With-Loops, Overloading, and Parallelism

Peter Achten, Sven-Bodo Scholz

Software Science
Radboud University Nijmegen

Radboud University

Single Assignment C --- Recap of the Basics

- ➢ Array Programming
- ➢ Shape-Polymorphism
- ➢ Purely Functional (no side-effects)

- ➢ C-like look and feel
- ➢ C-like semantics

Single Assignment C --- A Language for HP^3 : High-Productivity, High-Performance, High-Portability

- Array Programming
- Shape-Polymorphism
- Purely Functional (no side-effects)

- C-like look and feel
- C-like semantics

- Powerful type system
- Powerful tensor comprehensions
- Can generate HPC code for parallel machines:
  - clusters,
  - multi-cores
  - GPUs
  - …
- No changes to the source code required!

Recap Nesting vs Shape

“outer shape”: [3]

```
x = [1,2,3];
```

“inner shape”: []

```
shape(x) == [3]
```

“outer shape”: [2]

```
y = [x,x];
```

“inner shape”: [3]

```
shape(y) == [2,3]
```

“outer shape”: [4]

```
z = [y,y,y,y];
```

“inner shape”: [2,3]

```
shape(z) == [4,2,3]
```

Recap Tensor Comprehensions

“outer shape”: [2,3]

```
{iv -> 42           | iv < [2,3]}
```

“inner shape”: []

“outer shape”: [2,3]

```
{iv -> [1,2,3,4]    | iv < [2,3]}
```

“inner shape”: [4]

```
shape ({ iv -> [iv,iv]     | iv < [2,3]})  ==   ???
```

Recap Function Signatures & Type Pattern

```
float[n:shp,m:ishp] MyTake (int[n] shp, float[n:oshp,m:ishp] a)


x = MyTake ( [2,3], reshape ([4,5,2,3,7], tof (iota (840))));

=> n == 2,
   shp == [2,3],
   oshp == [4,5],
   m == 3,
   ishp == [2,3,7],

   shape (x) == [2,3,2,3,7]
```

Recap Function Signatures & Type Pattern

```
float[n:shp,m:ishp] MyTake (int[n] shp, float[n:oshp,m:ishp] a)

x = MyTake ( [2,3], reshape ([4,5], tof (iota (20))));

=> shape (x) == [2,3]
```

Recap Function Signatures & Type Pattern

```
float[n:shp,m:ishp] MyTake (int[n] shp, float[n:oshp,m:ishp] a)
                    | all ((shp >= 0) && (shp <= oshp))
```

Dual purpose:
- match shape and dim components
- declare expected domain constraints
 (note here: fully dependent constraints!)

If checks are *wanted:* **sac2c –check c**
=> as many checks resolved *statically* as possible,
  the remaining ones are checked *dynamically!*

Putting it all Together

```
float[n:shp,m:ishp] MyTake (int[n] shp, float[n:oshp,m:ishp] a)
                    | all ((shp >= 0) && (shp <= oshp))
{
    return {iv -> a[iv] | iv < shp};
}
```

Where do the Basic Operations Come from?

```
double l2norm( double[*] A)
{
  return( sqrt( sum( square( A)));
}

double square( double A)
{
  return( A*A);
}

double *( double A, double B)
{
  return _mul_SxS_ (A, B);
}
```

Where do these Operations Come from?

```
double square( double A)
{
  return( A*A);
}

double[+] square( double[+] A)
{
  return {iv -> square (A[iv]) } ;
}
```

Desugared Tensor-Comprehensions: With-Loops

```
with {
   ([0,0] <= iv < [3,4]) : square( iv[0]);
} : genarray( [3,4], 42);
```

[0,0]	[0,1]	[0,2]	[0,3]
[1,0]	[1,1]	[1,2]	[1,3]
[2,0]	[2,1]	[2,2]	[2,3]

indices

0	0	0	0
1	1	1	1
4	4	4	4

values

With-Loops

```
with {
   ([0,0] <= iv <= [1,1]) : square( iv[0]);
   ([0,2] <= iv <= [1,3]) : 42;
   ([2,0] <= iv <= [2,2]) : 0;
} : genarray( [3,4], 21);
```

[0,0]	[0,1]	[0,2]	[0,3]
[1,0]	[1,1]	[1,2]	[1,3]
[2,0]	[2,1]	[2,2]	[2,3]

indices

0	0	42	42
1	1	42	42
0	0	0	21

values

With-Loops – overlapping partitions

```
with {
  ([0,0] <= iv <= [1,1]) : square( iv[0]);
  ([0,0] <= iv <= [1,3]) : 42;
  ([0,0] <= iv <= [2,2]) : 0;
} : genarray( [3,4], 21);
```

