FOOD & DRINK
GALLERIES
SHOPPING
REAL ESTATE
SERVICES
ABOUT THE DISTRICT
CLASSES & EVENTS
MAPS & DIRECTIONS

Virtual Tour of the Pearl

Back

Home>Recent Press>Pearl Walking Map>Special Events Venues
First Thursday>PDBA Members>Contact Us>Site Map

FOOD & DRINK
GALLERIES
SHOPPING
REAL ESTATE
SERVICES
ABOUT THE DISTRICT
Virtual Tour
History
Magazine
Recent Press
Pearl Walking Map
Portland Streetcar
Neighborhood Association
Business Association
CLASSES & EVENTS
MAPS & DIRECTIONS

History of the Pearl

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 cobble stoned Twelfth Street. Whatever the origin, there's the suggestion of both beauty and ugliness in the name—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 Solhiem, 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." As local business people were looking to label the growing area—the "warehouse district" or the "brewery district" were two suggestions—an Alaska Airlines writer borrowed Augustine's phrase, according to Solheim. The name stuck.

"Everyone hated it," says Pulliam Deffenbaugh Gallery owner Rod Pulliam, who opened his gallery 10 years ago. Few other galleries, such as Quartersaw and Blackfish, have histories that go back that far. But many artists lived or worked in the area in loft buildings such as the Maddox on Hoyt Street. Back then, says Pulliam, light industry, vacant buildings, and blue collar cafes outnumbered the galleries and lofts.

Despite initial cynicism 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.

Home>Recent Press>Pearl Walking Map>Special Events Venues
First Thursday>PDBA Members>Contact Us>Site Map

How to Write a Functional Pearl

ICFP, Portland, Oregon, 2006

Richard Bird

My brief from the Program Chair

"Well done Functional Pearls are often the highlight of an ICFP conference, but many of the submitted ones somehow miss the mark, by being too trivial, too complicated, or somehow not quite the elegant solution one hopes for. So it would be interesting to hear about your experiences as to what makes a good one and how to go about creating it."

What is a functional pearl?

Recent ICFP calls for papers have said:

> "**Functional pearls**: Elegant, instructive examples of functional programming.
> ... Pearls need not report original research results; they may instead present re-usable programming idioms or elegant new ways of approaching a problem."

Some previous calls added an off-putting sentence:

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

So, pearls are

polished   elegant   instructive   entertaining

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 Jon Bentley had written in the 1980s under the title *Programming Pearls* in the C. ACM.

Bentley wrote about his pearls:

> "Just as natural pearls grow from grains of sand that have 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?

Because I was a **GOFER** man.

One major reason that functional programming stimulated the interest of many at that time was that it was

GOod For Equational Reasoning.

Perhaps, the editors no doubt thought, I could give examples of GOFER-ing a clear but inefficient functional specification into a less obvious but more efficient program?

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.

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;
- Special issue in JFP, 2004 devoted to pearls;
- Also a collection in *The Fun of Programming*, edited by J. Gibbons and O. de Moor, Palgrave, 2003.

Pearls contain:

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

Reviewing for JFP

I send out each pearl for review, including my own. Reviewers are instructed to stop reading when

- They get bored;
- The material gets too complicated;
- Too much specialist knowledge is needed;
- The writing is bad.

Some pearls are better serviced as standard research papers. 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;
- 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*).

More advice

- Give a talk on the pearl to non-specialists, your students, your department.
- If you changed the order of presentation for the talk, consider using the new order in the next draft;
- Put the pearl away for a couple of months at least;
- Take it out and polish it again.

Advice on 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

A Simple Sudoku Solver

ICFP, Portland, Oregon, 2006

Richard Bird

A quote from *The Independent* Newspaper

HOW TO PLAY Fill in the grid so that every row, every column and every 3×3 box contains the digits 1 - 9. There's no maths involved. You solve the puzzle with reasoning and logic.

Our aim

Our aim is to define a function

```
solve :: Grid -> [Grid]
```

for filling in a grid correctly in all possible ways.

We begin with a specification, then use equational reasoning to calculate a more efficient version.

