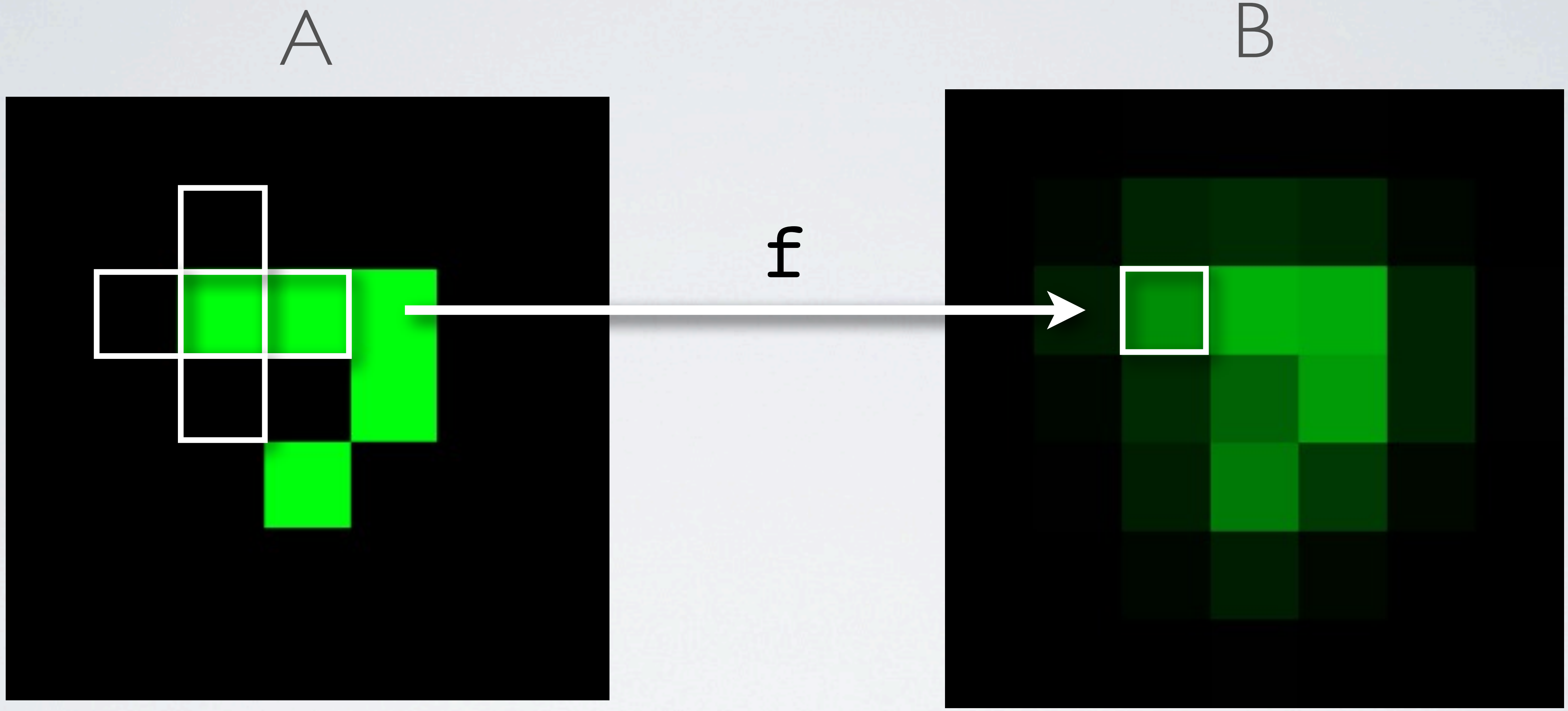
EFFICIENT AND *CORRECT* STENCIL COMPUTATIONS