[0,0]	[0,1]	[0,2]	[0,3]
[1,0]	[1,1]	[1,2]	[1,3]
[2,0]	[2,1]	[2,2]	[2,3]

indices

$\rightarrow$

0	0	42	42
1	1	42	42
0	0	0	21

values

Fold-With-Loops

```
with {
   ([0,0] <= iv <= [1,1]) : square( iv[0]);
   ([0,2] <= iv <= [1,3]) : 42;
   ([2,0] <= iv <= [2,3]) : 0;
} : fold( +, 0);
```

[0,0]	[0,1]	[0,2]	[0,3]
[1,0]	[1,1]	[1,2]	[1,3]
[2,0]	[2,1]	[2,2]	[2,3]

indices

map →

0	0	42	42
1	1	42	42
0	0	0	0

values

reduce →

170

Multi-Operator-With-Loops

```
a, b = with {
          ([0,0] <= iv <= [1,1]) {
              v = square( iv[0]);
            }: (v,v);
          ([0,2] <= iv <= [1,3]) : (42, 21);
          ([2,0] <= iv <= [2,3]) : (0, 1);
        } : (genarray ([3,4], zero(a)), fold( +, 0));
```

a

0	0	42	42
1	1	42	42
0	0	0	0

b

fold (

0	0	21	21
1	1	21	21
1	1	1	1

) = 90

Tensor-Comprehensions and With-Loops

```
{ iv -> a[iv] + 1}
```

```
with {
   ( 0*shape(a) <= iv < shape(a)) : a[iv] + 1;
} : genarray( shape( a), zero(a))
```

Tensor-Comprehensions and With-Loops

```
{ iv -> a[iv] + 1 | 0*shape(a) <= iv < shape(a)}
```

```
with {
   ( 0*shape(a) <= iv < shape(a)) : a[iv] + 1;
} : genarray( shape( a), zero(a))
```

Tensor-Comprehensions and With-Loops

```
{ iv -> a[iv] + 1 | 0*shape(a) <= iv < shape(a);
  iv -> zero(a)     | iv < shape(a)}
```

```
with {
  ( 0*shape(a) <= iv < shape(a)) : a[iv] + 1;
} : genarray( shape( a), zero(a))
```

Tensor-Comprehensions and With-Loops: Overlapping Partitions

```
with {
  ([0,0] <= iv <= [1,1]) : square( iv[0]);
  ([0,0] <= iv <= [1,3]) : 42;
  ([2,0] <= iv <= [2,2]) : 0;
} : genarray( [3,4], 21);
```

0	0	42	42
1	1	42	42
0	0	0	21

```
{ iv -> square( iv[0]) | iv <= [1,1];
  iv -> 42             | iv <= [1,3];
  iv -> 0              | [2,0] <= iv <= [2,2];
  iv -> 21             | iv < [3,4] }
```

Observation

- most operations boil down to With-loops
- With-Loops are **the** source of concurrency

With-Loops and Concurrency

```
res = with {
          ([0,0] <= iv < [3,4]) : expr (iv);
      } : genarray( [3,4], 42);
```

lexical scoping => ***expr*** can only refer to variables defined before this definition!
side-effect-free => ***expr*** neither relies on some shared state nor does it change some shared state!
for each value of `iv` we compute ***expr*** `(iv)` exactly once!

