Flattening Combinations of Arrays and Records

TFP'24 – 11 January 2024

Reg Huijben
Jordy Aaldering
Peter Achten
Sven-Bodo Scholz

INTRODUCTION

Functional programming

- Programmer productivity
- What not how

Array languages:

- Single assignment C – SaC
- Excellent performance
- Flattening

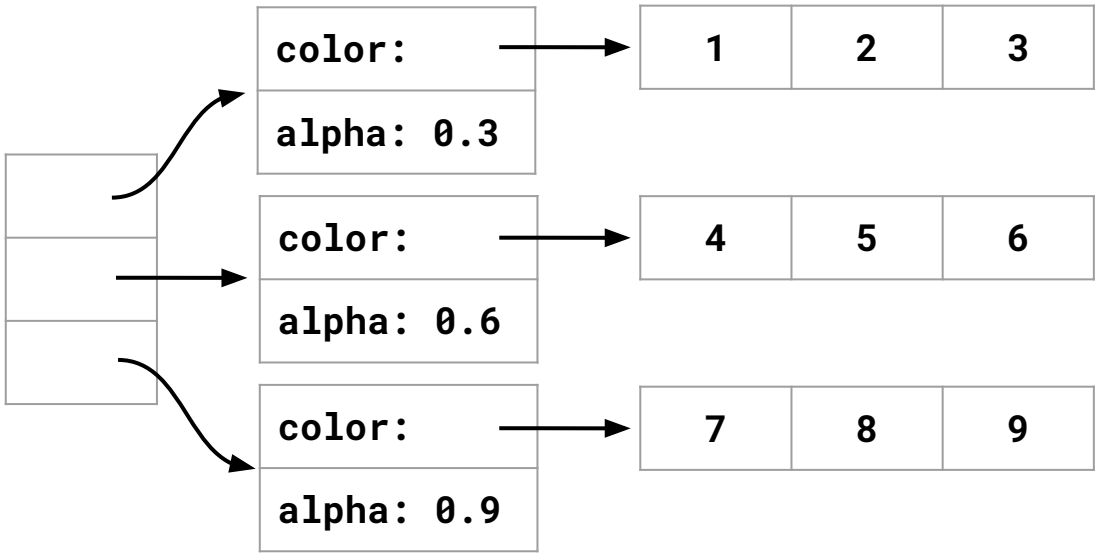
INTRODUCTION

Records:

- Expressiveness
- Group related data
- Memory efficiency?
- Performance?

```
[ Pixel { color=[1,2,3], alpha=0.3 },
  Pixel { color=[4,5,6], alpha=0.6 },
  Pixel { color=[7,8,9], alpha=0.9 } ]
```