No ‘maths’, no monads: just wholesome, pure - and lazy functional programming.

Basic data types

```
> type Matrix a = [Row a]
> type Row a    = [a]

> type Grid     = Matrix Digit
> type Digit    = Char

> digits  :: [Digit]
> digits  =  ['1'..'9']

> blank   :: Digit -> Bool
> blank   =  (== '0')
```

We suppose that the given grid contains only digits and blanks.

Specification

Here is the specification:

```
> solve1 :: Grid -> [Grid]
> solve1 = filter valid . expand . 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:

```
choices :: Grid -> Matrix Choices
expand  :: Matrix Choices -> [Grid]
valid   :: Grid -> Bool
```

Installing choices

The simplest choice of `Choices` is

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

Then we have

```
> choices :: Grid -> Matrix Choices
> choices = map (map choice)
>  where choice d | blank d   = digits
>                 | otherwise = [d]
```

Expansion

Expansion is just matrix cartesian product:

```
> expand :: Matrix Choices -> [Grid]
> expand = cp . map cp
```

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

```
> cp :: [[a]] -> [[a]]
> cp = foldr op [[]]
> op xs yss = [x:ys | x <- xs, ys <- yss]
```

Valid grids

A valid grid is one in which no row, column or box contains duplicates.

```
> valid  :: Grid -> Bool
> valid g = all nodups (rows g) &&
>           all nodups (cols g) &&
>           all nodups (boxs g)
```

We omit the definition of `nodups`.

That leaves the definition of `rows`, `cols`, and `boxs`.

Rows, columns and boxs

```
> rows, cols, boxs :: Matrix a -> [Row a]
```

```
> rows = id
```

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

Boxs is just a little more interesting:

```
> boxs = map unsplit . unsplit . map cols .
>        split . map split
```

```
> unsplit           = concat
> split []          = []
> split (x:y:z:xs) = [x,y,z]:split xs
```

Wholemeal programming

Instead of thinking about coordinate systems, and doing arithmetic on subscripts to extract information about rows, columns and boxes, we have gone for definitions of these functions that treat the matrix as a complete entity in itself.

Geraint Jones has aptly called this style

> Wholemeal Programming

Wholemeal programming is good for you: it helps to prevent a disease called indexitis, and encourages lawful program construction.

Laws

For example, here are three laws that are valid on $N^2 \times N^2$ matrices:

```
rows . rows = id
cols . cols = id
boxs . boxs = id
```

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

```
map rows . expand = expand . rows
map cols . expand = expand . cols
map boxs . expand = expand . boxs
```

We will make use of these laws in a short while.

Three more laws

The following laws concern `filter`:

If `f . f = id`, then

```
filter (p . f) = map f . filter p . map f
```

Secondly,

```
filter (all p) . cp = cp . map (filter p)
```

Thirdly,

```
filter p . concat = concat . map (filter p)
```

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 c that already occur as single entries in the row, column and box containing c.

We therefore seek a function

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

so that

```
   filter valid . expand
 = filter valid . expand . prune
```

How would you define `prune`?

Pruning a row

```
> pruneRow :: Row Choices -> Row Choices
> pruneRow row = map (remove ones) row
>  where ones = [d | [d] <- row]

> remove xs [d] = [d]
> remove xs ds  = ds \\ xs
```

The function `pruneRow` satisfies

```
  filter nodups . cp
= filter nodups . cp . pruneRow
```

Calculation

Remember, we want

```
  filter valid . expand
= filter valid . expand . prune
```

We have

```
  filter valid . expand
= filter (all nodups . boxs) .
  filter (all nodups . cols) .
  filter (all nodups . rows) . expand
```

We send each of these filters one by one into battle with `expand`.

GOFER it!

Let `f` ∈ `{rows,cols,boxs}` and abbreviate `nodups` to `p`:

```
  filter (all p . f) . expand
=  {since f . f = id}
  map f . filter (all p) . map f . expand
=  {since map f . expand = expand . f}
  map f . filter (all p) . expand . f
=  {definition of expand}
  map f . filter (all p) . cp . map cp . f
=  {law of filter and cp}
  map f . cp . map (filter p . cp) .  f
   {property of pruneRow}
= map f . cp . map (filter p . cp . pruneRow) . f
```

Going backwards!