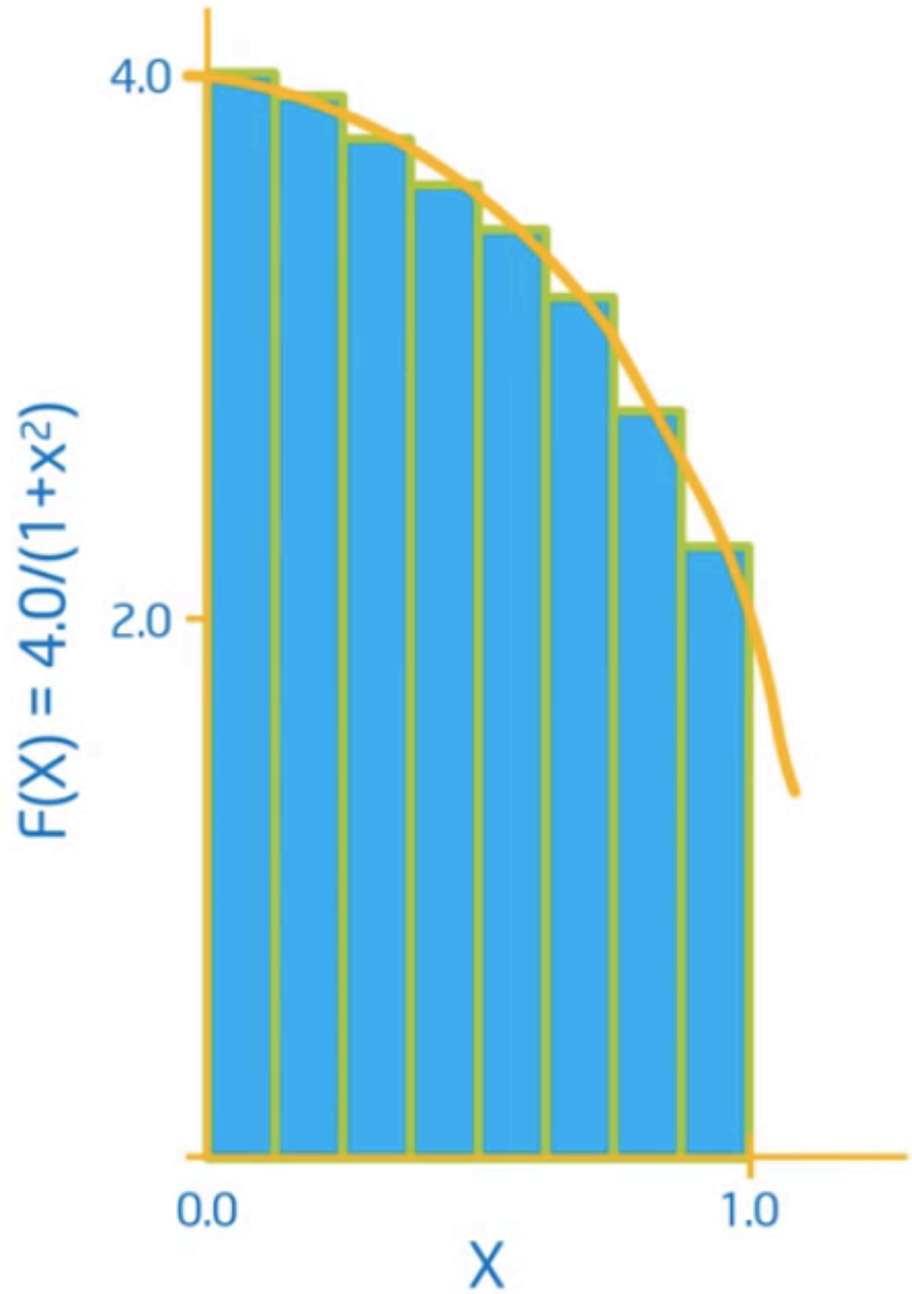
⇒ Semantics guarantees that the order of evaluation does not affect the result
⇒ Parallelism is possible!

Approximation of Pi

```
double f( double x)
{
  return 4.0 / (1.0+x*x);
}

int main()
{
  num_steps = 10000;
  step_size = 1.0 / tod (num_steps);
  x = (0.5 + tod (iota (num_steps)))
      * step_size;
  y = { iv -> f( x[iv])};
  pi = sum (step_size * y);

  printf( " ...and pi is: %f\n", pi);
  return(0);
}
```