Nested

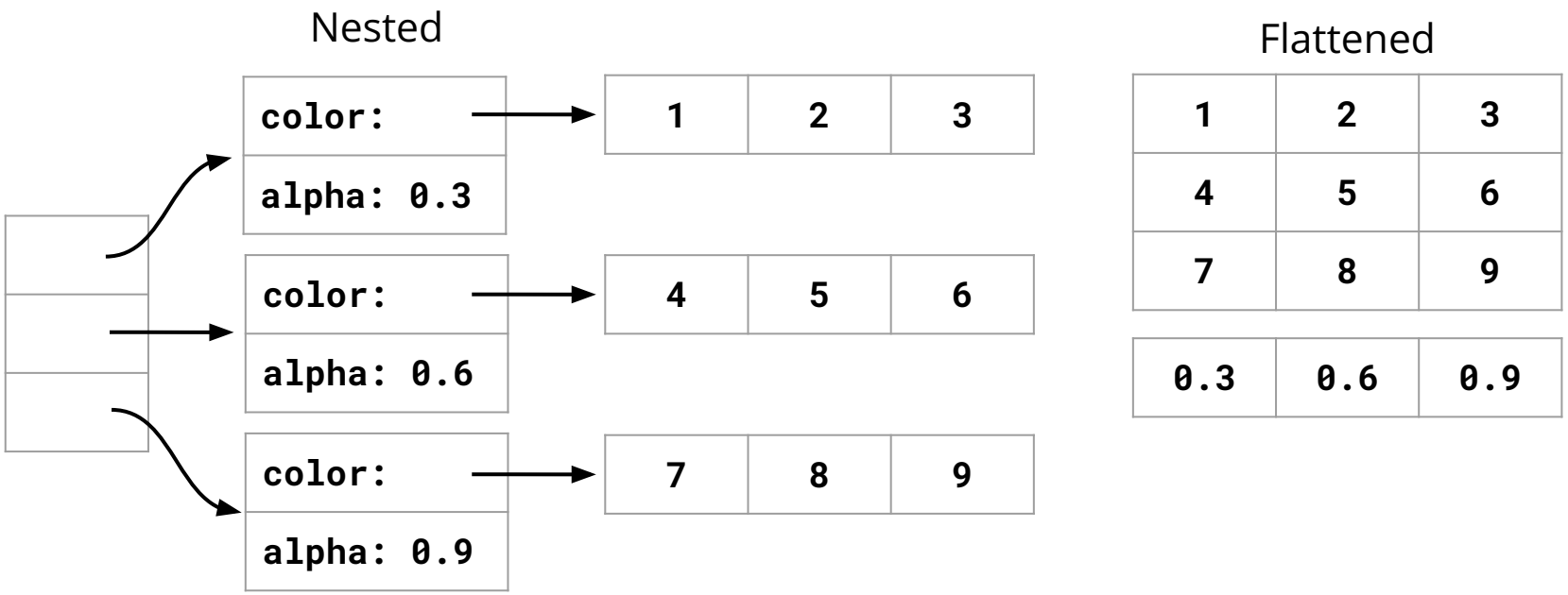


FLATTENING

Flattened:

- Less memory
- No pointers
- Performance benefits
- Homogeneous

```
[ Pixel { color=[1,2,3], alpha=0.3 },
  Pixel { color=[4,5,6], alpha=0.6 },
  Pixel { color=[7,8,9], alpha=0.9 } ]
```