Via *Pattern Matching* & *Static Typing*

Dominic Orchard, Alan Mycroft

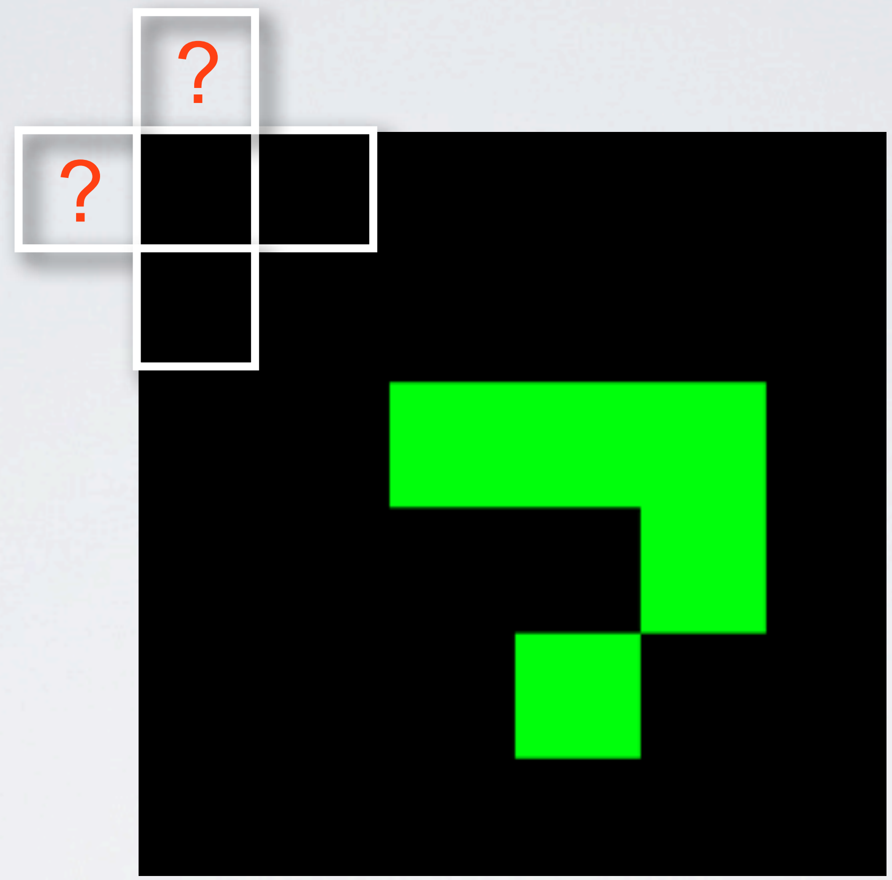
7th September, Bordeaux, IFIP DSL 2011

Stencil Computations




```
for (i=0; i<N; i++)
    for (j=0; j<M; j++)
        B[i][j] = f(A[i][j],   A[i-1][j], A[i+1][j],
                           A[i][j-1], A[i][j+1]);
```

