A Compilation Scheme for a Hierarchy of Array Types

Dietmar Kreye
University of Kiel, Germany

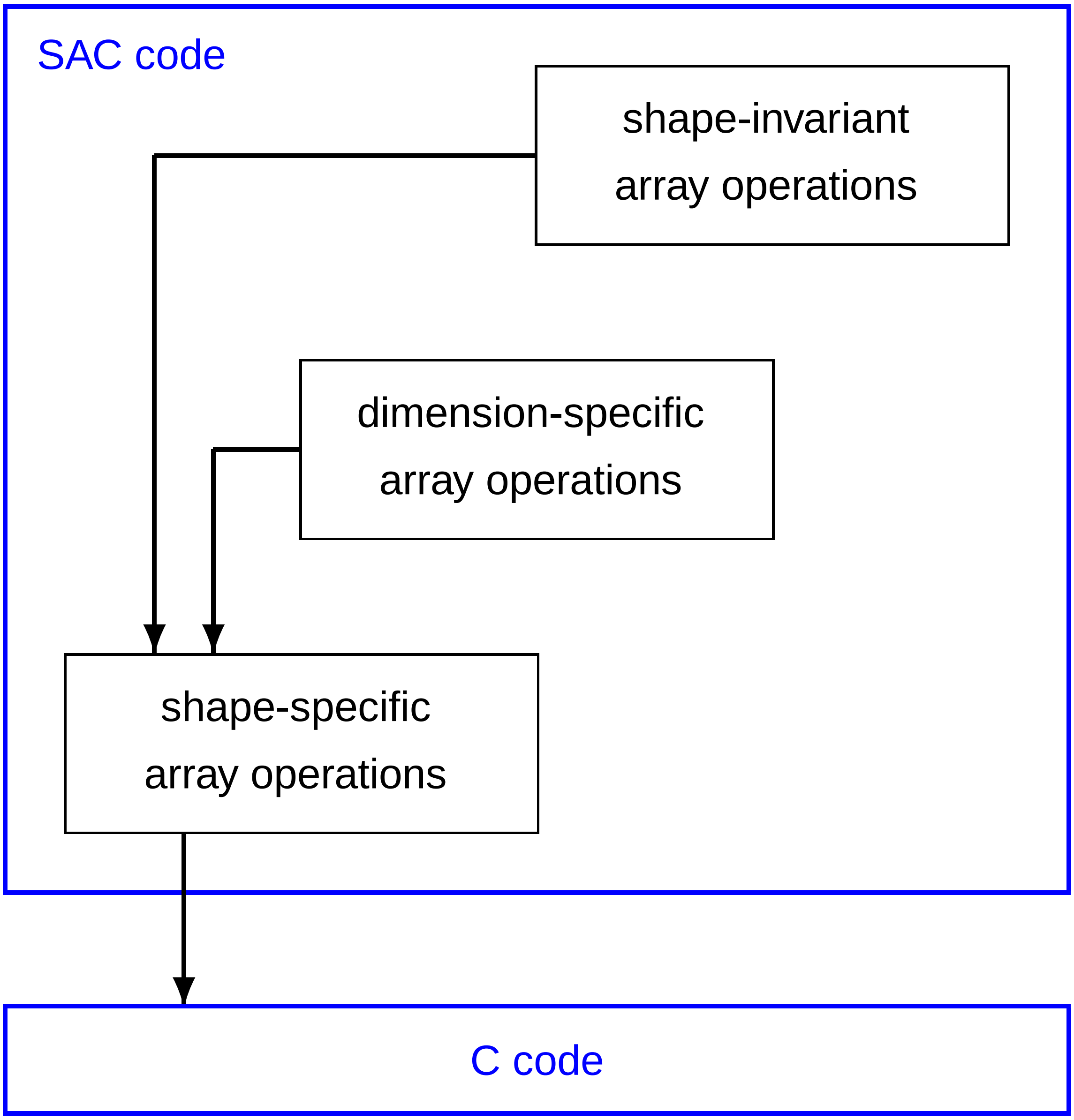
13th International Workshop on the
Implementation of Functional Languages

SAC: Compilation Process

SAC:

Strict functional array processing language based on C syntax.