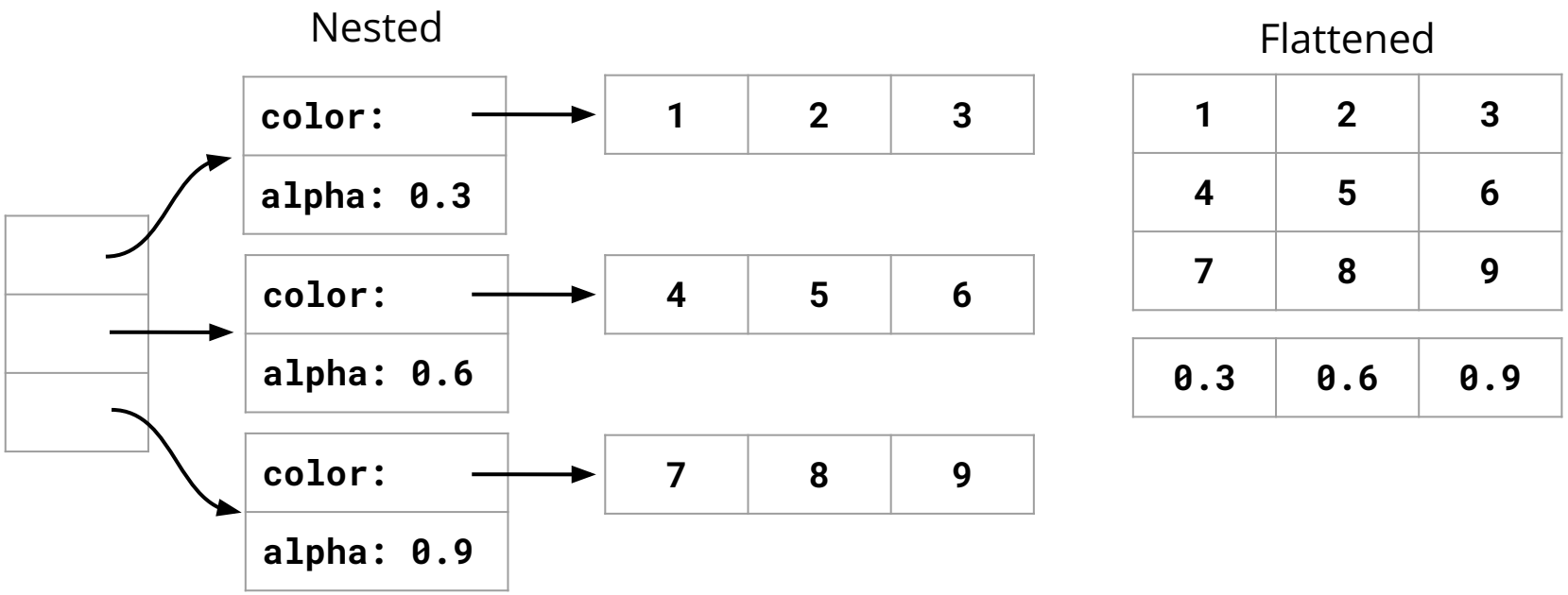


FLATTENING

Flattening is non-trivial:

- Mixing data-types
- Nested records
- Nested lists
- Memory size

```
[ Pixel { color=[1,2,3], alpha=0.3 },
  Pixel { color=[4,5,6], alpha=0.6 },
  Pixel { color=[7,8,9], alpha=0.9 } ]
```



RESTRICTIONS

No recursion

Constant size

- Ensures homogeneous
- How much memory to allocate

```
struct Pixel {
    int[3] color;
    double alpha;
}
```