Stencil Computations

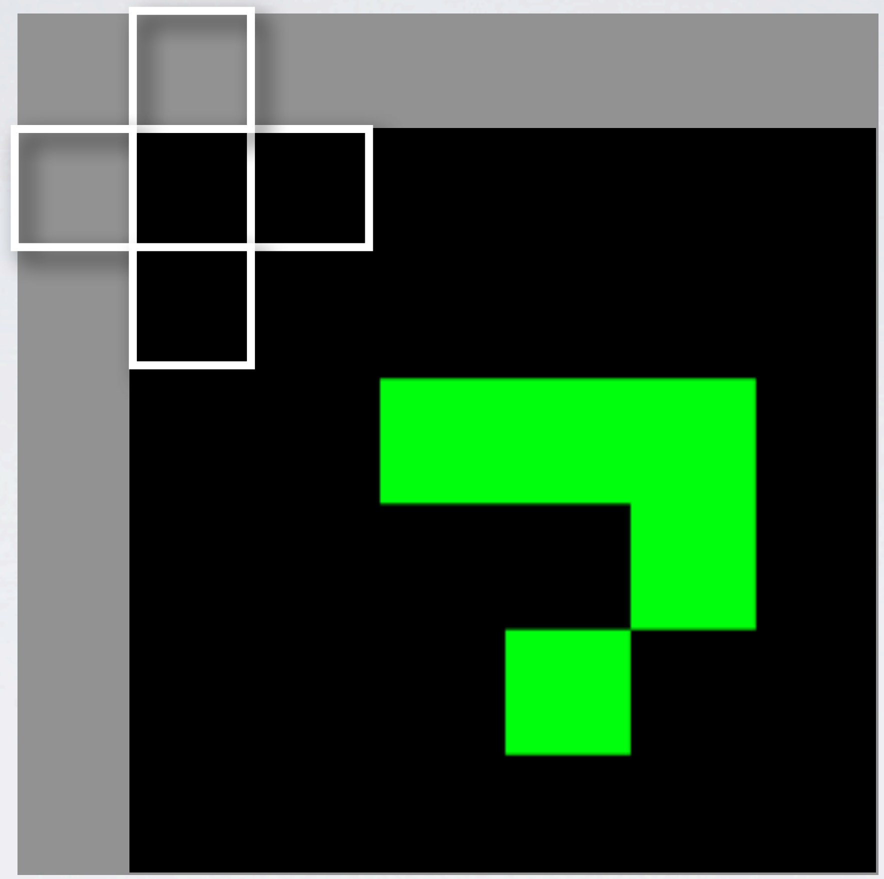


when

i=0, j=0

```
for (i=0; i<N; i++)
    for (j=0; j<M; j++)
        B[i][j] = f(A[i][j],   A[i-1][j], A[i+1][j],
                        A[i][j-1], A[i][j+1]);
```

Stencil Computations



A & B of size (N+2 x N+2)

```
for (i=0; i<(N+2); i++)
    for (j=0; j<(M+2); j++)
        if (i<1 || i>N || j<1 || j>M)
            A[i][j] = 0.3; -- boundary value

for (i=1; i<(N+1); i++)
    for (j=1; j<(M+1); j++)
        B[i][j] = f(A[i][j],   A[i-1][j], A[i+1][j],
                               A[i][j-1], A[i][j+1]);
```

Common Bugs

- Out-of-bounds errors e.g. `A[-1][0]`
  - Crash
  - Silent (even worse)
- Uninitialised boundaries e.g. `A[0][0]` = ?
- Typos e.g. `A[j-1][i]` instead of `A[i][j-1]`

Cause: **arbitrary index expressions**

Solutions

- Static analyses (in general undecidable)
- Runtime bounds checking
  - ***Pro:*** Prevents numerical correctness bugs
  - ***Cons:*** Adds overhead (~20% on a single iteration over a 512x512 image)
- Typos can still be a problem

Philosophy

- General purpose languages lose program information
- General purpose languages obscure optimisations and obfuscate (in)correctness
- Domain specific languages permit better optimisation and reasoning via restricted syntax (Not "*mere syntax*"!)
- Restricted syntax $\Longrightarrow$ more (decidable) information
- More information $\Longrightarrow$ better optimisation & reasoning

Ypnos (pronounced *ip-nos*)

- Internal DSL in Haskell for (n-dimensional) stencil computations
- Shallow embedding + specialised syntax via macros
- Slogan (of this paper/talk):

  Well-typed Ypnos programs have only safe indexing (i.e. no out-of-bounds errors)

- Correctness guarantees $\Longrightarrow$ efficiency guarantees: bounds checks can be eliminated as all array access is safe
- Guarantees parallelisation (not discussed today)

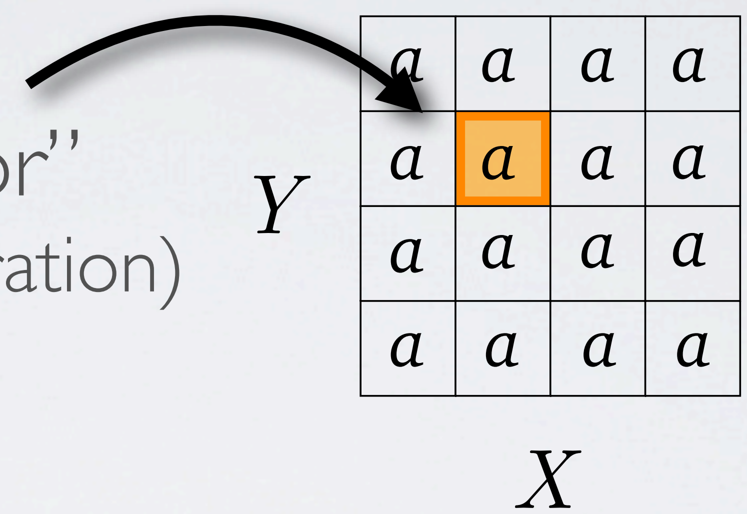
Safety in Ypnos

- No general, arbitrary indexing
- Specialised syntax for array access (grid patterns) and boundary definitions
- Restricted syntax provides static information, encoded in types
- Safety invariant enforced via type-checking

Grids

- Grid value comprise an <u>array</u> and a <u>cursor</u>

"cursor"
(current index under consideration)



