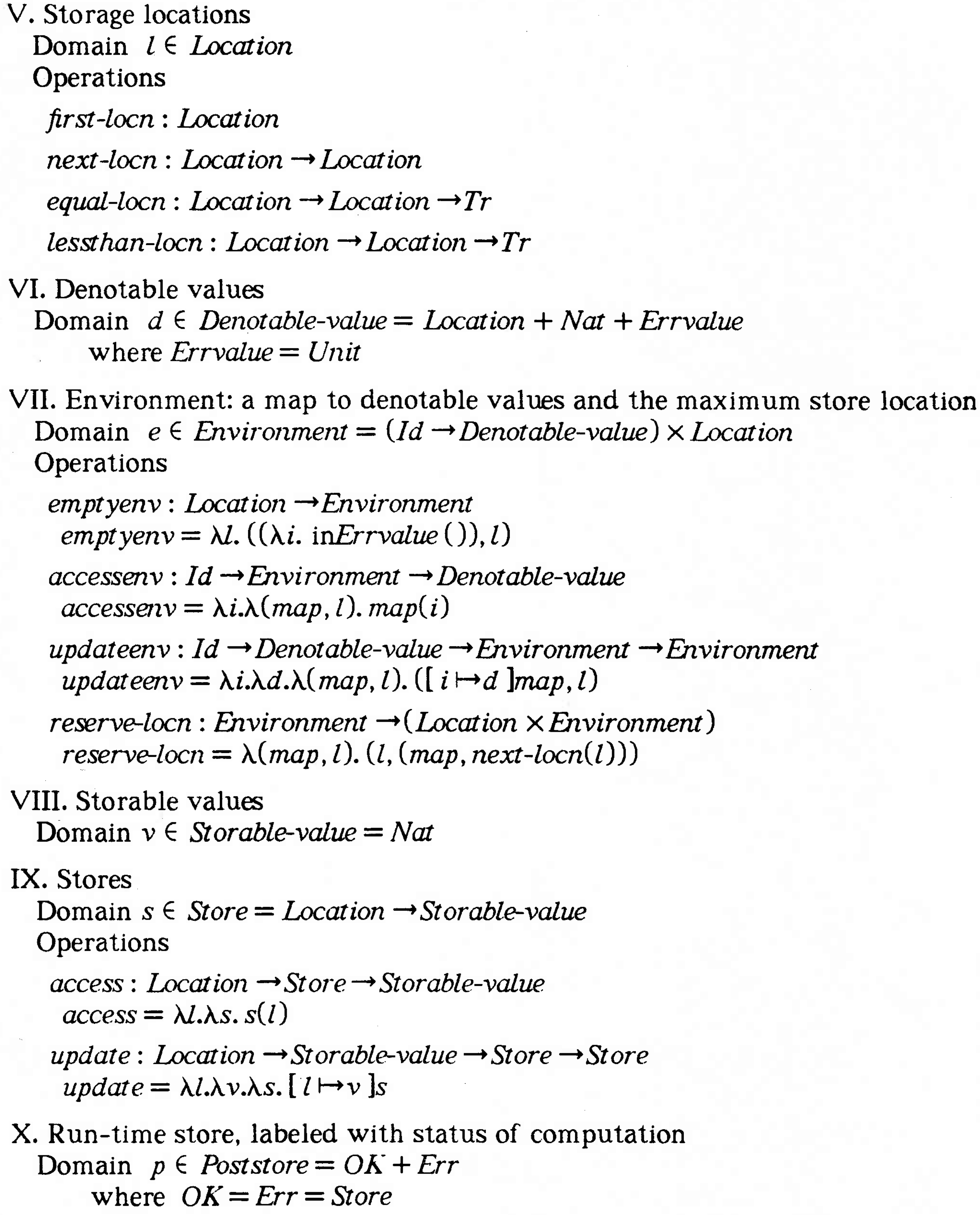
Figure 7.1



V. Storage locations
Domain $l \in Location$
Operations

$first\text{-}locn : Location$

$next\text{-}locn : Location \rightarrow Location$

$equal\text{-}locn : Location \rightarrow Location \rightarrow Tr$

$lessthan\text{-}locn : Location \rightarrow Location \rightarrow Tr$

VI. Denotable values
Domain $d \in Denotable\text{-}value = Location + Nat + Errvalue$
where $Errvalue = Unit$

VII. Environment: a map to denotable values and the maximum store location
Domain $e \in Environment = (Id \rightarrow Denotable\text{-}value) \times Location$
Operations

