The name of Portland's best known art district, The Pearl, suggests urban legend. Perhaps an oyster canning factory once sat amidst the aging warehouses, or Chinese seafarers hid pearls beneath cobblestoned Twelfth Street. Whatever the origin, there's the suggestion of both beauty and ugliness in the name—there's an elegant gem nestled in a drab, rough shell.

The story goes like this: Thomas Augustine, a local gallery owner, coined the phrase more than 10 years ago to suggest that the buildings in the warehouse district were like crusty oysters, and that the galleries and artists' lofts within were like pearls. "There were very few visible changes in the area," says Al Solheim, a developer who has been involved in many projects in the district. "People would drive by and not have a clue as to what was inside. A local businesspeople were looking to update the ground floor buildings, and the warehouse district, or the "Pearl district," was an obvious choice. People would love the idea of live work spaces, so we very few visible changes in the area," says Al Solheim, a developer who has been involved in many projects in the district. "People would drive by and not have a clue as to what was inside. A local businesspeople were looking to update the ground floor buildings, and the warehouse district, or the "Pearl district," was an obvious choice. People would love the idea of live work spaces, so we borrowed the name."

Despite initial criticism about the name, few deny that it's catchy. The Portland Institute for Contemporary Art (PICA)'s inventive announcement for its 1998 annual Dada Ball included a tuna can with a fake pearl inside.
How to Write a Functional Pearl

Richard Bird
ICFP, Portland, Oregon, 2006
and how your experiences as to what makes a good one and how complicated, or somehow not quite the elegant solution somehow miss the mark, by being too trivial, too ICFP conference, but many of the submitted ones Well done Functional Pearls are often the highlight of an

My brief from the Program Chair
What is a functional pearl?

Polished, elegant, instructive, and entertaining.

So, pearls are...

"It is not enough simply to describe a program!"

Some previous calls added an off-putting sentence:

"Elegant new ways of approaching a problem..."

...Pearls need not report original research results; they need not present reusable programming idioms or original research.

"Functional pearls: Elegant, instructive examples of functional programming."

Recent ICFP calls for papers have said:

"What is a functional pearl?"
Origins

In 1990, when JFP was being planned, I was asked by the then editors, Simon Peyton Jones and Philip Wadler, to contribute a regular column to be called Functional Pearls. The idea they had in mind was to emulate the very successful series of essays that John Bentley had written in the 1980s under the title Programming Pearls in the CACM. The idea they had in mind was to emulate the very successful Programming Pearls. The programs are fun, and they teach grown from real problems that have irritated programmers. The Programming Pearls that have irritated oysters, those Programming Pearls have grown from grains of sand that just as natural pearls grow from grains of sand that irritated oysters. "Just as natural pearls grow from grains of sand that irritated oysters, these Programming Pearls have grown from real problems that have irritated programmers. The programs are fun, and they teach important programming techniques and fundamental design principles."

Bentley wrote about his pearls:

"Just as natural pearls grow from grains of sand that irritated oysters, these Programming Pearls have grown from real problems that have irritated programmers. The programs are fun, and they teach important programming techniques and fundamental design principles."
Why me?

One major reason that functional programming simulated

Because I was a GOFER man.

Good for Equational Reasoning.

Perhaps, the editors no doubt thought, I could give examples of

Less obvious but more efficient programs:

Perhaps, the editors no doubt thought, I could give examples of

My personal research agenda: to study the extent to which the

whole arsenal of efficient algorithm design techniques can be

expressed, organised, taught and communicated through the laws

of functional programming.

One major reason that functional programming simulated the

Because I was a GOFER man.

Why me?
The state of play

Some 64 pearls will have appeared in JFP by the end of 2006; also a sprinkling of pearls at ICFP and MPC; some 64 pearls will have appeared in JFP by the end of 2006;

Pearls contain:

- Instructive examples of program calculation or proof;
- Nifty presentations of old or new data structures;
- Interesting applications and programming techniques;


Also a special issue in JFP, 2004 devoted to pearls;

Also a sprinkling of pearls at ICFP and MPC;

Also a collection in The Fun of Programming, edited by J.
I send out each pearl for review, including my own. Reviewers are instructed to stop reading when they get bored.

Some pearls are better served as standard research papers.

They get bored;

Too much specialist knowledge is needed;

The material gets too complicated;

The writing is bad.