- Grid type constructor encodes information

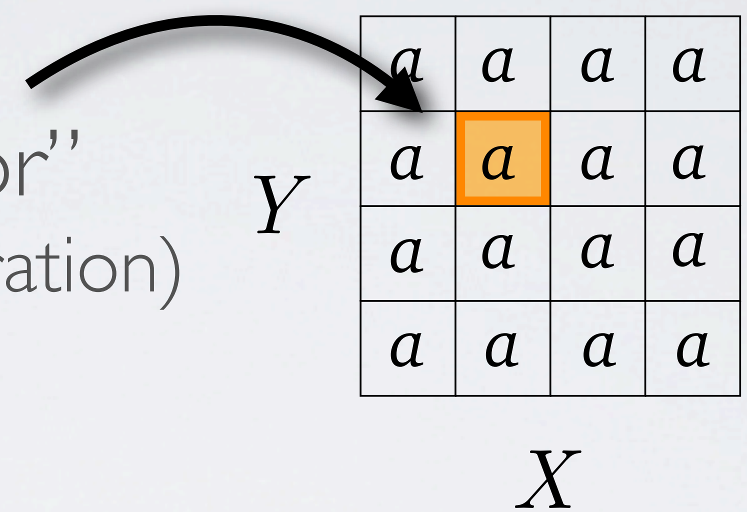
$$Grid\ \underbrace{dim}_{\text{Type-level dimensionality}}\ bounds\ a$$

e.g. $X \times Y$

Grids

- Grid value comprise an <u>array</u> and a <u>cursor</u>

"cursor"
(current index under consideration)

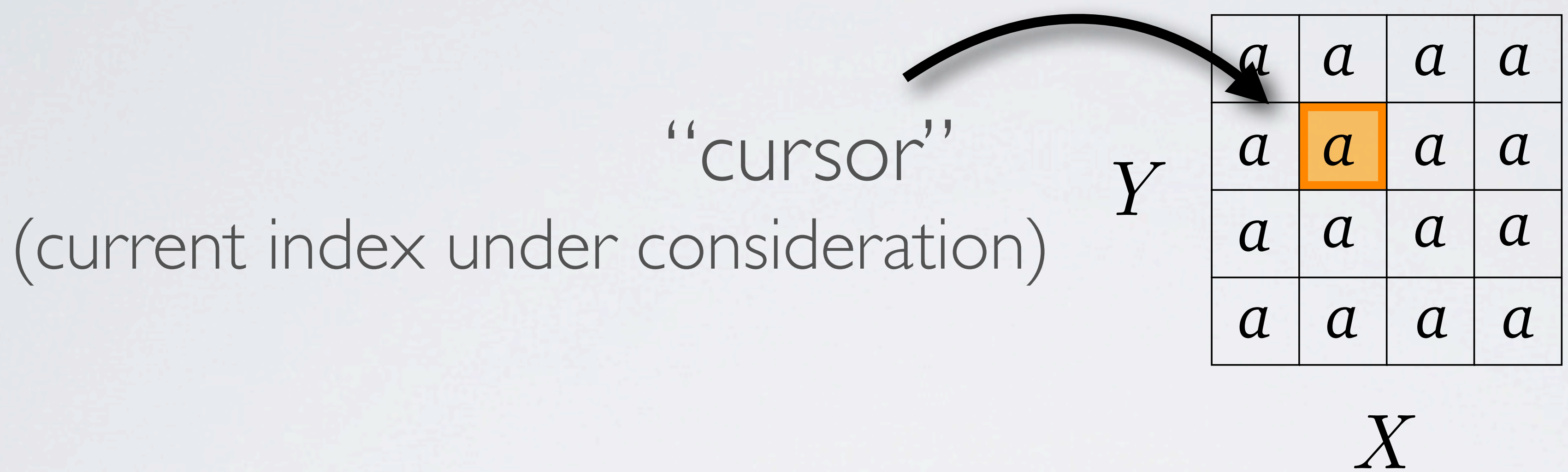


- Grid type constructor encodes information

$$Grid\ dim\ \underbrace{bounds}_{\text{Type-level boundary info}}\ a$$

Grids

- Grid value comprise an <u>array</u> and a <u>cursor</u>



- Grid type constructor encodes information

$$\mathit{Grid}\ \mathit{dim}\ \mathit{bounds}\ \underbrace{a}_{\text{Element type of the grid}}$$