Approximation of Pi --- Concu

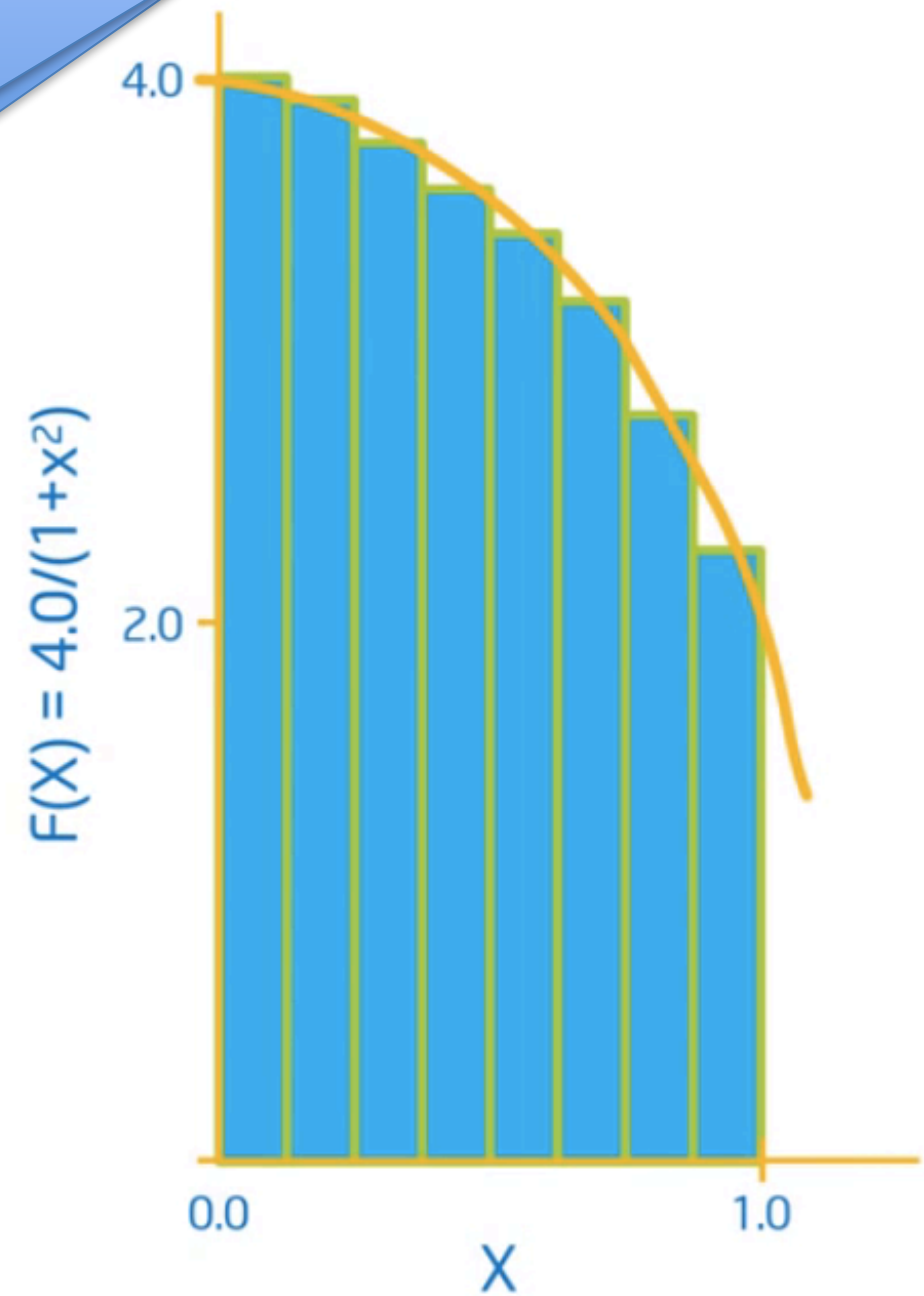
flops: 10.000 * 9 + 2

flops assuming 10k cores: 8 + 13 + 2

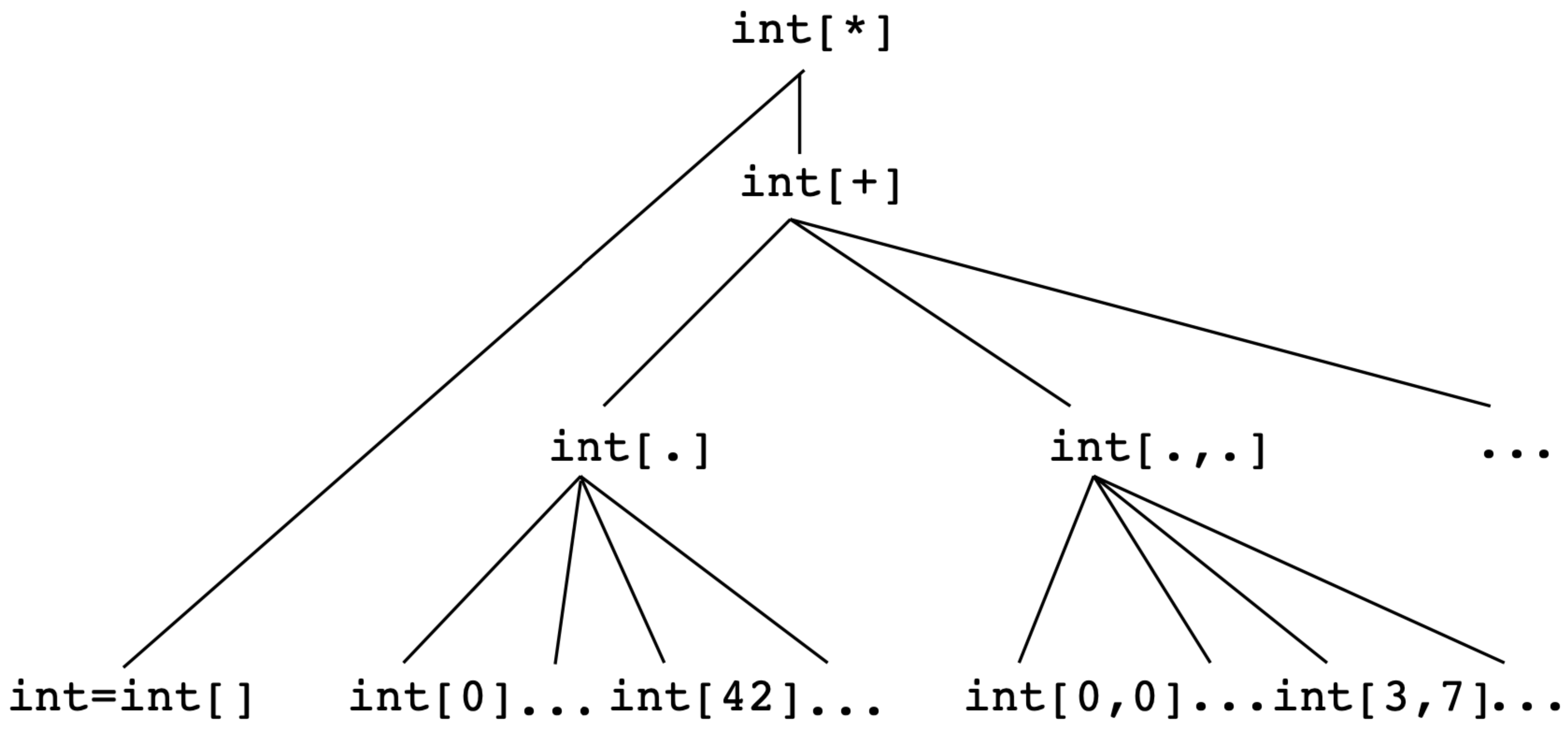
```
double f( double x)
{
  return 4.0 / (1.0+x*x);
}

int main()
{
  num_steps = 10000;
  step_size = 1.0 / tod (num_steps);
  x = (0.5 + tod (iota (num_steps)))
        * step_size;
  y = { iv -> f( x[iv])};
  pi = sum (step_size * y);

  printf( " ...and pi is: %f\n", pi);
  return(0);
}
```