Most need more time in the oyster.
Advice

Throw away the rule book for writing research papers;
Get in quick, get out quick;
Be self-contained, no long lists of references and related work;
Get in quick, get out quick;
Be engaging;
Remember, writing and reading are joint ventures;
You are telling a story, so some element of surprise is welcome;
Find an author whose style you admire and copy it (my personal favourites are Martin Gardner and Don Knuth).
Take it out and polish it again.

Put the pearl away for a couple of months at least.

Consider using the new order in the next draft.

If you changed the order of presentation for the talk,

Give a talk on the pearl to non-specialists, your students,

More advice
Advice

"Whatever advice you give, be short." Horace

"The only thing to do with good advice is to pass it on. It is never of any use to oneself." Oscar Wilde

"I owe my success to having listened respectfully to the very best advice, and then going away and doing the exact opposite." G.K. Chesterton

"The only thing to do with good advice is to pass it on. It is never of any use to oneself." Oscar Wilde

"Whatever advice you give, be short." Horace

Advice on advice
A Simple Sudoku Solver
A quote from The Independent Newspaper

How TO PLAY Fill in the grid so that every row, every column and every 3 x 3 box contains the digits 1 - 9. There's no maths involved. You solve the puzzle with reasoning and logic.
Our aim is to define a function:

\[
\text{solve} :: \text{Grid} \rightarrow \text{[Grid]}
\]

Our aim is to define a function to solve a function `Grid -> [Grid]`.
We suppose that the given grid contains only digits and blanks.

```haskell
< blank :: Digit -> Bool
< blank = (== '0')
< digits :: [Digit]
< digits = ['1'..'9']
< digits :: [Digit]
< type Digit = Char
< type Grid = Matrix Digit
< type Matrix a = [Row a]
< type Row a = [a]
```

Basic data types
Here is the specification:

\[ \text{solve1} :: \text{Grid} \rightarrow [\text{Grid}] \]
\[ \text{solve1} = \text{filter} \circ \text{valid} \circ \text{expand} \circ \text{choices} \]

In words: first install all possible choices for the blank entries, then compute all grids that arise from making every possible choice, then return only the valid grids.

The types:

\[ \text{valid} :: \text{Grid} \rightarrow \text{Bool} \]
\[ \text{expand} :: \text{Matrix Choices} \rightarrow [\text{Grid}] \]
\[ \text{choices} :: \text{Grid} \rightarrow \text{Matrix Choices} \]

Here is the specification:
Installing choices

The simplest choice of choices is

```haskell
choices = map (map choice) $ choices :: Grid -> Matrix Choices
```

Then we have

```haskell
| otherwise = [d] |
where choice d | blank d = digits
  choices = map (map choice)
```

The simplest choice of choices is

```haskell
> type Choices = [Digit]
```

Installing choices
Expansion is just matrix cartesian product:

```
[[[]]] map f ⍳ ⍵ = expand ⍳ ⍵
```

The cartesian product of a list of lists is given by:

```
expand = map ⍳ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ ⍵ }"
A valid grid is one in which no row, column or box contains duplicates. That leaves the definition of rows, cols, and boxes.

We omit the definition of nodups.

That leaves the definition of nodups.

\[
\text{valid} = \text{allnodups(rows } g) \land \text{allnodups(cols } g) \land \text{allnodups(boxs } g) \Rightarrow \text{Grid } g \Rightarrow \text{Bool}
\]

Valid grids
Rows, columns and boxes

```haskell
xs :: [a] = [x,y,z] :: [x:y:z:xs] = split ['] = split [] = concat

split . map split . map cols . unsplit . split . map cols . split = boxes = map unsplit . split . map cols