```
  map f . cp . map (filter p . cp . pruneRow) . f
=  {law of filter and cp}
  map f . filter (all p) . cp . map cp . map pruneRow . f
=  {definition of expand}
  map f . filter (all p) . expand . map pruneRow . f
=  {since f . f = id}
  filter (all p . f) . map f . expand . map pruneRow . f
=  {since map f . expand = expand . f}
  filter (all p . f) . expand . f . map pruneRow . f
=  {introducing pruneBy f = f . map pruneRow . f}
  filter (all p . f) . expand . pruneBy f
```

Hence

```
  filter (all p . f) . expand
= filter (all p . f) . expand . pruneBy f
```

The result

After a tad more equational reasoning, we obtain

```
> prune :: Matrix Choices -> Matrix Choices
> prune =
>  pruneBy boxs . pruneBy cols . pruneBy rows
>  where pruneBy f = f . map pruneRow . f
```

Now we have a second version of the program:

```
> solve2 :: Grid -> [Grid]
> solve2 = filter valid . expand . prune . choices
```

In fact, we can have as many `prunes` as we like.

Single-cell expansion

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

For more devious puzzles we can combine pruning with another simple idea: single-cell expansion.

Suppose we define a function

```
expand1 :: Matrix Choices -> [Matrix Choices]
```

that expands the choices for one cell only. This function is to satisfy the property that, upto permutation of the answer,

```
expand = concat . map expand .  expand1
```

Parsimonious 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:

```
> expand1   :: Matrix Choices -> [Matrix Choices]
> expand1 cm =
>  [rows1 ++ [row1 ++ [c]:row2] ++ rows2 | c <- cs]
>  where
>  (rows1,row:rows2) = break (any smallest) cm
>  (row1,cs:row2)    = break smallest row
>  smallest cs       = length cs == n
>  n                 = minimum (lengths cm)

> lengths = filter (/=1) . map length . concat
```

Properties of parsimonious expansion

- `expand1 cm = []` if `cm` contains a null choice;
- `expand1 cm = undefined` if `cm` contains only single choices.

Hence

```
expand = concat . map expand .  expand1
```

only holds when applied to matrices with at least one non-single choice, possibly a null choice.

Say a matrix is complete if all choices are singletons, and blocked if the singleton choices contain a duplicate.

Incomplete but blocked matrices can never lead to valid grids. A complete and non-blocked matrix of choices corresponds to a unique valid grid.

Blocked and complete matrices

```
> complete :: Matrix Choices -> Bool
> complete = all (all single)

> single [_] = True
> single _   = False

> blocked :: Matrix Choices -> Bool
> blocked cm = any hasdups (rows cm) ||
>              any hasdups (cols cm) ||
>              any hasdups (boxs cm)

> hasdups row = dups [d | [d] <- row]
```

More calculation

Assuming a matrix is non-blocked and incomplete, we have

```
   filter valid . expand
 = filter valid . concat . map expand . expand1
 = concat . map (filter valid . expand) .
   expand1
 = concat . map (filter valid . expand . prune) .
   expand1
```

Writing

```
search = filter valid . expand . prune
```

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

```
search = concat . map search . expand1 . prune
```

A reasonable sudoku solver

```
> solve :: Grid -> [Grid]
> solve =  search . choices

> search :: Matrix Choices -> [Grid]
> search cm
>  |blocked pm  = []
>  |complete pm = [map (map head) pm]
>  |otherwise   = concat $ map search $ expand1 pm
>  where pm = prune cm
```

Tests

I tested the solver on Simon Peyton Jones' 36 puzzles recorded at

`http://haskell.org/haskellwiki/Sudoku`

It solved them in 8.8 seconds (on a 1GHz pentium 3 PC).

I also tested them on 6 minimal puzzles (each with 17 non-blank entries) chosen randomly from the 32000 given at the site.

It solved them in 111.4 seconds.

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.