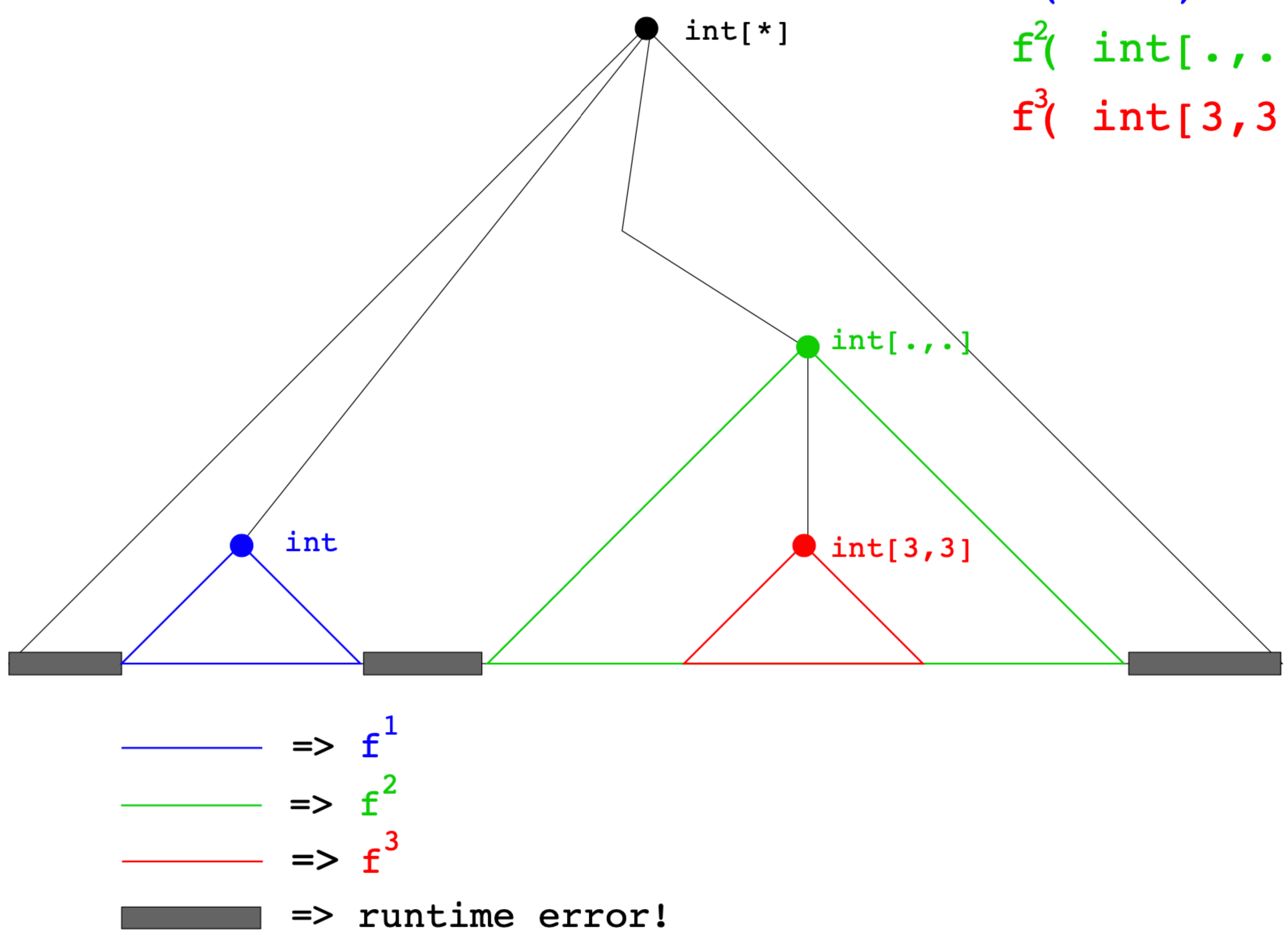


The Hierarchy of Types in SaC



