10. Languages with Contexts III: Compound Data Structures

Wolfgang Schreiner RISC-Linz

Compound Data Structures

- Structural decomposition into other values.
- Lists: domain A^*
  - *Constructors: NIL, CONS.*
  - *Selectors: HEAD, TAIL.*
- Tuples: domain $A \times B \times C \ldots$
  - *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 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

$IDArray = (Index \rightarrow Location) \times$
$Lower\text{-}bound \times Upper\text{-}Bound$

- *Index* is index set

  lessthan, greaterthan, equals.

- $Lower\text{-}bound = Upper\text{-}bound = Index$.
- 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
  - $\mathit{1DArray} = (\mathit{Index} \to \mathit{Location}) \times \mathit{Index} \times \mathit{Index}$.
  - $(n{+}1)\mathit{DArray} = (\mathit{Index} \to n\mathit{DArray}) \times \mathit{Index} \times \mathit{Index}$.

  $a \in \mathit{MDArray} = \Sigma_{m=1}^{\infty}\ m\mathit{DArray}$
  $= ((\mathit{Index} \to \mathit{Location}) \times \mathit{Index} \times \mathit{Index})$
  $+ ((\mathit{Index} \to ((\mathit{Index} \to \mathit{Location}) \times \mathit{Index} \times \mathit{Index})) \times \mathit{Index} \times \mathit{Index})$
  $+ ((\mathit{Index} \to ((\mathit{Index} \to ((\mathit{Index} \to \mathit{Location}) \times \mathit{Index} \times \mathit{Index})) \times \mathit{Index} \times \mathit{Index})) \times \mathit{Index} \times \mathit{Index}) + \ldots$

- *1DArray* maps indices to locations, *2DArray* maps indices to 1D arrays, ...

Multidimensional Arrays

a ∈ *MDArray* = in*kDArray(map, lower, upper)*
for some $k \geq 1$

access-array: Index → *MDArray* →
(Location + MDArray + Errvalue)
access-array = $\lambda i.\lambda r.$ cases r of
is*1DArray*(a) → *index*$_1$ a i
[] is*2DArray*(a) → *index*$_2$ a i
...
[] is*kDArray*(a) → *index*$_k$ a i
...
end

index$_m$ = λ*(map, lower, upper)*.$\lambda i.$
(i *lessthan lower) or* (i *greaterthan upper)* →
inErrvalue() [] *mInject(map*(i*))*

1Inject = $\lambda l.$in*Location*(l)
...
(n+1)Inject = $\lambda a.$ in*MDArray*(in*nDArray*(a))

Multidimensional Arrays

- *access-array* is represented by 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)*$_\perp$

$l \in$ *Natlocn* = *Boollocn* = *Location*
$a \in$ *Array* = *(Nat* → *Denotable-value)* ×
    Nat × *Nat*
$r \in$ *Record* = *Environment* =
    Id → *Denotable-value*

Semantics of Type Declarations

T: Type-structure $\rightarrow$ *Store* $\rightarrow$
(*Denotable-value* $\times$ *Poststore*)
T[[**nat**]] $= \lambda s.$
let $(l,p) =$ (*allocate-locn* s)
in (in*Natlocn*(l), p)
T[[**bool**]] $= \lambda s.$
let $(l,p) =$ (*allocate-locn* s)
in (in*Boollocn*(l), p)
T[[**array** [N_1 ... N_2] **of** T]] $= \lambda s.$
let $n_1 =$ **N**[[N_1]] in let $n_2 =$ **N**[[N_2]]
in n_1 *greaterthan* $n_2 \rightarrow$
(in*Errvalue*(), (*signalerr* s))
[] *get-storage* n_1 (*empty-array* n_1 n_2) s
T[[**record** D **end**]] $= \lambda s.$
let $(e,\ p) =$ (**D**[[D]] *emptyenv* s)
in (in*Record*(e), p)

Type structure expressions are mapped to storage allocation actions!