$emptyenv : Location \rightarrow Environment$
$emptyenv = \lambda l.\, ((\lambda i.\ \mathrm{in}Errvalue()), l)$

$accessenv : Id \rightarrow Environment \rightarrow Denotable\text{-}value$
$accessenv = \lambda i.\lambda(map, l).\, map(i)$

$updateenv : Id \rightarrow Denotable\text{-}value \rightarrow Environment \rightarrow Environment$
$updateenv = \lambda i.\lambda d.\lambda(map, l).\, ([\, i \mapsto d\,]map, l)$

$reserve\text{-}locn : Environment \rightarrow (Location \times Environment)$
$reserve\text{-}locn = \lambda(map, l).\, (l, (map, next\text{-}locn(l)))$

VIII. Storable values
Domain $v \in Storable\text{-}value = Nat$

IX. Stores
Domain $s \in Store = Location \rightarrow Storable\text{-}value$
Operations

$access : Location \rightarrow Store \rightarrow Storable\text{-}value$
$access = \lambda l.\lambda s.\, s(l)$

$update : Location \rightarrow Storable\text{-}value \rightarrow Store \rightarrow Store$
$update = \lambda l.\lambda v.\lambda s.\, [\, l \mapsto v\,]s$

X. Run-time store, labeled with status of computation
Domain $p \in Poststore = OK + Err$
where $OK = Err = Store$

Figure 7.1 (continued)

Operations

$return : Store \rightarrow Poststore$

$return = \lambda s.\ \mathrm{in}OK(s)$

$signalerr : Store \rightarrow Poststore$

$signalerr = \lambda s.\ \mathrm{in}Err(s)$

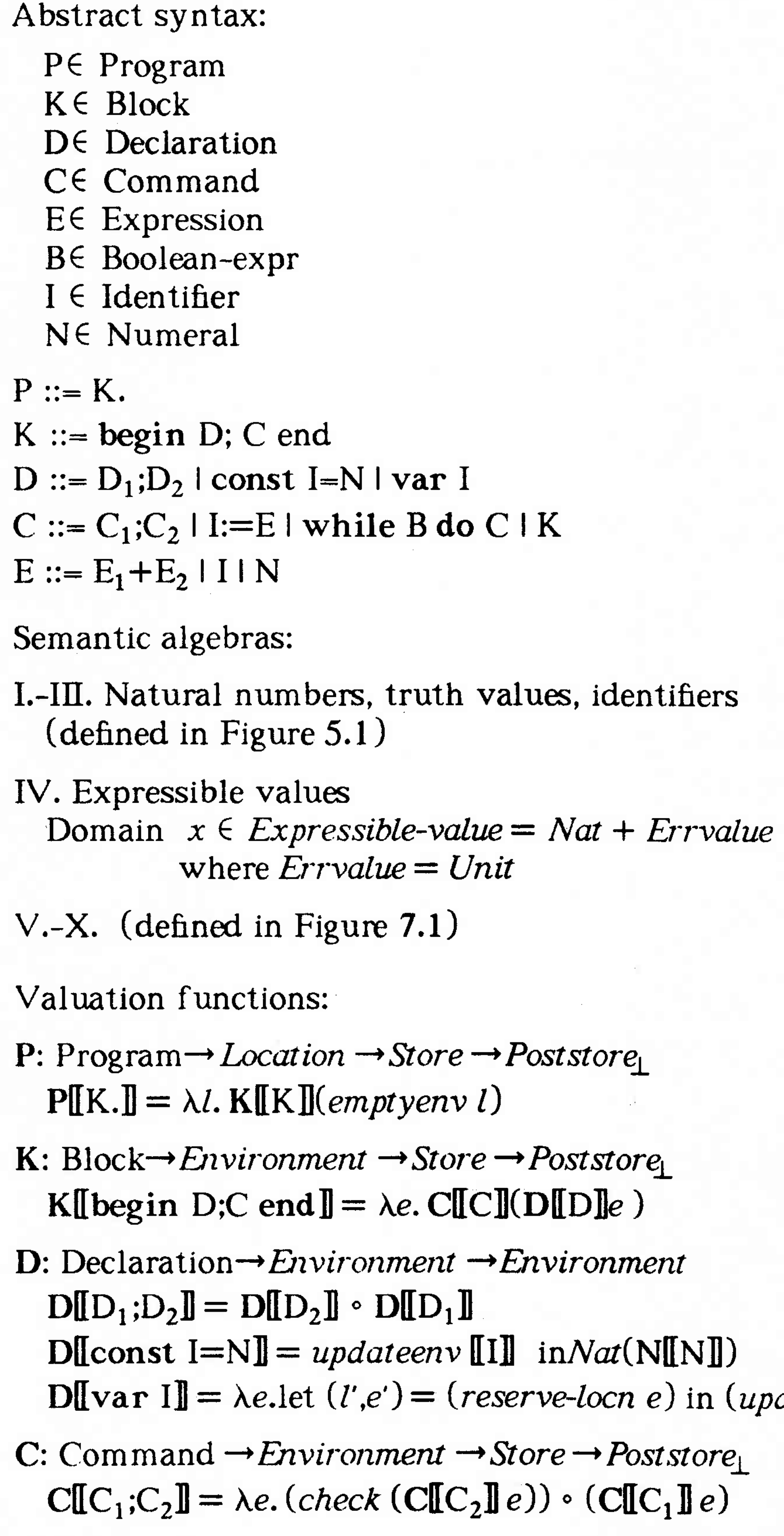
$check : (Store \rightarrow Poststore_{\perp}) \rightarrow (Poststore_{\perp} \rightarrow Poststore_{\perp})$