➜ generic high-level program specification ←



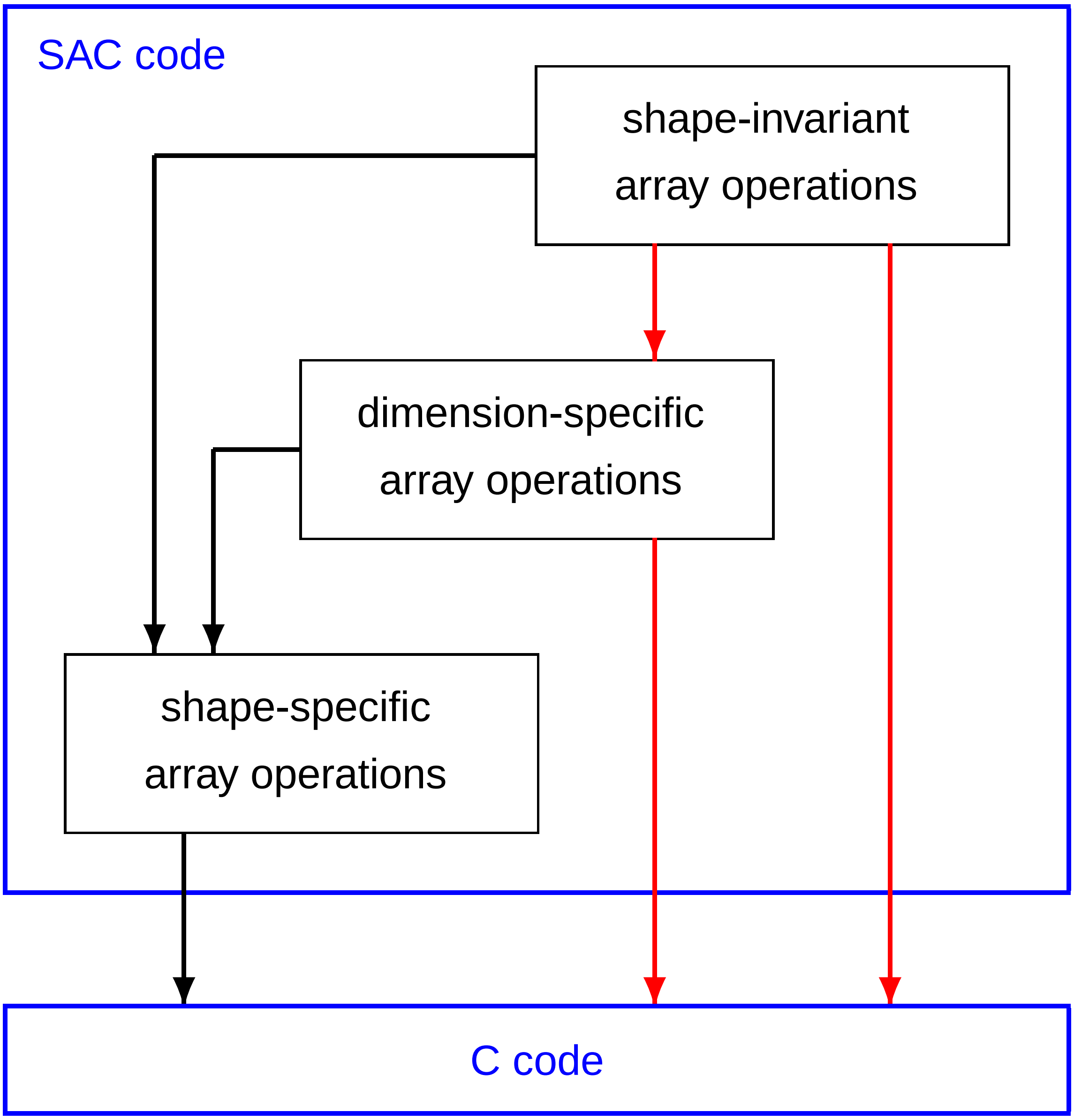
➜ high runtime performance ←

SAC: Compilation Process

SAC:

Strict functional array processing language based on C syntax.