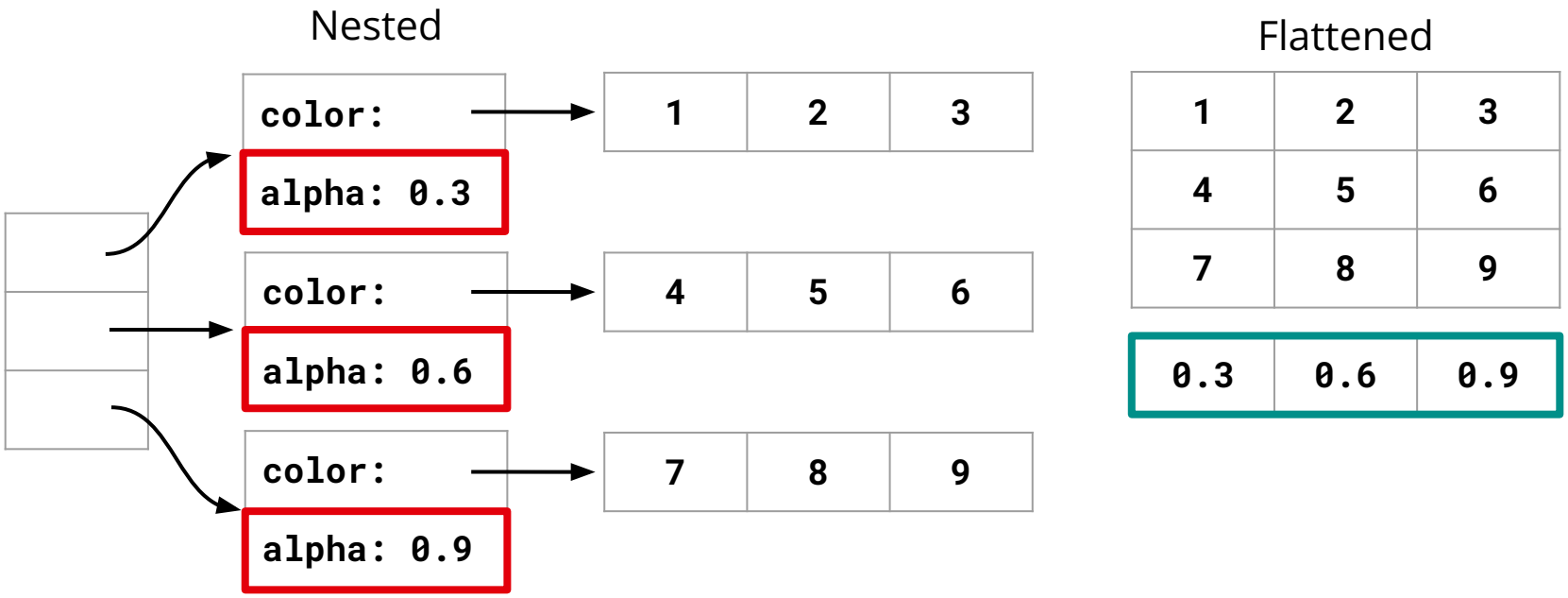
FLATTENING

Main benefits of flattened notation:

- Memory locality
- Fewer allocations

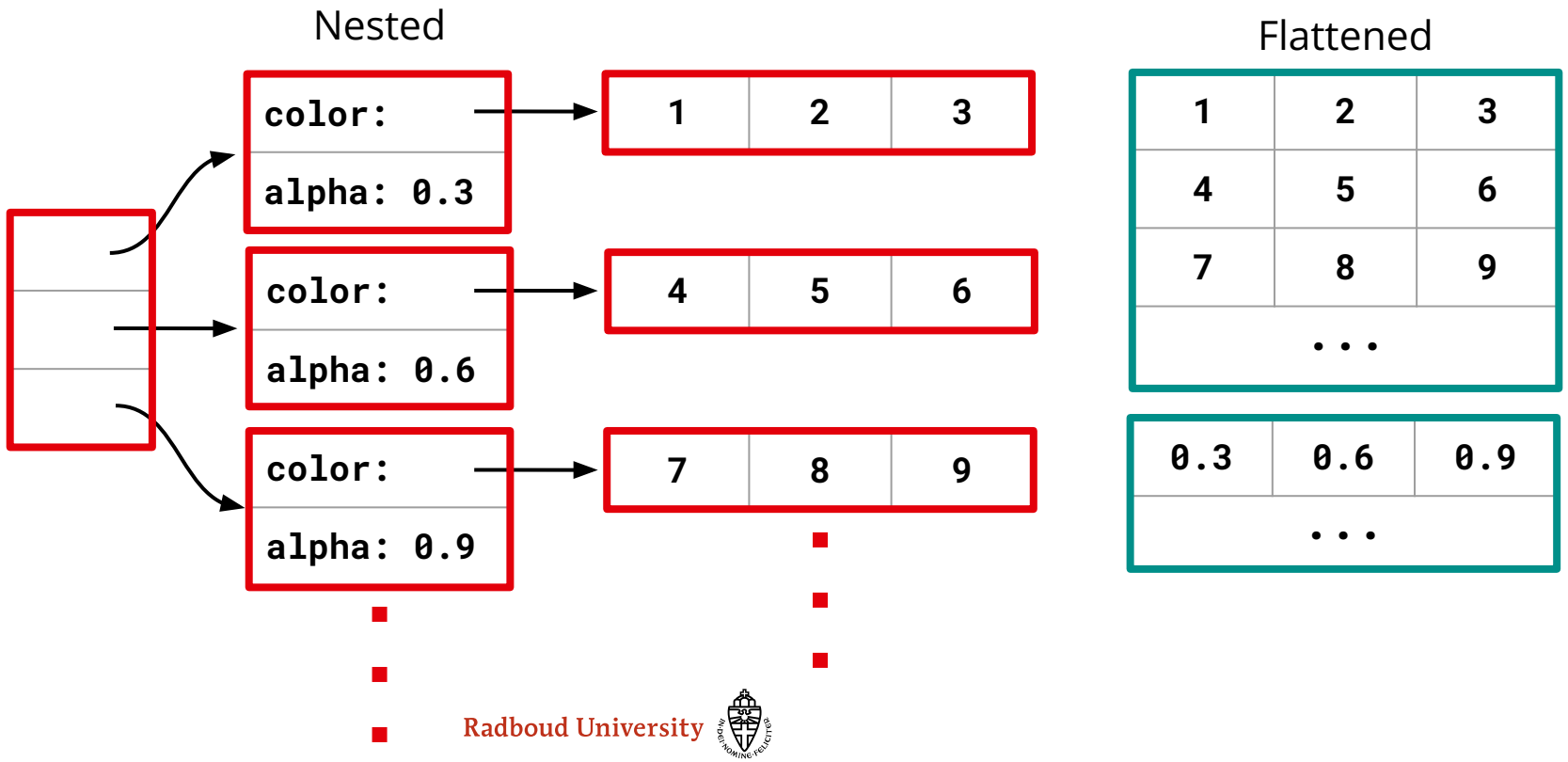
LOCALITY

```
image = [ Pixel { color=[1,2,3], alpha=0.3 },
          Pixel { color=[4,5,6], alpha=0.6 },
          Pixel { color=[7,8,9], alpha=0.9 } ];
a = avg(image.alpha);
```



