Compound Data Structures

- Structural decomposition into other values.
- Lists: domain A*
  - Constructors: NIL, CONS.
  - Selectors: HEAD, TAIL.
- Tuples: domain A x B x C …
  - Constructor: (…, …, …)
  - Selector: $\downarrow$ i
- Problem: imperative languages
  - Variable forms of the objects exist.
  - Objects subcomponents can be altered.
- Several versions of arrays variables.

Arrays

- Collection of homogeneous objects indexed by a set of scalar values.
- Homogeneity $\Rightarrow$ all components have same structure.
- Allocation of storage and type-checking easy.
- Both tasks can be performed by compiler.
- Selector operation: indexing.
  - Scalar index set: primitive domain with relational and arithmetic operations.
  - Restricted by lower and upper bounds.

Linear Vector of Values

- IDArray = (Index $\rightarrow$ Location) x Lower-bound x Upper-Bound
- Index is index set
  - lessthan, greaterthan, equals.
- Lower-bound = Upper-bound = Index.
- First component maps indices to the locations that contain the storable values.
- Second and third component denote the bounds allowed on array indices.

Multidimensional Arrays

- Array may contain other arrays.
- Three-dimensional array is vector whose components are two-dimensional vectors.
- Hierarchy of arrays defined as infinite sum
  - 1DArray = (Index $\rightarrow$ Location) x Index x Index.
  - (n+1)DArray = (Index $\rightarrow$ nDArray) x Index x Index.
- 1DArray maps indices to locations, 2DArray maps indices to 1D arrays, ...

Multidimensional Arrays

- a $\in$ MDArray =
  inkDArray(map, lower, upper) for some k >= 1
- access-array:
  Index $\rightarrow$ MDArray $\rightarrow$(Location + MDArray + Errvalue)
- access-array = λi. λr. cases r of
  is1DArray(a) $\rightarrow$ index_1 a i
  [] is2DArray(a) $\rightarrow$ index_2 a i
  . . .
  [] iskDArray(a) $\rightarrow$ index_K a i
  . . .
  end

index $_m$ = λ(map, lower, upper). λ i.
(i lessthan lower) or (i greaterthan upper) $\rightarrow$
inErrvalue() [] MInject(map(i))

1Inject = λ l.inLocation(l)
. . .
(n+1)Inject = λ a. inMDArray(innDArray(a))

Multidimensional Arrays

- access-array is represented by an infinite function expression.
- By using pair representation of disjoint union elements, operation is convertible to finite, computable format.
- Operation performs one-level indexing upon array a returning another array if a has more than one dimension.
- Still model is to clumsy to be used in practice.
- Real programming languages allow arrays of numbers, record structures, sets, . . . !

System of Type Declarations

T ∈ Type-structure
S ∈ Subscript
T ::= nat| bool| array [N_1... N_2] of T |record D end
D ::= D_1;D_2|var I:T
C ::= …| I[S]:= E |...
E ::= ... | I[S] |...
S ::= E | E, S

Denotable-value =
(Natlocn + Boollocn + Array + Record + Errvalue)⊥

l ∈ Natlocn = Boollocn = Location
a ∈ Array = (Nat → Denotable-value) x Nat x Nat
r ∈ Record = Environment = Id → Denotable-value

Semantics of Type Declarations

- T: Type-structure → Store → (Denotable-value x Poststore)
- T[[nat]] = λ s:
  let (l,p) = (allocate-locn s)
  in (inNatlocn(l), p)
- T[[bool]] = λ s:
  let (l,p) = (allocate-locn s)
  in (inBoollocn(l), p)

- T[[array [N_1... N_2]of T]] = λ s:
  let n_1 = N[[N_1]] in let n_2 = N[[N_2]]
  in n_1 greaterthan n_2 (inErrvalue(), (signalerr s))
  [] get-storage n_1 (empty-array n_1 n_2) s
- T[[record D end]] = λ s: let (e, p) = (D[[D]] emptyenv s)
  in (inRecord(e), p)

Type structure expressions are mapped to storage allocation actions!

Semantics of Type Declarations

- get-storage: Nat → Array → Store → (Denotable-value x Poststore)
- get-storage = λ n: λ a: λ s: n greater n 2 →
  (inArray(a), return s)
  [] let (d, p) = T[[T]]s
  in (check(get-storage (n plus one)
  (augment-array n d a)))(p)

- augment-array: Nat → Denotable-value → Array → Array
- augment-array = λ n: λ d: λ(map, lower, upper):
  ([n |→ d] map, lower, upper)

- empty-array: Nat → Nat → Array
- empty-array = λ n1 : λ n2 :((λ n:inErrvalue()); n1 ; n2)

- get-storage iterates from lower bound of array to upper bound allocating the proper amount of storage for a component.
- augment-array inserts the component into the array.

Declarations

- D: Declaration → Environment → Store →
  (Environment x Poststore)
- D[[D1;D2]] = λ e: λ s:
  let (e', p') = (D[[D1]]e s) in (check (D[[D2]]e'))(p)

- D[[var I:T]] = λ e: λ s:
  let (d, p) = T[[T]]s in ((updateenv [[I]] d e), p)

- A declaration activates the storage allocation strategy specified by its type structure.

Array Indexing

S: Subscript →Array → Environment → Store → Storable-value

S[[E]] = λ a:λe: λ s:

cases (E[[E]]e s) of . . .

[] isNat(n) access-array n a … end

S[[E, S]] = λ a: λ e: λ s:

cases (E[[E]]e s) of

… [] isNat(n) → (cases (access-array n a) of

… [] isArray(a') → S[[S]]a' e s ...end) ...

end

Array Assignment

C[[I[S] := E]] = λ e: λ s:

cases (accessenv [[I]] e) of ...

[] isArray(a) → (cases (S[[S]]a e s) of ...

isNatlocn(l) →(cases (E[[E]]e s) of

... [] isNat(n) → return(update l inNat(n) s) ...end)

...end)

...end

Assignment is first order (location, not an array, is on left-hand-side).