Qube

Array Programming with Dependent Types

Kai Trojahner

DTP 2011

Array Programming at a Glance

- Generalization of scalar operations

$$1 + 2 \Rightarrow 3$$

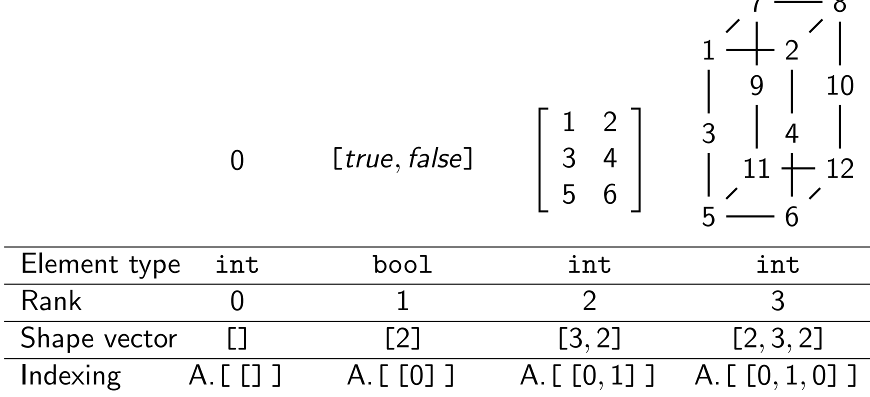
$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \end{bmatrix} \Rightarrow \begin{bmatrix} 2 & 3 & 4 \\ 6 & 7 & 8 \end{bmatrix}$$

- Compact programs without loops (GOL in APL, Wikipedia)

```
life←{↑1 ⍵∨.∧3 4=+/,¯1 0 1∘.⊖¯1 0 1∘.⌽⊂⍵}
```

- Implicitly data-parallel
- **APL** (Iverson, 1957), **MATLAB**, ...
  predefined array primitives, interpreted, untyped
- **SAC**, **Qube**, ...
  user-defined array operations, compiled, typed

Multidimensional Arrays



	0	[*true*, *false*]	$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$	(cube 1–12)
Element type	`int`	`bool`	`int`	`int`
Rank	0	1	2	3
Shape vector	[]	[2]	[3, 2]	[2, 3, 2]
Indexing	A.[[]]	A.[[0]]	A.[[0, 1]]	A.[[0, 1, 0]]

Array-Specific Program Errors

1. Incompatible array elements

   `[1,2] + [true,false]` ⇝ `1 + true`

2. Rank error

   + ⇝ `.[ [1,0] ]`

3. Shape error

   + ⇝ `.[ [0,2] ]`

Qube

- Functional array programming
  - OCaml-like syntax
  - Call-by-value semantics
  - Compiles to LLVM
  - Rank-generic programming
  - Easy to use
- Dependent types
  - Precise specifications
  - Type safe array programs
  - SMT-based checking

Typed Rank-Generic Programming in **Qube**

Example (Array addition)

```
let add r:nat s:(natvec r) a:[int|s] b:[int|s] =
  gen s with x -> a.[x] + b.[x]
```

```
add  0  []
add  2  [2,3]
```

$\Rightarrow$

$\Rightarrow$

Language Design

`add 2 [2,3]` ▦ ▦ $\Rightarrow$ ▦

Key observation

- Multidimensional arrays are parametrized by integer vectors.
- Vectors are parametrized by integers.