ALLOCATIONS

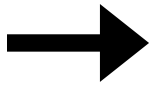
```
image = [ Pixel { color=[1,2,3], alpha=0.3 },
          Pixel { color=[4,5,6], alpha=0.6 },
          Pixel { color=[7,8,9], alpha=0.9 }
          ... ];
```



TRANSFORMATION

- Remove records from program
- Record fields become arguments
- So no records at run-time

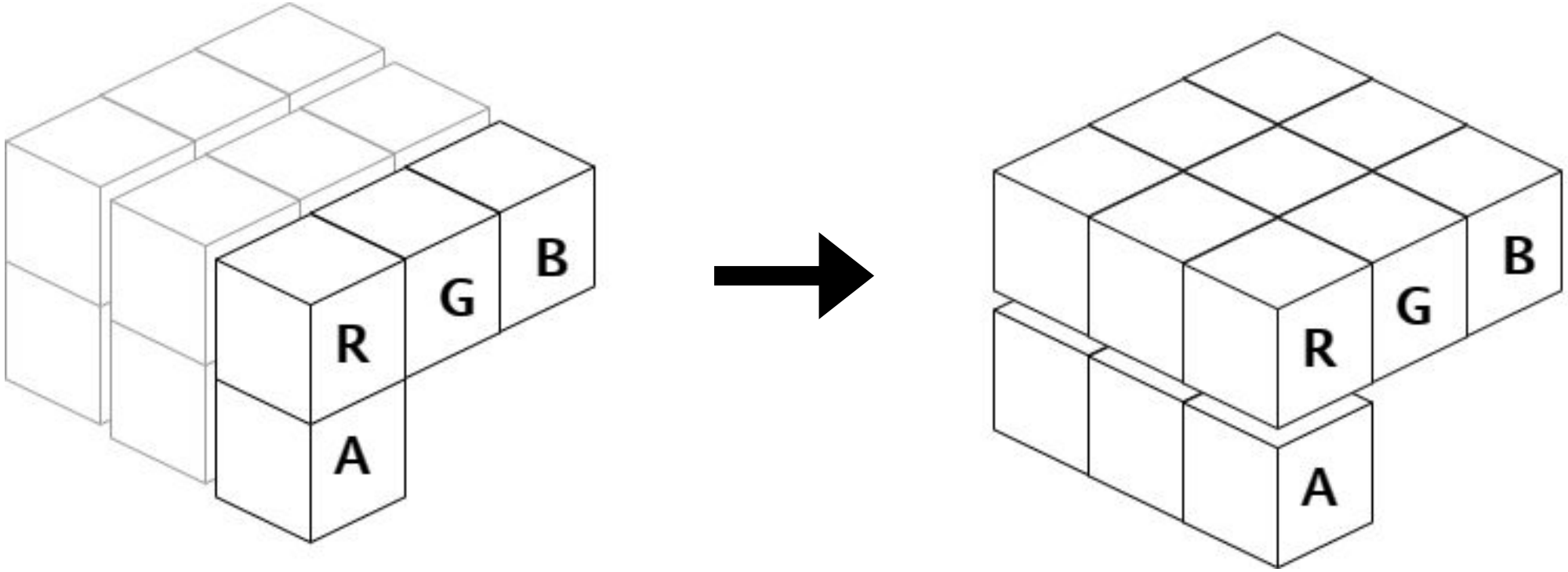
```
struct Pixel[x,y]
fade (struct Pixel[x,y] image)
{
    image.alpha /= 2.0;
    return image;
}
```



```
int[x,y,3], double[x,y]
fade (int[x,y,3] colors, double[x,y] alphas)
{
    alphas /= 2.0;
    return (colors, alphas);
}
```