➜ generic high-level program specification 🡐



➜ high runtime performance 🡐

Arrays in SAC: Representation

- 2×3 array:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

data vector: [1,2,3,4,5,6]
shape vector: [2,3]

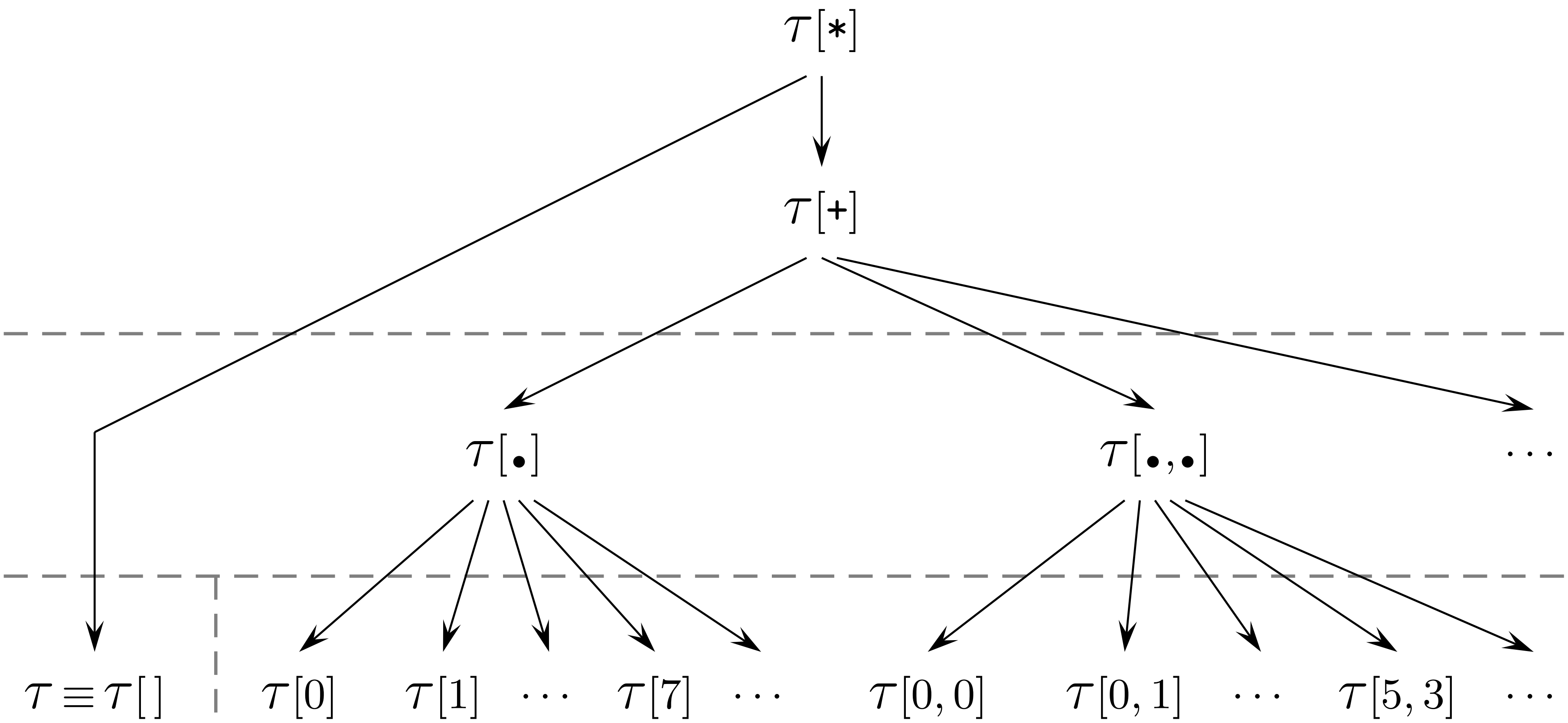
- Scalars are arrays with empty shape:

$$7$$

data vector: [7]
shape vector: [ ]

Arrays in SAC: Type System

SAC provides for each base type $\tau \in \{\texttt{int}, \texttt{float}, ...\}$ an entire hierarchy of array types:



Arrays in SAC: Operations

Primitive Array Operations:

- `dim(` A`)`
- `shape(` A`)`
- `sel(` A`,` idx`)` $\equiv$ A`[`idx`]`

`With`-loop Construct:

- sort of array comprehension
- allows for a shape-invariant definition of array operations

User-defined Array Operations:

- arbitrary number of parameters and return values
- overloading

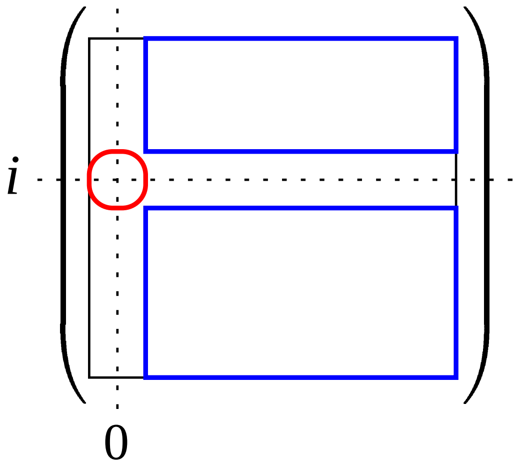
Example: Determinant of a 2-dimensional Array

$$\det \begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad - bc$$

```
int Det( int[2,2] A)
{
  return(   A[[0,0]] * A[[1,1]]
          - A[[0,1]] * A[[1,0]]);
}
```

Laplace expansion along the first column:

$$\det(A) = \sum_{i=0}^{n-1} (-1)^i \cdot A[[i,0]] \cdot \det(\mathcal{A}_{i0})$$




```
int Det( int[.,.] A)
{
  shp = shape( A);
  if (shp[[0]] == shp[[1]]) {
    ret = with ([0] <= [i] < [shp[[0]]]) {
                B = Elim( A, [i,0]);
                det = Det( B);
                val = (-1)^i * A[[i,0]] * det;
              } fold( +, val);
  } else {
    ret = ERROR( "array is not quadratic");
  }
  return( ret);
}
```

Compilation: Current Approach (1)

```
int Det( int[2,2] A)
{ ... }

int Det( int[.,.] A)
{
  shp = shape( A);
  if (shp[[0]] == shp[[1]]) {
    ... Det( B) ...
  } else { ... }
  return(...);
}

int main()
{
  int[3,3] A;
  ... Det( A) ...
}
```

→ Static Shape Inference →

Function Specialization →

```
int Det__i_2_2( int[2,2] A)
{ ... }

int Det__i_3_3( int[3,3] A)
{
  int[2,2] B;
  shp = shape( A);
  if (shp[[0]] == shp[[1]]) {
    ... Det__i_2_2( B) ...
  } else { ... }
  return(...);
}

int main()
{
  int[3,3] A;
  ... Det__i_3_3( A) ...
}
```

Compilation: Current Approach (2)

```
int Det__i_2_2( int[2,2] A)
{ ... }

int Det__i_3_3( int[3,3] A)
{
  int[2,2] B;
  shp = shape( A);
  if (shp[[0]] == shp[[1]]) {
    ... Det__i_2_2( B) ...
  } else { ... }
  return(...);
}

int main()
{
  int[3,3] A;
  ... Det__i_3_3( A) ...
}
```

High-level Code Optimizations

Constant Folding

Constant Propagation

...

```
int Det__i_2_2( int[2,2] A)
{ ... }

int Det__i_3_3( int[3,3] A)
{
  int[2,2] B;
  ... Det__i_2_2( B) ...
  return(...);
}

int main()
{
  int[3,3] A;
  ... Det__i_3_3( A) ...
}
```

<u>Pros and Cons of Current Approach</u>

<u>Pros:</u>

- All shapes statically known
  - ⇒ Best possible potential for code optimizations
  - ⇒ No backend support for [•,...], [+], [*] types needed
- Function overloading resolved statically
  - ⇒ Reduced runtime overhead

<u>Cons:</u>

- Static shape inference unfeasible in some cases:
  - ➢ Recursive functions whose argument shapes change with each recursive call
  - ➢ Input data have unknown shapes
  - ➢ Separate compilation of modules

<u>New Approach:</u>

- Integrate support for <u>all</u> array types
- Still infer shapes as precisely as possible
- Resolve function overloading statically wherever possible