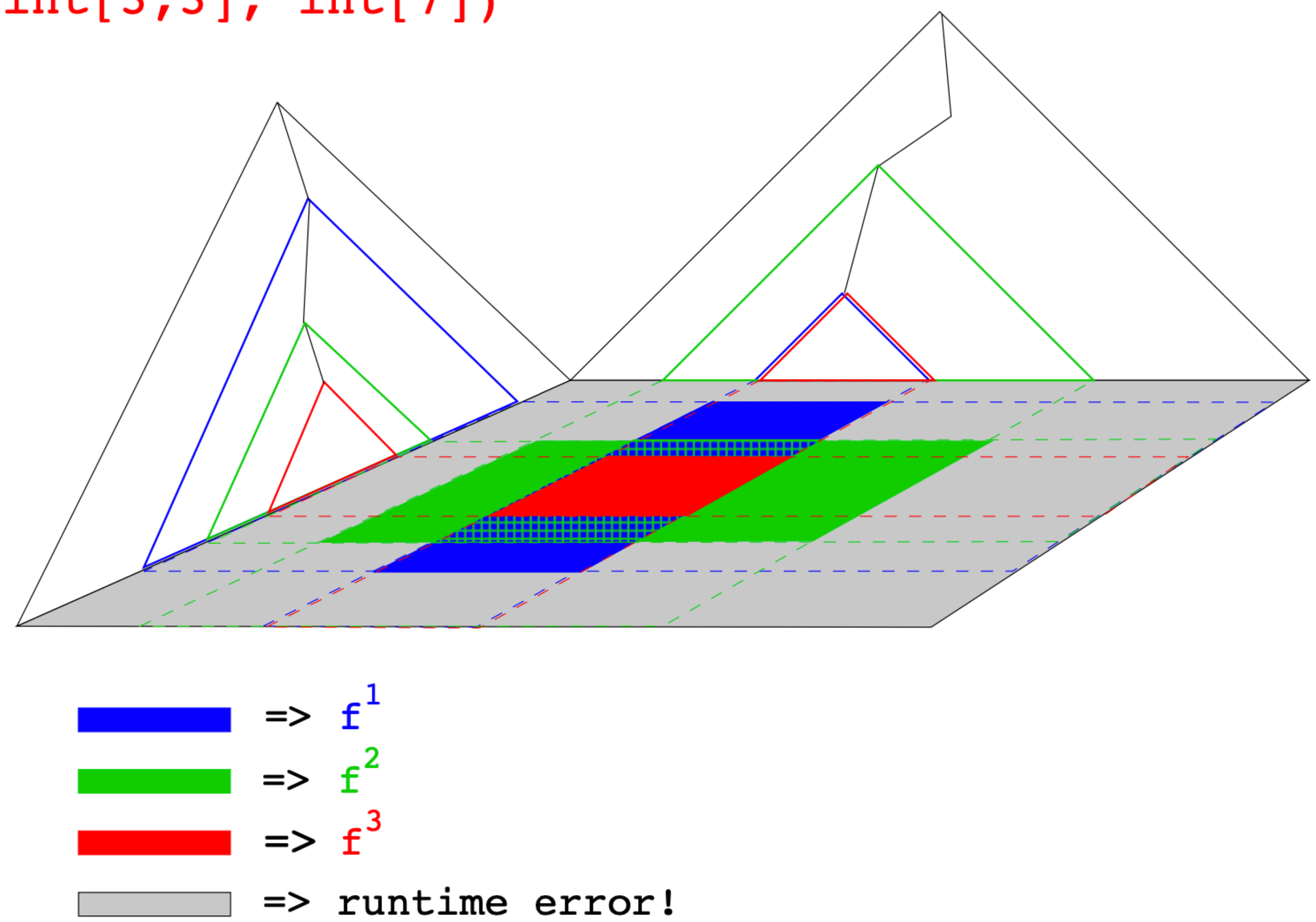
Overloading in SaC

```
f¹( int)
f²( int[.,.])
f³( int[3,3])
```