Grid Patterns

- Special kind of pattern match on $Grid$ values

$$\underbrace{\texttt{f | @c |}}_{\text{Grid pattern}}\texttt{ = ...}$$

`c` bound to the "cursor" element

e.g. `c = A[i]`

$$\underbrace{\texttt{f | l @c r |}}_{\text{Grid pattern}}\texttt{ = ...}$$

`l` bound to left of "cursor"

`r` bound to right of "cursor"

e.g.
```
l = A[i-1]
c = A[i]
r = A[i+1]
```

Grid Patterns (2)

```
f :: Grid X b Double → Double
f | l @c r | = (l+c+r)/3.0
```

Computes average of neighbours

$$\texttt{run} :: (\underbrace{Grid\ d\ b\ x \rightarrow y}_{\text{Stencil function (kernel)}}) \rightarrow Grid\ d\ b\ x \rightarrow Grid\ d\ \{\}\ y \quad *$$

$$\texttt{runA} :: (Grid\ d\ b\ a \rightarrow a) \rightarrow Grid\ d\ b\ a \rightarrow Grid\ d\ b\ a$$

Applies a stencil function at every possible "cursor" position

```
g' = run f g
```

*Note: related to extension operation of a *comonad*

Grid Patterns (3)

```
f X:| l @c r | = ...
```

Dimension annotation

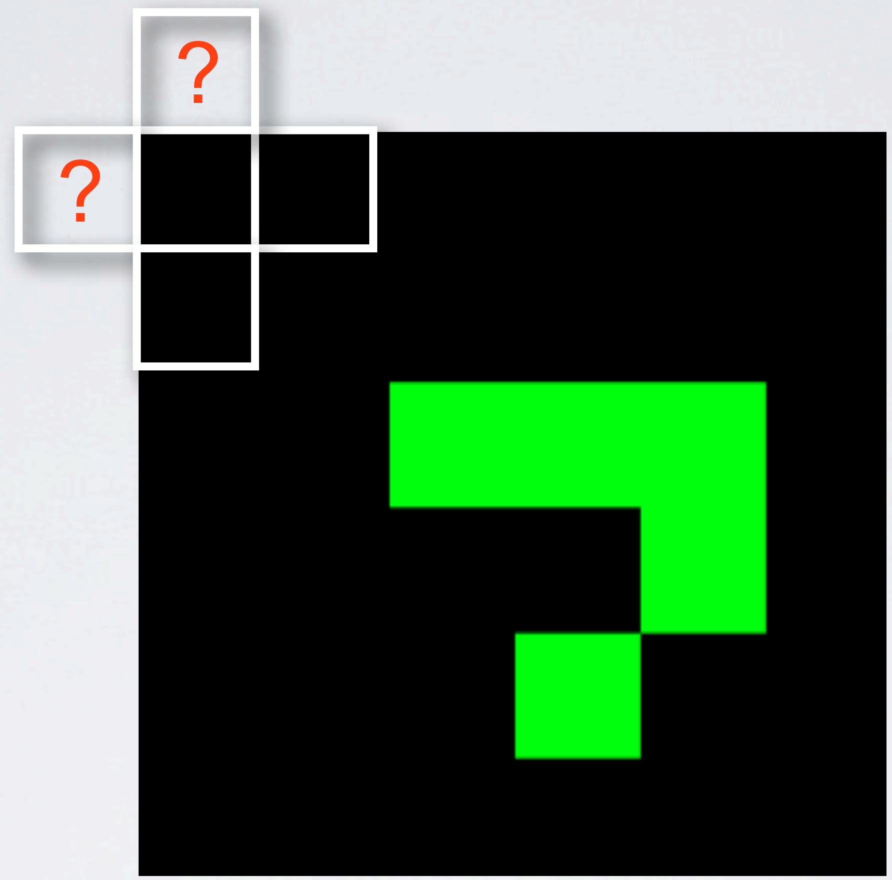
Two-dimensional syntactic sugar

```
laplace2D (X*Y):| _  t _ | = t + l + r + b - 4*c
                | l @c r |
                | _  b _ |
```

e.g.