$check\ f = \underline{\lambda} p.\ \mathrm{cases}\ p\ \mathrm{of}$

$\quad \mathrm{is}OK(s) \rightarrow (f\ s)$

$\quad [\,]\ \mathrm{is}Err(s) \rightarrow p\ \mathrm{end}$

Figure 7.2



Abstract syntax:

P∈ Program
K∈ Block
D∈ Declaration
C∈ Command
E∈ Expression
B∈ Boolean-expr
I ∈ Identifier
N∈ Numeral

P ::= K.
K ::= **begin** D; C **end**
D ::= D_1;D_2 | **const** I=N | **var** I
C ::= C_1;C_2 | I:=E | **while** B **do** C | K
E ::= E_1+E_2 | I | N

Semantic algebras:

I.-III. Natural numbers, truth values, identifiers
(defined in Figure 5.1)

IV. Expressible values
Domain $x \in$ *Expressible-value* = *Nat* + *Errvalue*
where *Errvalue* = *Unit*

V.-X. (defined in Figure 7.1)

Valuation functions:

P: Program→*Location*→*Store*→$Poststore_{\perp}$
P[[K.]] = λ*l*. **K**[[K]](*emptyenv l*)

K: Block→*Environment*→*Store*→$Poststore_{\perp}$
K[[**begin** D;C **end**]] = λ*e*. **C**[[C]](**D**[[D]]*e*)

D: Declaration→*Environment*→*Environment*
D[[D_1;D_2]] = **D**[[D_2]] ∘ **D**[[D_1]]
D[[**const** I=N]] = *updateenv* [[I]] in*Nat*(**N**[[N]])
D[[**var** I]] = λ*e*.let (*l*′,*e*′) = (*reserve-locn e*) in (*updateenv* [[I]] in*Location*(*l*′) *e*′)

C: Command→*Environment*→*Store*→$Poststore_{\perp}$
C[[C_1;C_2]] = λ*e*. (*check* (**C**[[C_2]]*e*)) ∘ (**C**[[C_1]]*e*)

Figure 7.2 (continued)

```
C[[I:=E]] = λe.λs. cases (accessenv [[I]] e) of
        isLocation(l)→(cases (E[[E]]e s) of
            isNat(n)→(return (update l n s))
            [] isErrvalue()→(signalerr s) end)
        [] isNat(n)→(signalerr s)
        [] isErrvalue()→(signalerr s) end
C[[while B do C]] = λe. fix(λf.λs. cases (B[[B]]e s) of
        isTr(t)→(t →(check f) ∘ (C[[C]]e) [] return )(s)
        [] isErrvalue()→(signalerr s) end)
C[[K]] = K[[K]]

E: Expression → Environment → Store → Expressible-value
E[[E1+E2]] = λe.λs. cases (E[[E1]]e s) of
        isNat(n1)→(cases (E[[E2]]e s) of
            isNat(n2)→ inNat(n1 plus n2)
            [] isErrvalue()→ inErrvalue() end)
        [] isErrvalue()→ inErrvalue() end
E[[I]] = λe.λs. cases (accessenv [[I]] e) of
        isLocation(l)→ inNat(access l s)
        [] isNat(n)→ inNat(n)
        [] isErrvalue()→ inErrvalue() end
E[[N]] = λe.λs. inNat(N[[N]])

B: Boolean-expr → Environment → Store → (Tr + Errvalue)  (omitted)

N: Numeral → Nat  (omitted)
```