Overloading Multi-Parameter Functions

```
f1( int[+], int[7])
f2( int[.,.], int[.])
f3( int[3,3], int[7])
```



SaC overloading vs templates vs type classes vs traits

SaC:
```
int foo( int n) …
double foo( double n) …
```

C++:
```
template <typename T>
T foo( T n) …
```

Haskell:
```
class Foo T where
foo:: T -> T

instance Foo Int where
foo n = …
instance Foo Double where
foo n = …
```

Rust:
```
pub trait Foo {
   fn foo( &self) -> &self
}

impl Foo for int {
   fn  foo (&self) = …
}
impl Foo for double {
   fn  foo (&self) = …
}
```

Revisiting Matrix Multiply

```
float[m,p] matmul (float[m,n] a, float[n,p] b)
{
   return {[i,j] -> vsum( {[k] -> (a[i,k] * b[k,j]) }) };
}
```

assume n = m = p

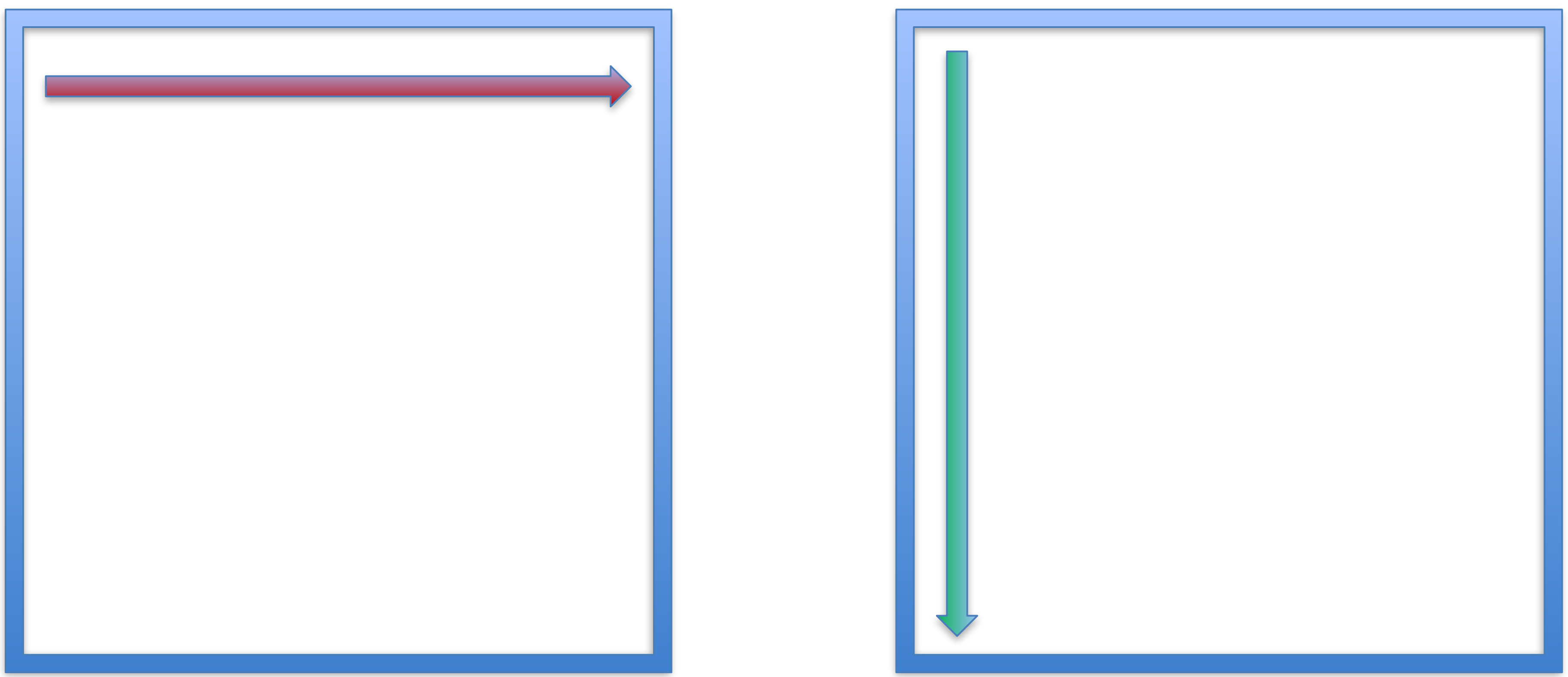
Flops: $2*n^3$

Data: $2*n^2$

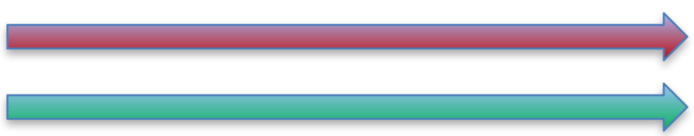
should be compute bound, no?!

Revisiting Matrix Multiply

```
float[m,p] matmul (float[m,n] a, float[n,p] b)
{
   return {[i,j] -> vsum( {[k] -> (a[i,k] * b[k,j]) }) };
}
```



if n is large,



are gone from the cache, when we do the next element!

=> n^3 data reads
=> memory bound!

The Idea of Tiling

shape [4,4]

A

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

shape [2,2,2,2]

$A_{0,0}$ $A_{0,1}$

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

$A_{1,0}$ $A_{1,1}$

A x B =

$A_{0,0} \times B_{0,0} + A_{0,1} \times B_{1,0}$	$A_{0,0} \times B_{0,1} + A_{0,1} \times B_{1,1}$
$A_{1,0} \times B_{0,0} + A_{1,1} \times B_{1,0}$	$A_{1,0} \times B_{0,1} + A_{1,1} \times B_{1,1}$

much better cache reuse!
=> no longer memory bound!

Tiling and Reshaping

```
float[m,p] matmul (float[m,n] a, float[n,p] b)
{
   return {[i,j] -> vsum( {[k] -> (a[i,k] * b[k,j]) }) };
}
```

$A \times B =$

$A_{0,0} \times B_{0,0} + A_{0,1} \times B_{1,0}$	$A_{0,0} \times B_{0,1} + A_{0,1} \times B_{1,1}$
$A_{1,0} \times B_{0,0} + A_{1,1} \times B_{1,0}$	$A_{1,0} \times B_{0,1} + A_{1,1} \times B_{1,1}$

$A_{0,0}$ $A_{0,1}$

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

$A_{1,0}$ $A_{1,1}$

```
float[m,p,+] matmul (float[m,n,+] a, float[n,p,+] b)
{
   return {[i,j] -> vsum( {[k] -> matmul (a[i,k], b[k,j]) }) };
}
```

Rank-Polymorphism at Work

shape
[4,4]

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

shape
[2,2,2,2]

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

```
float[m,p,+] matmul (float[m,n,+] a, float[n,p,+] b)
{
    return {[i,j] -> vsum( {[k] -> matmul (a[i,k], b[k,j]) }) };
}

matmul (a::[2,2,2,2], b::[2,2,2,2])
=> res = {[i,j] -> vsum( {[k] -> matmul (a[i,k], b[k,j]) }) }
                                 matmul (a::[2,2], b::[2,2])
                                 => res::[2,2]
=> res::[2,2,2,2]
```

Rank-Polymorphism for Tiling

```
float[m,p,+] matmul (float[m,n,+] a, float[n,p,+] b)
{
   return {[i,j] -> vsum( {[k] -> matmul (a[i,k], b[k,j]) }) };
}

float[m,p] matmul (float[m,n] a, float[n,p] b)
{
   return {[i,j] -> vsum( {[k] -> (a[i,k] * b[k,j]) }) };
}
```

tiling is encoded in the shape, rather than in the code!
yields BLAS performance! (see FHPNC'23 paper)