TRANSFORMATION

- List of records, to records of lists

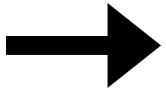


TRANSFORMATION

- List of records, to records of lists

```
struct Pixel {
    int[3] color;
    double alpha;
}

Pixel[x,y] image;
```



```
int[x,y,3] imageColors;
double[x,y] imageAlphas;
```

What to do for:

- Nested records
- Lists of records
- Records of lists

PRIMITIVE FUNCTIONS

Expanding arguments changes function application

```
[2]
imageSize (struct Pixel[x,y] image)
{
    return shape(image);
}

[2]
imageSize (int[x,y,3] colors, double[x,y] alphas)
{
    return shape(colors, alphas);
}
```

?

PRIMITIVE FUNCTIONS

Arbitrarily take the first record field

```
[2]                    ?
imageSize (struct Pixel[x,y] image)
{
    return shape(image);
}

[2]                ?
imageSize (int[x,y,3] colors, double[x,y] alphas)
{
    return shape(colors);
}
```

PRIMITIVE FUNCTIONS

We require the shape is constant

- Dimensionality is known
- Remove that many elements

```
[2]
imageSize (struct Pixel[x,y] image)
{
    return shape(image);
}

[2]
imageSize (int[x,y,3] colors, double[x,y] alphas)
{
    return drop(-1, shape(colors));
}
```

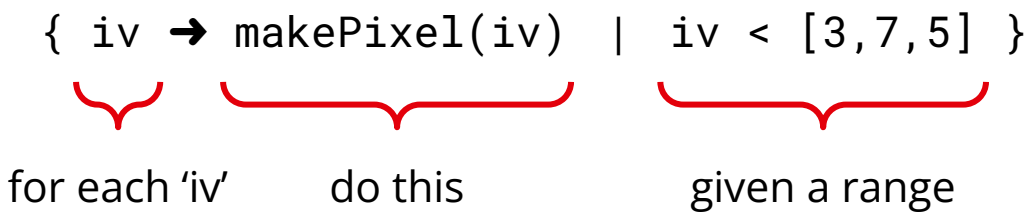
PARALLEL OPERATIONS

Parallel operations on the expanded arguments

- Key to achieving good runtime performance
- Sharing of computations between generated code

SaC's vehicle for parallelism:

- Tensor comprehension



PARALLEL OPERATIONS

Allow for multiple return values

```
struct Pixel[x,y]
makeImage (int[2] size)
{
    image = { iv ➜ makePixel(iv)  |  iv < size };
    return image;
}
```

```
int[x,y,3], int[x,y]
makeImage (int[2] size)
{
    colors, alphas = { iv ➜ makePixel(iv)  |  iv < size };
    return (colors, alphas);
}
```