local environment to process its commands. However, C⟦C_2⟧ retains its own copy of *e*, so the environments created within C⟦C_1⟧ do not affect C⟦C_2⟧'s version. (Of course, whatever alterations C⟦C_1⟧*e* makes upon the store are passed to C⟦C_2⟧*e*.) This language feature is called *static scoping*. The context for a phrase in a statically scoped language is determined solely by the textual position of the phrase. One of the benefits of static scoping is that any identifier declared within a block may be referenced only by the commands within that block. This makes the management of storage locations straightforward. So-called *dynamically scoped* languages, whose contexts are not totally determined

Figure 7.3

Let D_0 = const A=1
D_1 = var X
$C_0 = C_1;C_2;C_3$
C_1 = X:=A+2
C_2 = **begin var** A;C_4 **end**
C_3 = X:=A
C_4 = **while** X=0 **do** A:=X

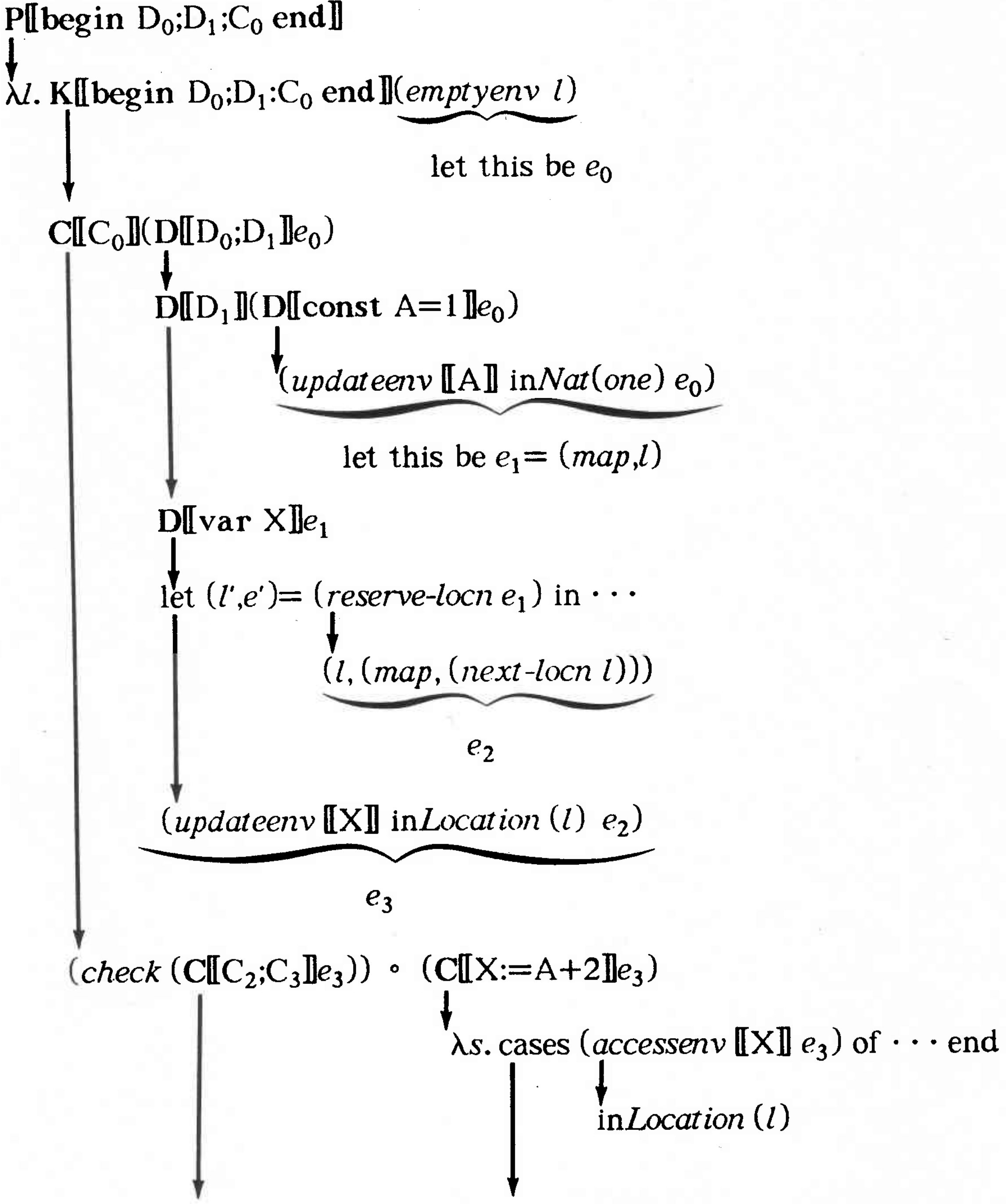


Figure 7.3 (continued)

7.1.1 Stack-Managed Storage

The store of a block-structured language is used in a stack-like fashion— locations are bound to identifiers sequentially using *nextlocn*, and a location bound to an identifier in a local block is freed for re-use when the block is exited. The re-use of locations happens automatically due to the equation for $\mathbf{C}[\![C_1;C_2]\!]$. Any locations bound to identifiers in $[\![C_1]\!]$ are reserved by the environment built from e for $\mathbf{C}[\![C_1]\!]$, but $\mathbf{C}[\![C_2]\!]$ re-uses the original e (and its original location marker), effectively deallocating the locations.

Figure 7.4

IX′. Stack-based store

Domain $\mathit{Store} = (\mathit{Location} \rightarrow \mathit{Storable\text{-}value}) \times \mathit{Location}$