Boxes is just a little more interesting:

cols = foldr (zipWith (:)) (repeat [])

rows = id

[rows, cols, boxes :: Matrix a => Row a]
```
Wholemeal programming. Instead of thinking about coordinates, and encountering lawful programming is good for you. It helps to prevent a disease called indexitis, and encourages lawful programming.

Geraint Jones has aptly called this style "Wholemeal Programming." Functions that treat the matrix as a complete entity in itself. Arithmetic on subscripts to extract information about rows, columns and boxes; we have gone for definitions of these instead of thinking about coordinate systems, and doing...
We will make use of these laws in a short while.

map boxes . expand = expand . boxes
map cols . expand = expand . cols
map rows . expand = expand . rows

Here are three more, valid on $\mathbb{N}^2 \times \mathbb{N}^2$ matrices of choices:

boxes . id = id
cols . id = id
rows . id = id

For example, here are three laws that are valid on $\mathbb{N}^2 \times \mathbb{N}^2$.
The following laws concern \( \text{filter} \):

- \( \text{filter} \circ \text{concat} = \text{concat} \circ \text{map} \circ \text{filter} \)
- \( \text{filter} \circ \text{all} = \text{map} \circ \text{filter} \circ \text{id} \)
- If \( f \cdot f = \text{id} \), then

The following laws concern \( \text{filter} \):

Three more laws

We will also make use of these laws in due course.
Pruning a matrix of choices

Though executable in theory, the specification is hopeless in practice. To make a more efficient solver, a good idea is to remove any choices from a cell that already occur as single entries in the row, column, and box containing the cell. A good idea is to remove any choice that already occurs as a single entry in the row, column, and box containing the cell.

How would you define `prune`?

```
prune :: Matrix Choices -> Matrix Choices
```

We therefore seek a function `prune` that, given a matrix of choices, removes any choices that already occur as single entries in the row, column, and box containing the cell. A good idea is to remove any choice that already occurs as a single entry in the row, column, and box containing the cell.
The function \textit{pruneRow} satisfies

\begin{verbatim}
filternodups.cp.pruneRow = filternodups.cp.
pruneRow = \#
\end{verbatim}

\textit{Pruning a row}
We send each of these filters one by one into battle with expand.

\[
\text{filter (all nodups : rows)} \cdot \text{expand}
\]
\[
\text{filter (all nodups : cols)} \cdot \text{filter (all nodups : boxes)}
\]
= \text{filter valid} \cdot \text{expand}

We have

\[
\text{filter valid} \cdot \text{expand} \cdot \text{prune}
\]
= \text{filter valid} \cdot \text{expand} \cdot \text{prune}

Remember, we want

Calculation
Let $f \in \text{rows, cols, boxes}$ and abbreviate noudups to p:

\[
\begin{align*}
\text{filter} & \circ \text{cp} \circ \text{map} (\text{filter} \circ \text{cp} \circ \text{pruneRow}) \cdot f \\
& = \text{map} f \circ \text{cp} \circ \text{map} (\text{filter} \circ \text{cp} \circ \text{pruneRow}) \\
& \quad \circ \text{property of pruneRow} \\
& = \text{map} f \circ \text{cp} \circ \text{map} (\text{filter} \circ \text{cp} \circ \text{pruneRow}) \cdot f \\
& \quad \circ \text{law of filter and cp} \\
& = \text{map} f \cdot \text{filter} (\text{id} \circ \text{p} \circ \text{cp}) \circ \text{map} f \circ \text{cp} \\
& \quad \circ \text{definition of expand} \\
& = \text{map} f \cdot \text{filter} (\text{id} \circ \text{p} \circ \text{cp}) \circ \text{expand} \\
& \quad \circ \text{since map} f \cdot \text{expand} = \text{expand} \circ \text{map} f \\
& \quad \circ \text{since} f \cdot f = \text{id} \\
& \quad \circ \text{filter} (\text{id} \circ \text{p} \circ f) \circ \text{expand}
\end{align*}
\]
Going backwards!

\[
\text{filter} \left( \text{all} \ p \ . \ f \right) \cdot \text{expand} \cdot \text{prune} \ f
\]

\[
\text{filter} \left( \text{all} \ p \ . \ f \right) \cdot \text{expand} \cdot \text{prune} \ f
\]

Hence

\[
\text{filter} \left( \text{all} \ p \ . \ f \right) \cdot \text{expand} \cdot \text{prune} \ f
\]

\[
\{ \text{definition of expand} \} = \text{filter} \left( \text{all} \ p \ . \ f \right) \cdot \text{expand} \cdot \text{map prune row} \cdot \ f
\]

\[
\{ \text{since map prune row} = \text{filter} \} = \text{filter} \left( \text{all} \ p \ . \ f \right) \cdot \text{expand} \cdot \text{map prune row} \cdot \ f
\]

\[
\{ \text{definition of expand} \} = \text{filter} \left( \text{all} \ p \ . \ f \right) \cdot \text{expand} \cdot \text{cp} \cdot \text{map prune row} \cdot \ f
\]

\[
\{ \text{law of filter and cp} \} = \text{filter} \left( \text{all} \ p \ . \ f \right) \cdot \text{cp} \cdot \text{map prune row} \cdot \ f
\]
In fact, we can have as many prunes as we like.

After a tad more equational reasoning, we obtain

\[
\text{solve} = \text{filter valid} \cdot \text{expand} \cdot \text{prune} \cdot \text{choices}
\]

Now we have a second version of the program:

\[
\text{solve} = \text{filter valid} \cdot \text{expand} \cdot \text{prune} \cdot \text{choices}
\]
Single-cell expansion

The simplest Sudoku problems are solved by repeatedly pruning
the matrix of choices until only singleton choices are left.