Semantics of Type Declarations

get-storage: Nat $\rightarrow$ *Array* $\rightarrow$ *Store* $\rightarrow$
    (Denotable-value $\times$ *Poststore)*
get-storage $= \lambda n.\lambda a.\lambda s.$ n *greater* $n_2 \rightarrow$
    (in*Array*(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 $\rightarrow$ *Denotable-value* $\rightarrow$
    Array $\rightarrow$ *Array*
augment-array $= \lambda n.\lambda d.\lambda($*map, lower, upper*$).$
    $([n \mapsto d]$*map, lower, upper)*
empty-array: Nat $\rightarrow$ *Nat* $\rightarrow$ *Array*
empty-array $= \lambda n_1.\lambda n_2.((\lambda n.$in*Errvalue*$()), n_1, n_2)$

- *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 $\rightarrow$ *Environment* $\rightarrow$ *Store* $\rightarrow$
(*Environment* $\times$ *Poststore*)
D[[D_1; D_2]] $= \lambda e.\lambda s.$
let $(e', p') =$ (**D**[[D_1]]$e\ s$)
in (*check* (**D**[[D_2]]e'))(p)
D[[**var** I:T]] $= \lambda e.\lambda s.$
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]] = $\lambda a.\lambda e.\lambda s.$
cases (**E**[[E]]$e\ s$) of
. . .
[] is*Nat*(n) → *access-array* $n\ a$
. . .
end
S[[E, S]] = $\lambda a.\lambda e.\lambda s.$
cases (**E**[[E]]$e\ s$) of
. . .
[] is*Nat*(n) →
(cases (*access-array* $n\ a$) of
. . .
[] is*Array*(a') → **S**[[S]]$a'\ e\ s$
. . .
end)
. . .
end

Array Assignment

```
C[[I[S] := E]] = λe.λs.
    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).

Heterogeneous Arrays

- Components can be elements of different structures,
- Dimension and index range can change during execution,
- Array can possess itself as an element.
- Pre-execution analysis is not possible.

Late binding languages as APL and SNOBOL.

Heterogeneous Arrays

access-value: $Nat^* \to$ *Denotable-value* $\to$
    Denotable-value
access-value $= \lambda$*nlist*.λd.
    null nlist $\to d$
    [] (cases d of
        is*Natlocn*(l) $\to$ in*Errvalue*()
        ...
        [] is*Array*(*map*, *lower*, *upper*) $\to$
            let $n =$ *hd nlist*
            in (n *lessthan lower*)
                or (n *greaterthan upper*) $\to$
                in*Errvalue*() []
                (*access-value* (*tl nlist*) (*map* n))
        ...
    end)

- Index list denotes path to component,
- Structure searched for component,
- Empty index signifies end of search.

Heterogeneous Arrays

```
update-value: Nat* → Denotable-value →
    Denotable-value → Denotable-value
update-value = λnlist. λnew-value. λcurrent-value.
    null nlist → new-value
    [] (cases current-value of
        isNatlocn(l) → inErrvalue()
        ...
        [] isArray(map, l, u) →
            let n = hd nlist in
            let lnew = (n lt l → n [] l) in
            let unew = (n gt u → n [] u)
            in augment-array n (update-value
                (tl nlist) new-value (map n))
                (map, lnew, unew)
        ...
        [] isErrvalue() → augment-array n
            (update-value (tl nlist) new-value
            inErrvalue()) (empty-array n n)
        ...
    end)
```

Heterogeneous Arrays

augment-array: *Nat* $\rightarrow$ *Denotable-value* $\rightarrow$ *Array* $\rightarrow$
Denotable-value
augment-array $= \lambda n.\lambda d.\lambda($*map*, *lower*, *upper*$)$.
in*Array*$([i \mapsto d]$*map*, *lower*, *upper*$)$

- Array can grow extra dimensions,
- Array indices can be expanded,
- Outer structure of array is preserved by *augment-array*.