Operations

$$\mathit{access} : \mathit{Location} \rightarrow \mathit{Store} \rightarrow (\mathit{Storable\text{-}value} + \mathit{Errvalue})$$

$$\mathit{access} = \lambda l.\lambda(\mathit{map}, \mathit{top}).\ l\ \mathit{lessthan\text{-}locn}\ \mathit{top} \rightarrow \mathrm{in}\mathit{Storable\text{-}value}(\mathit{map}\ l)\ \square\ \mathrm{in}\mathit{Errvalue}()$$

$$\mathit{update} : \mathit{Location} \rightarrow \mathit{Storable\text{-}value} \rightarrow \mathit{Store} \rightarrow \mathit{Poststore}$$

$$\mathit{update} = \lambda l.\lambda v.\lambda(\mathit{map}, \mathit{top}).\ l\ \mathit{lessthan\text{-}locn}\ \mathit{top} \rightarrow \mathrm{in}\mathit{OK}([l \mapsto v]\mathit{map}, \mathit{top})\ \square\ \mathrm{in}\mathit{Err}(\mathit{map}, \mathit{top})$$

$$\mathit{mark\text{-}locn} : \mathit{Store} \rightarrow \mathit{Location}$$

$$\mathit{mark\text{-}locn} = \lambda(\mathit{map}, \mathit{top}).\ \mathit{top}$$

$$\mathit{allocate\text{-}locn} : \mathit{Store} \rightarrow \mathit{Location} \times \mathit{Poststore}$$

$$\mathit{allocate\text{-}locn} = \lambda(\mathit{map}, \mathit{top}).\ (\mathit{top}, \mathrm{in}\mathit{OK}(\mathit{map}, \mathit{next\text{-}locn}(\mathit{top})))$$

$$\mathit{deallocate\text{-}locns} : \mathit{Location} \rightarrow \mathit{Store} \rightarrow \mathit{Poststore}$$

$$\mathit{deallocate\text{-}locns} = \lambda l.\lambda(\mathit{map}, \mathit{top}).\ (l\ \mathit{lessthan\text{-}locn}\ \mathit{top})\ \mathit{or}\ (l\ \mathit{equal\text{-}locn}\ \mathit{top}) \rightarrow \mathrm{in}\mathit{OK}(\mathit{map}, l)\ \square\ \mathrm{in}\mathit{Err}(\mathit{map}, \mathit{top})$$