Qube comprises three layers

3. $\textbf{Qube}_{[]}$: Multidimensional arrays
2. $\textbf{Qube}_{\rightarrow}$: Integer vectors
1. $\textbf{Qube}_{\lambda}$: Applied λ-calculus with dependent types

Qube$_\lambda$

- Applied λ-calculus
  fun $x : T \to e$, if e then e else e, let $x = e$ in e,...
- Refinement type $\{x : T \mid e\}$

```
let z = 0
val z : { v:int | v = 0 }

type nat = { x:int | 0 <= x }

type index u:int = { x:int | 0<=x & x<u }
```

- Dependent function type $x : T \to T_x$

```
val + : x:int -> y:int -> { v:int | v = x+y }
```

$\text{Qube}_{\rightarrow}$

- Integer vector type `intvec` *e*

```
let a = [1,2,3]
val a : { v:intvec 3 | a.(0)=1 & a.(1)=2 & a.(2)=3 }
```

- Vector predicate `vfa` $e, .., e\ (x, .., x \rightarrow e)$

```
type natvec n:nat = { x:intvec n | vfa x (xi -> 0<=xi) }

type indexvec n:nat s:(intvec n) =
     { x:intvec n | vfa x,s (xi,si -> 0<=xi & xi<si) }
```

- Vector comprehension `vmap` $e, .., e\ (x, .., x \rightarrow e)$

```
let c = vmap a,b (ai,bi -> ai+bi)
val c : { v:intvec 3 | vfa v,a,b (vi,ai,bi -> vi=ai+bi) }
```

Qube$_{[]}$

- Array type $[T\,|\,e, .., e]$

```
let a = [1,2,3,4,5,6|[2,3]]
val a : [int|[2,3]]

[int|[2,3]] ≡ [int|[2],[3]]
```

- Selection $e.[e, .., e]$

```
let a0 = a.[[0,0]]
let a0 = a.[[0],[0]]
```

- Array comprehension gen $e, .., e$ with $x, .., x \to e$

```
let a = gen [2,3] with x -> x.(0)+3*x.(1)
let a = gen [2],[3] with x,y -> x.(0)+3*y.(0)
```

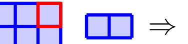
- Array reduction loop $x = e;\ e, .., e$ with $x, .., x \to e$

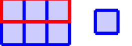
```
let sum = loop s = 0; [2,3] with x -> s + a.[x]
let sum = loop s = 0; [2],[3] with x,y -> s + a.[x,y]
```

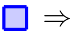
Generalized Selection

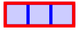
Example (Generalized Selection)

```
let gsel fr:nat fs:(natvec fr) cr:nat cs:(natvec cr)
         a:[int|fs,cs] x:(indexvec fr fs) =
  gen cs with y -> a.[x,y]
```





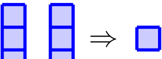


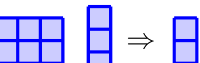


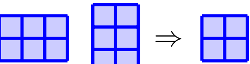
Inner Product

Example (Inner Product)

```
let ip m:nat n:nat r:(natvec m) s:nat t:(natvec n)
       a:[double|r,[s]] b:[double|[s],t]  =
   gen r,t with i,j ->
     loop sum = 0; [s] with [k] ->
       sum + a.[i,[k]] * b.[[k],j]
```







Discrete Convolution

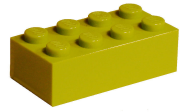
$$\left[(f * g)(i) = \sum_{j=0}^{m}(f(i+j) \cdot g(m-j))\right]_{i=0}^{n-m+1}$$

Example (Rank-generic convolution)

```
let convolve r:nat n:(natvec r) f:[double|n]
             m:(natvec_le r n) g:[double|m] =
  let g' = reverse r m g in
  gen vmap n,m (na,ma -> na-ma+1) with i ->
    loop sum:double = 0.; m with j ->
      sum + f.[vmap i,j (ia,ja -> ia+ja)] * g'.[y]
```

Programming in **Qube**

- User-defined operations as building blocks
  - Generalized scalar operations
    `add`, `sub`, `square`, `sum`
  - Structural operations
    `gsel`, `take`, `drop`, `cat`, `shift`, `rotate`
  - Higher-order array operations
    `map`, `fold_left`, `ip`, `op`
- Precise specifications
- Machine checked implementations
- Composition yields correct programs



```
let ssd r:nat s:(natvec r) a:[int|s] b:[int|s] =
  sum r s (square r s (sub r s a b))
```

