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]);

Wednesday, 14 September 2011
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]);

when
    i=0, j=0
Stencil Computations

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 $\implies$ more (decidable) information

• More information $\implies$ 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 $\implies$ 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 **array** and a **cursor**

```
+---+---+---+---+
| a | a | a | a |
+---+---+---+---+
| a | a | a | a |
+---+---+---+---+
| a | a | a | a |
+---+---+---+---+
| a | a | a | a |
+---+---+---+---+
```

```
<table>
<thead>
<tr>
<th></th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td></td>
</tr>
</tbody>
</table>
```

- "cursor" (current index under consideration)

- Grid type constructor encodes information

Grid dim bounds a

Type-level dimensionality

e.g.  $X \times Y$

Wednesday, 14 September 2011
Grids

- Grid value comprise an array and a cursor

```
+---+---+---+---+
|   |   |   | a |
+---+---+---+---+
| a |   |   | a |
+---+---+---+---+
| a | a | a | a |
+---+---+---+---+
| a | a | a | a |
+---+---+---+---+
| a | a | a | a |
+---+---+---+---+
```

"cursor" (current index under consideration)

- Grid type constructor encodes information

\[ \text{Grid dim bounds a} \]

Type-level boundary info
Grids

- Grid value comprise an array and a cursor

```
  a a a a
  a a a a
  a a a a
  a a a a
```

“cursor” (current index under consideration)

- Grid type constructor encodes information

\[
\text{Grid dim bounds} \ a
\]

Element type of the grid
Grid Patterns

• Special kind of pattern match on *Grid* values

\[
f | @c | = \ldots
\]

Grid pattern

\[
f | l @c r | = \ldots
\]

Grid pattern

\[
\text{c bound to the “cursor” element}
\]
\[
e.g. \quad c = A[i]
\]

\[
\text{l bound to left of “cursor”}
\]
\[
e.g. \quad l = A[i-1]
\]
\[
\text{r bound to right of “cursor”}
\]
\[
e.g. \quad r = A[i+1]
\]
Grid Patterns (2)

\[ f :: \text{Grid} \times b \text{ Double} \rightarrow \text{Double} \]
\[ f \mid l @c r \mid = (l+c+r)/3.0 \]

Computes average of neighbours

Stencil function (kernel)

\[
\begin{align*}
\text{run} & :: (\text{Grid} \; d \; b \; x \rightarrow y) \rightarrow \text{Grid} \; d \; b \; x \rightarrow \text{Grid} \; d \; \{\} \; y \\
\text{runA} & :: (\text{Grid} \; d \; b \; a \rightarrow a) \rightarrow \text{Grid} \; d \; b \; a \rightarrow \text{Grid} \; d \; b \; a
\end{align*}
\]

Applies a stencil function at every possible “cursor” position

\[ g' = \text{run} \; f \; g \]

*Note: related to extension operation of a comonad*
Grid Patterns (3)

\[ f_{X:| l @c r |} = \ldots \]

Dimension annotation

Two-dimensional syntactic sugar

\[
\text{laplace2D (X*Y):} \mid \_ \_ t _ \_ \mid = t + l + r + b - 4*c
\]

\[
\mid l @c r \mid
\]

\[
\mid _ _ b _ \mid
\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]
Boundaries

• Special boundary definition syntax
• Pattern match on boundary regions
Boundaries

• Special boundary definition syntax
• Pattern match on boundary regions

• Permits complicated boundaries: wrapping, reflection
Boundaries (2)

\[
\begin{align*}
\text{boundary} & \ (-1, -1) \rightarrow 0.0 \\
\text{boundary} & \ (*i, -1) \rightarrow 0.0 \\
\text{boundary} & \ (+1, -1) \rightarrow 0.0 \\
\text{boundary} & \ (-1, *j) \rightarrow 0.0 \\
\text{boundary} & \ (+1, *j) \rightarrow 0.0 \\
\text{boundary} & \ (-1, +1) \rightarrow 0.0 \\
\text{boundary} & \ (*i, +1) \rightarrow 0.0 \\
\text{boundary} & \ (+1, +1) \rightarrow 0.0
\end{align*}
\]

Abbreviated form

\[
\text{boundary from (-1, -1) to (+1, +1)} \rightarrow 0.0
\]
Laplace Example

Macros (Haskell Quasiquoting):

- `[dimensions| X, Y |]`
- `[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)

\[ \text{index} :: i \to \text{Grid} \, d \, b \, a \to a \]

Type-level representation of a relative index

Tuple of type-level integers (inductive)

\[
\begin{align*}
\vdots & \\
-2 & \mapsto \text{Neg} \,(S\,(S\,Z)) \\
-1 & \mapsto \text{Neg} \,(S\,Z) \\
0 & \mapsto \text{Pos} \,Z \\
1 & \mapsto \text{Pos} \,(S\,Z) \\
2 & \mapsto \text{Pos} \,(S\,(S\,Z)) \\
\vdots & \vdots
\end{align*}
\]

\[ (\text{Neg} \,(S\,Z), \text{Pos} \,(S\,(S\,Z))) \]

represented as value

\[ (-1, +2) \]

with type

\[ (\text{IntT}(\text{Neg} \,(S\,Z)), \text{IntT}(\text{Pos} \,(S\,(S\,Z)))) \]
Grid Patterns (Translation)

\[ f \mid l \mathbin{@} c \mathbin{@} r \mid = \ldots \]

\[
f = (\lambda g \to \text{let } c = \text{index (Pos } Z) g \\
    l = \text{index (Neg (S } Z)) g \\
    r = \text{index (Pos (S } Z)) g \\
    \text{in } \ldots)
\]
Grid Patterns (Translation)

\[ \text{index} :: \text{Safe } i \ b \Rightarrow i \rightarrow \text{Grid } d \ b \ a \rightarrow a \]

Haskell type constraint: \( C \Rightarrow \tau \)

- (Binary) predicate/relation \( \text{Safe} \) enforces safety
- Checks an index against the boundary information of a grid: \( b \)
- \( \text{index} \) implemented without bounds checking (i.e. unsafe)
Boundary (Translation)

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

\[ \downarrow \]

Translates into special list structure (GADT)

(ConsB (Static (\(\lambda\)(Neg (S Z), Neg (S Z)) -> 0.0))
(ConsB (Static (\(\lambda\)(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!
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

\[ \text{Grid dim bounds a} \]

e.g.
\[ \vdash \text{Grid} (X \times Y) (\text{Cons} (\text{Neg}(SZ), \text{Neg}(SZ))
  (\text{Cons} (\text{Int}, \text{Neg}(SZ)) ...)) \ a \]
Enforcing Safety

\[ f \mid l @c r \mid = \ldots \]

\[ f = (\lambda g \to \text{let } c = \text{index (Pos Z)} g \]
\[ l = \text{index (Neg (S Z)) } g \]
\[ r = \text{index (Pos (S Z)) } g \]
\[ \text{in } \ldots ) \]

\[ f :: (\text{Safe (Pos Z)} b, \]
\[ \text{Safe (Pos (S Z)) } b, \]
\[ \text{Safe (Neg (S Z)) } b) \Rightarrow \]
\[ \text{Grid X b Double } \to \text{Double} \]
\[ f \mid l @c r \mid = (l+c+r)/3.0 \]

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}
  \approx 5\% \text{ 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}
  \approx 3\% \text{ 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/ynpos
(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