MULTI-OPERATOR FOLD

Fold should work on records

```
struct Pixel
maxPixel (struct Pixel a, struct Pixel b)
{
    ...
}


pixel = with {
            ([0,0] <= iv < size): pixels[iv];
        } fold(maxPixel, Pixel{});
```

MULTI-OPERATOR FOLD

Fold should work on multiple arguments

```
int[3], double
maxPixel (int[3] colorA, double alphaA, int[3] colorB, double alphaB)
{
    ...
}
```

```
color, alpha = with {
            ([0,0] <= iv < size): pixels[iv];
        } fold(maxPixel, [0,0,0], 0.0);
```

UNUSED ARGUMENTS

Number of arguments will explode

- Especially for nested records
- Some might not be needed

```
double
maxAlpha (struct Pixel[x,y] image)
{
    return max(image.alpha);
}

double
maxAlpha (int[x,y,3] colors, double[x,y] alphas)
{
    return max(alphas);
}
```

UNUSED ARGUMENTS

Get rid of any unused arguments

- Iterative optimisation
- Is argument used in function body?

Challenges:

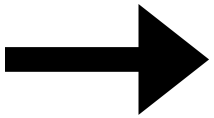
- Overloading
- Exporting

OVERLOADING

Overloaded function might already exist

```
double
maxAlpha (struct Pixel[x,y] image)
{
    return max(image.alpha);
}

double
maxAlpha (double[x,y] array)
{
    return array[0,0];
}
```



```
double
maxAlpha (double[x,y] alphas)
{
    return max(alphas);
}

double
maxAlpha (double[x,y] array)
{
    return array[0,0];
}
```

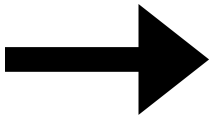
OVERLOADING

In the case of SaC we generate unique function names anyways

- Use original signature for name
- Keep original signature until code generation

```
double
maxAlpha (struct Pixel[x,y] image)
{
    return max(image.alpha);
}

double
maxAlpha (double[x,y] array)
{
    return array[0,0];
}
```



```
double
maxAlpha_i_d (double[x,y] alphas)
{
    return max(alphas);
}

double
maxAlpha_d (double[x,y] array)
{
    return array[0,0];
}
```

EXPORTING

Users expect a certain usage given a signature

- Only remove after exporting
- But optimisation happens before export

```
export {maxAlpha};

double
maxAlpha (int[x,y,3] colors, double[x,y] alphas)
{
    return max(alphas);
}


myColors = makeColors(3);
myAlphas = [0.3, 0.6, 0.9];
maxAlpha(myColors, myAlphas);
```

EXPORTING

Solution:

- Mark during optimisation
- Update applications during optimisation
- Actually remove after export

```
export {maxAlpha};

double
maxAlpha (/* unused */ int[x,y,3] colors, double[x,y] alphas)
{
    return max(alphas);
}


myColors = makeColors(3);
myAlphas = [0.3, 0.6, 0.9];
maxAlpha(/* unused */ dummyValue, myAlphas);
```

UNUSED ARGUMENTS

After optimisation and export:

- Rename
- Remove

```
export {maxAlpha_i_d};

double
maxAlpha_i_d (double[x,y] alphas)
{
    return max(alphas);
}
```

```
myAlphas = [0.3, 0.6, 0.9];
maxAlpha_i_d(myAlphas);
```

CONCLUSION

Support for records without paying a memory or performance price

- Records fully compiled away
- Seamless flattening through records
- Special treatment required
- Some restrictions required

Unused argument removal universally applicable

Final stages of development

- Available soon in SaC 2.0

```
[ Pixel { color=[1,2,3], alpha=0.3 },
  Pixel { color=[4,5,6], alpha=0.6 },
  Pixel { color=[7,8,9], alpha=0.9 } ]
```