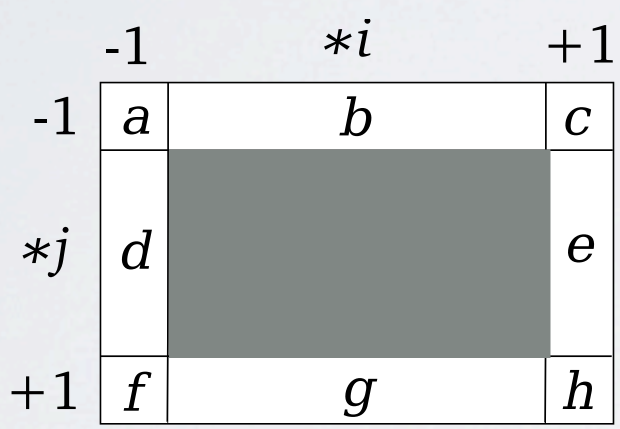
```
l = A[i-1][j]
c = A[i][j]
r = A[i+1][j]
```

```
t = A[i][j-1]
b = A[i][j+1]
```



Boundaries

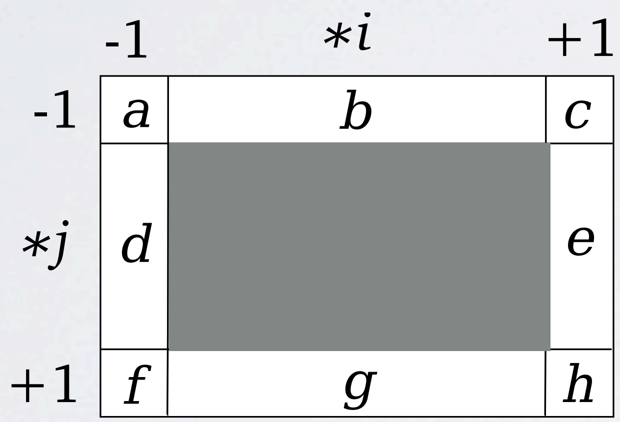
- Special boundary definition syntax
- Pattern match on boundary regions




```
boundary (-1, -1) -> 0.0  -- a
         (*i, -1) -> 0.0  -- b
         (+1, -1) -> 0.0  -- c
         (-1, *j) -> 0.0  -- d
         (+1, *j) -> 0.0  -- e
         (-1, +1) -> 0.0  -- f
         (*i, +1) -> 0.0  -- g
         (+1, +1) -> 0.0  -- h
```

Boundaries

- Special boundary definition syntax
- Pattern match on boundary regions




```
boundary (-1, -1) -> 0.0
         (*i, -1) -> f(i)
         (+1, -1) -> 1.0
         (-1, *j) -> 1.0
         (+1, *j) g -> g...
         (-1, +1) -> 0.0
         (*i, +1) -> 0.0
         (+1, +1) -> 0.0
```

- Permits complicated boundaries: wrapping, reflection

Boundaries (2)

```
boundary (-1, -1) -> 0.0
         (*i, -1) -> 0.0
         (+1, -1) -> 0.0
         (-1, *j) -> 0.0
         (+1, *j) -> 0.0
         (-1, +1) -> 0.0
         (*i, +1) -> 0.0
         (+1, +1) -> 0.0
```

Abbreviated form

```
boundary from (-1, -1) to (+1, +1) -> 0.0
```

Laplace Example

```
[dimensions| X, Y |]

laplace2D = [fun|  X*Y:| _  t  _ |
                       | l @c  r |
                       | _  b  _ | -> t + l + r + b - 4.0*c  |]

lapBoundary = [boundary| Double (-1, *j) -> 0.0
                                (1, *j)  -> 0.0
                                (*i, -1) -> 0.0
                                (*i, +1) -> 0.0 |]

grid = listGrid (Dim X :* Dim Y) (0, 0) (w, h) img_data lapBoundary
grid' = run laplace2D grid
```

Macros (Haskell Quasiquoting):

```
[dimensions| ... |]
[boundary| ... |]
[fun| ... |]
```

Enforcing Safety Invariant

- Encode access pattern of stencils in types
- Encode boundary information in types
- Check boundaries define enough elements for stencils to always have defined values
- Lots of use of GADTs, type families, class constraints