Illegal Function Application

Example

```
let add r:nat s:(natvec r) a:[int|s] b:[int|s] =
  gen s with x -> a.[ x ] + b.[ x ]

let _ = add 2 [2,3] [1,2,3,4,5,6|[2,3]] [true,false]
```

```
Type error in file add.q, line 4, column 40:
Expression has type [bool|[2]]
but is used here with type [int|[2,3]]
```

Array Bounds Violation

Example

```
let add_1d n:nat a:[int|[n]] b:[int|[n]] =
  gen [n] with [i] -> a.[ [i] ] + b.[ [i+1] ]
```

```
Type error in file add.q, line 2, column 34:
Index may violate the array boundaries.
```

Type Checking with Dependent Types

- Dependent array types are not unique

$$\texttt{[int|[3]]} = \texttt{[int|[}1+2\texttt{]]} \stackrel{?}{=} \texttt{[int|[}f\ x\texttt{]]}$$

- Type equality corresponds to validity of expressions

$$\frac{\Gamma \vdash e = e'}{\Gamma \vdash [T \mid e] = [T \mid e']}$$

Validity checking in **Qube**

1. Soundly approximate $\Gamma \vdash e$ as first-order formula ϕ,
2. Use an SMT solver to refute $\neg\phi$.

Type Checking

Example

```
let add_1d n:nat a:[int|[n]] b:[int|[n]] =
  gen [n] with [i] -> a.[ [i] ] + b.[ [i+1] ]
```

- Type checking requires proof of
  `Γ ⊢ 0 <= i+1 & i+1 < n`
- First-order formula $\neg\phi$
  $0 \leq n \quad \wedge \quad (0 \leq i \ \wedge \ i < n) \quad \wedge \quad \neg(0 \leq i+1 \ \wedge \ i+1 < n)$
  satisfiable: $n = 1, i = 0 \rightarrow$ boundary violation

Type Checking

Example (Improved program)

```
let add_1d n:nat a:[int|[n]] b:[int|[n]] =
  gen [n] with [i] -> a.[ [i] ] + b.[ [i] ]
```

- Type checking requires proof of
  $\Gamma \vdash$ `0 <= i & i < n`
- First-order formula $\neg\phi$
  $0 \leq n \;\wedge\; (0 \leq i \;\wedge\; i < n) \;\wedge\; \neg(0 \leq i \;\wedge\; i < n)$
  unsatisfiable $\rightarrow$ expression ok

Type Checking Rank-Generic Programs

Example (Rank-generic addition)

```
let add r:nat s:(natvec r) a:[int|s] b:[int|s] =
  gen s with x -> a.[ x ] + b.[ x ]
```

- Type checking requires proof of
  `Γ ⊢ vfa x,s (xi,si -> 0 <= xi & xi < si)`
- First-order formula $\neg\phi$ contains quantifiers
  $\ldots \wedge \neg(\forall i.\, 0 \leq i < r \rightarrow 0 \leq x[i] \wedge x[i] < s[i])$
- *Array Property Fragment* (Bradley et al., 2006)
  Quantifier instantiation yields
  $\ldots \wedge \neg(\bigwedge_{i \in \mathcal{I}} 0 \leq i < r \rightarrow 0 \leq x[i] \wedge x[i] < s[i])$

Static Comparison of Structured Vectors

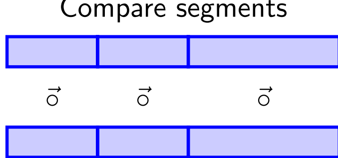
- Subtyping

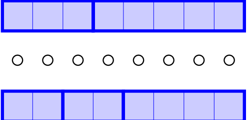
  `gsel 1 1 [2] [3] a [1]` with `a:[int|[2,3]]`:

  `[int|[2,3]]` $\equiv$ `[int|[2],[3]]`

- Selection

  `a.[x,y]` with `a:[int|fs,cs]`





Summary

Qube

- Functional array programming with dependent types
- Rank-generic programming
- Precise specifications
- Static checking based on SMT