```
expand = concat . map expand . expand

Suppose we define a function
```

**Simple idea**: single-cell expansion.

For more devious puzzles we can combine pruning with another

The simplest Sudoku problems are solved by repeatedly pruning
```
Single-cell expansion
A good choice of cell on which to perform expansion is one with a
smallest number of choices, not equal to 1 of course:

 Parsimonious expansion
Under a valid grid, complete and non-blocked matrices correspond to a unique valid grid.

Incomplete but blocked matrices can never lead to valid grids. A

Hence expand = concat . map expand . expand

Say a matrix is complete if each choice contains a singleton, and blocked

Hence

In the singleton choices contain a duplicate, complete, and

Hence

Incomplete but blocked matrices can never lead to valid grids.

expands, and blocked.

Choice, possibly a null choice.

Only holds when applied to matrices with at least one non-single

expands cm = undefined if cm contains only singleton choices.

expands cm = [] if cm contains a null choice;

Properties of parsimonious expansion
Blocked and complete matrices

```plaintext
[ row -> [d] | d ] = dups (rows cm)

any hasdups (boxes cm) || any hasdups (cols cm) || any hasdups (rows cm)

blocked cm = any hasdups (rows cm)

blocked :: MatrixChoices -> Bool

single_ = False

single' = True

[ single' ] = True

complete = 411 (411 single')

complete :: MatrixChoices -> Bool

Blocked and complete matrices
```
Assuming a matrix is non-blocked and incomplete, we have

$$\text{search} = \text{concat} \cdot \text{map} \cdot \text{filter valid} \cdot \text{expand} \cdot \text{prune}$$

we therefore have, on incomplete and non-blocked matrices,

$$\text{search} = \text{filter valid} \cdot \text{expand} \cdot \text{prune}$$

Writing

$$\text{expand} = \text{concat} \cdot \text{map} \cdot \text{filter valid} \cdot \text{expand} \cdot \text{prune}$$

$$\text{expand} = \text{concat} \cdot \text{map} \cdot \text{filter valid} \cdot \text{expand} \cdot \text{prune}$$

$$\text{expand} = \text{filter valid} \cdot \text{concat} \cdot \text{map} \cdot \text{expand} \cdot \text{prune}$$

$$\text{expand} = \text{filter valid} \cdot \text{expand}$$

$$\text{expand} = \text{filter valid} \cdot \text{expand}$$

$$\text{expand} = \text{filter valid} \cdot \text{expand}$$

More calculation
A reasonable Sudoku solver.
I tested the solver on Simon Peyton Jones’ 36 puzzles recorded at
http://haskell.org/haskellwiki/Sudoku
Itsolved them in 8.8 seconds (on a 1GHz Pentium 3 PC).
I also tested them on 6 minimal puzzles (each with 17 non-blank
etries) chosen randomly from the 32000 given at the site.
It solved them in 11.4 seconds.

Tests
Conclusions

There are about a dozen Haskell Sudoku solvers at http://haskell.org/haskellwiki/Sudoku. All of these, including a very nice solver by Lennart Augustsson, deploy coordinate calculations. Many use arrays and most use monads. I know of solvers that reduce the problem to Boolean satisfiability, constraint satisfaction, model checking, and so on.

Mine is about twice as slow as Lennart’s on the nefarious puzzle, but about thirty times faster than Yitz Gale’s solver on easy puzzles. I would argue that mine is certainly one of the simplest and shortest. At least it was derived, in part, by equational reasoning.