Grid Patterns (Translation)

$$index :: \underbrace{i}_{} \rightarrow Grid\ d\ b\ a \rightarrow a$$

Type-level representation of a relative index

Tuple of type-level integers (inductive)

$$\begin{array}{rcl}
\vdots & & \vdots \\
\text{-2} & \leadsto & Neg\,(S\,(S\,Z)) \\
\text{-1} & \leadsto & Neg\,(S\,Z) \\
0 & \leadsto & Pos\,Z \\
1 & \leadsto & Pos\,(S\,Z) \\
2 & \leadsto & Pos\,(S\,(S\,Z)) \\
\vdots & & \vdots
\end{array}$$

e.g.

$$(-1, +2)$$

represented as value

$$(Neg\,(S\,Z), Pos\,(S\,(S\,Z)))$$

with type

$$(IntT(Neg\ (S\,Z)), IntT(Pos\ (S\,(S\,Z))))$$

Grid Patterns (Translation)

```
f | l @c r | = ...
```

$\leadsto$

```
f = (λg -> let c = index (Pos Z) g
               l = index (Neg (S Z)) g
               r = index (Pos (S Z)) g
           in ...)
```

Grid Patterns (Translation)

$$index :: \underbrace{Safe\ i\ b}_{\text{Haskell type constraint: } C \Rightarrow \tau} \Rightarrow i \rightarrow Grid\ d\ b\ a \rightarrow a$$

- (Binary) predicate/relation $Safe$ enforces safety
- Checks an index against the *boundary* information of a grid: b
- $index$ implemented without bounds checking (i.e. unsafe)

Boundary (Translation)

```
boundary (-1, -1) -> 0.0
         (*i, -1) -> 0.0
         ...
```

⇝ Translates into special list structure (GADT)

```
(ConsB (Static (λ(Neg (S Z), Neg (S Z)) -> 0.0))
 (ConsB (Static (λ(i, Neg (S Z)) -> 0.0)) ...
```

`::` Type encodes the boundary regions described

```
BoundaryList (Cons (Neg (S Z), Neg (S Z))
                   Cons (Int, Neg (S Z)) ...) Static ...
```

Details in paper!

Boundary (Translation)

```
BoundaryList (Cons (Neg (S Z), Neg (S Z))
                 Cons (Int, Neg (S Z)) ...) Static ...
```

Added to a grid's type when grid constructed with boundary

$$Grid\ dim\ bounds\ a$$

e.g.

$$:: Grid\ (X \times Y)\ (Cons\ (Neg(SZ), Neg(SZ))\ (Cons\ (Int, Neg(SZ))\ \ldots))\ a$$

Enforcing Safety

```
f | l @c r | = ...
```

$\rightsquigarrow$

```
f = (λg -> let c = index (Pos Z) g
               l = index (Neg (S Z)) g
               r = index (Pos (S Z)) g
           in ...)
```

```
f :: (Safe (Pos Z) b,
      Safe (Pos (S Z)) b,
      Safe (Neg (S Z)) b) =>
     Grid X b Double → Double
f | l @c r | = (l+c+r)/3.0
```

Cannot apply **`f`** to a grid if the grid does not have satisfactory boundaries

(cut to demo)

Results

- **Correctness - Discover bugs at compile time!**
- Laplace (512x512) (Haskell vs. Ypnos)

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad \cong 5\%$$ slowdown per iteration

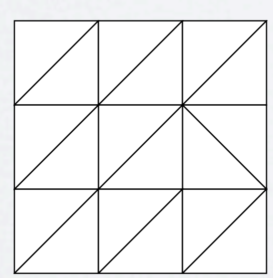
- Laplacian of Gaussians (512x512)

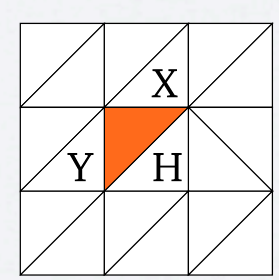
$$\begin{bmatrix} 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & -2 & -1 & 0 \\ -1 & -2 & 16 & -2 & -1 \\ 0 & -1 & -2 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 \end{bmatrix} \quad \cong 3\%$$ speedup per iteration

- Better speedups with more stencils

Further Work

- Parallel implementation
- Output C, CUDA/OpenCL, etc.
- Mechanisms for better error messages
- Many scientific applications: triangle/polygon meshes for better 2D surface of 3D shapes.





https://github.com/dorchard/ypnos
(Previous publication here http://dorchard.co.uk)

Thank You.

Some thanks:

Max Bolingbroke
Kathy Gray
Ben Lippmeier
Robin Message
Ian McDonnell
Tomas Petricek
Simon Peyton-Jones