Compilation: New Approach

```
int Det( int[2,2] A)
{ ... }

int Det( int[.,.] A)
{
  ... Det( B) ...
}

int main()
{
  int[3,3] A;
  int[+] B;
  ... Det( A) ...
  ... Det( B) ...
}
```

Static Shape Inference

Function Specialization

Resolution of Overloading

???

```
int Det__i_2_2( int[2,2] A) { ... }

int Det__i_X_X( int[.,.] A)
{ int[.,.] B;
  ... Det__i( B) ...
}

int Det__i_3_3( int[3,3] A)
{ int[2,2] B;
  ... Det__i_2_2( B) ...
}

int main()
{ int[3,3] A;
  int[+] B;
  ... Det__i_3_3( A) ...
  ... Det__i( B) ...
}
```

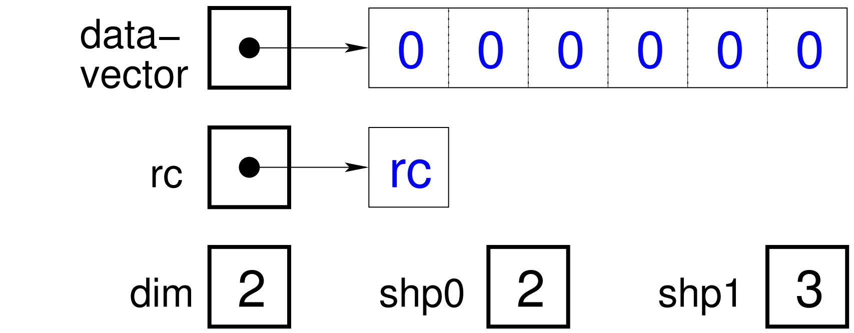
Compilation: Dispatch Function

```
inline
int Det__i( int[*] A)
{
  if (dim( A) == 2) {
    if (shape( A) == [2,2]) {
      ret = Det__i_2_2( A);
    } else if (shape( A) == [3,3]) {
      ret = Det__i_3_3( A);
    } else {
      ret = Det__i_X_X( A);
    }
  } else {
    ret = ERROR( "type error");
  }
  return( ret);
}
```

Code Generation: Array Representation

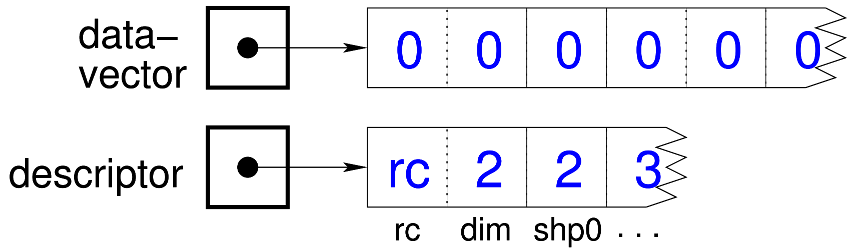
Current C Representation:

`int[2,3] A;`

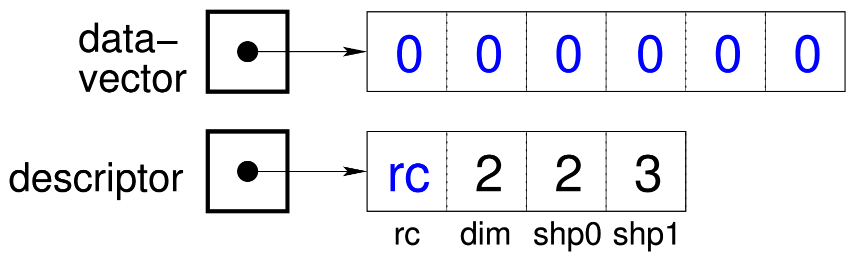


New C Representation:

`int[*] A;`



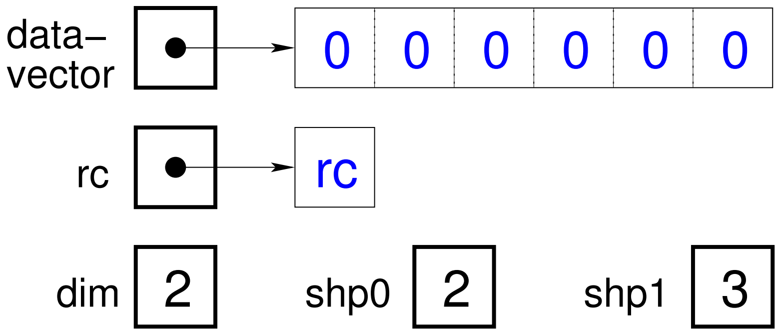
`int[2,3] B;`



Code Generation: Array Representation

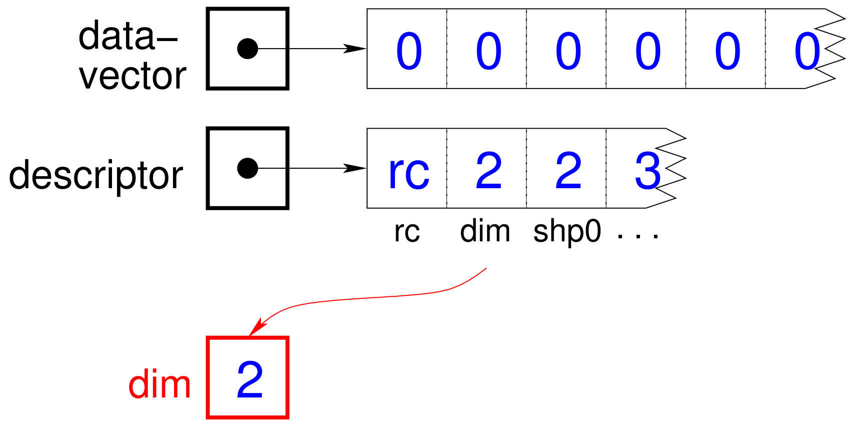
Current C Representation:

`int[2,3] A;`

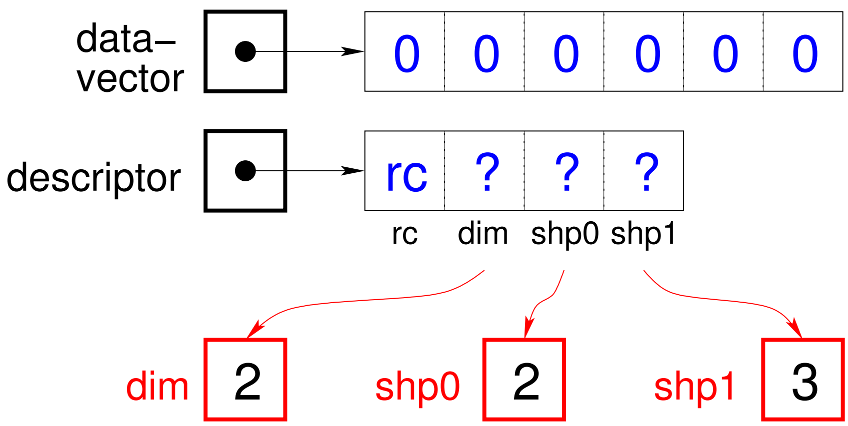


New C Representation:

`int[*] A;`



`int[2,3] B;`



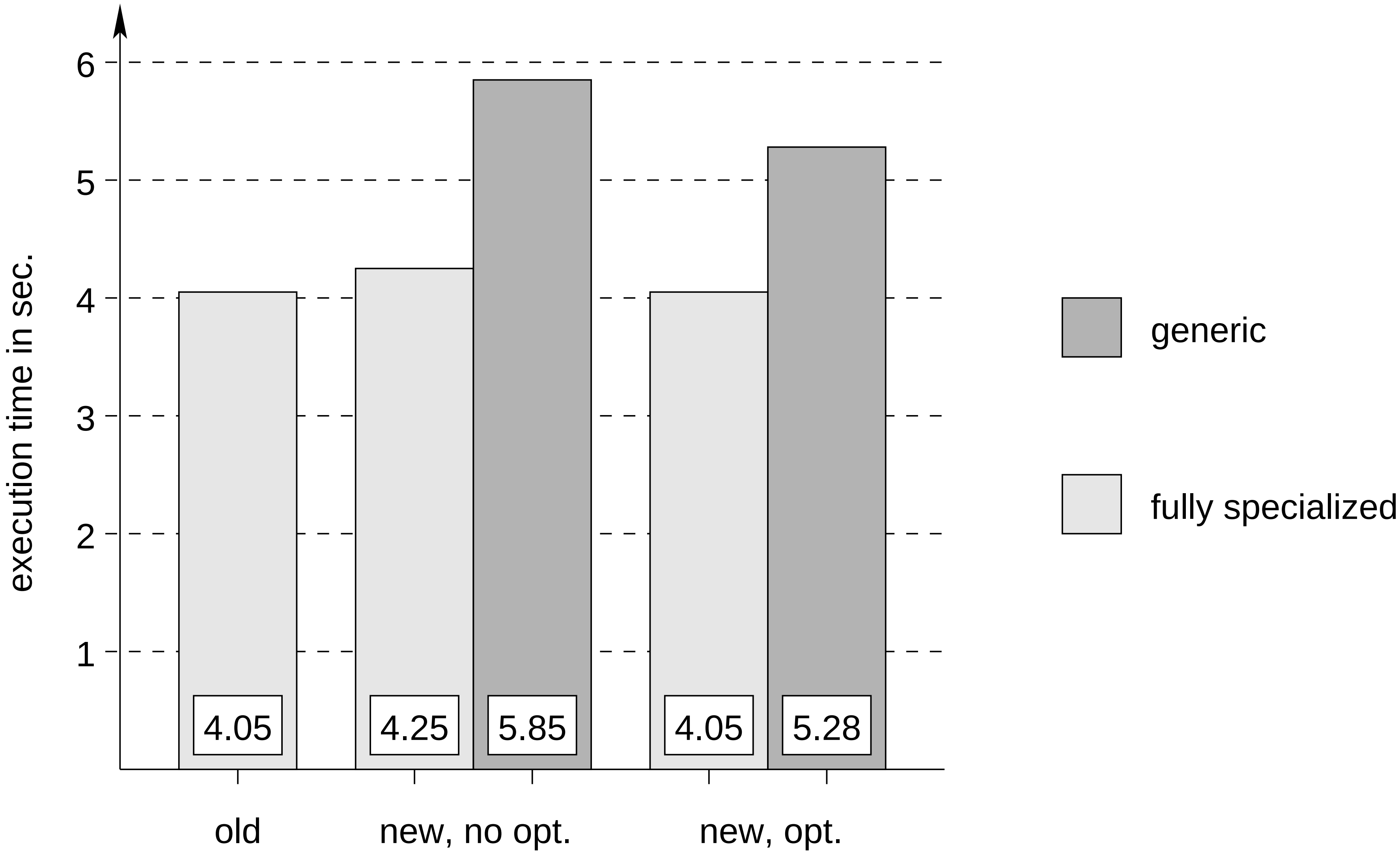
Code Generation: Array Representation (2)

Declaration in SAC	Declaration in C
$\mathcal{T}[\,]$ A;	$\mathcal{T}$ A;

$\mathcal{T}[4,3]$ A;	$\mathcal{T}$ *A;
	int *A_desc; /* rc, ~~dim~~, ~~shp0~~, ~~shp1~~ */
	const int A_dim = 2;
	const int A_shp0 = 4;
	const int A_shp1 = 3;

$\mathcal{T}[\bullet,\bullet]$ A;	$\mathcal{T}$ *A; int *A_desc; /* rc, ~~dim~~, shp0, shp1 */ const int A_dim = 2; int A_shp0; int A_shp1;
$\mathcal{T}[+]$ A; and $\mathcal{T}[*]$ A;	$\mathcal{T}$ *A; int *A_desc; /* rc, dim, shp0, ... */ int A_dim;

Performance Evaluation: Determinant of a 10×10 Array



Conclusions and Future Work

Conclusions:

- Resolution of Function Overloading:
  - high-level code transformation,
  - done statically wherever possible.
- C Representation of Arrays:
  - suitable for all array types,
  - hybrid representation used,
  - no runtime penalties.

Future Work:

Some high-level code optimizations (e.g. `with`-loop folding) are not applicable to arrays with unknown shape yet.