Assertions, Assertive Coding
+
Application to Sudoku Puzzles

Jan van Eijck
CWI & ILLC, Amsterdam

Specification and Testing, Week 5, 2013

Abstract

We show how to specify preconditions, postconditions, assertions and invariants, and how to wrap these around functional code or functional imperative code. We illustrate the use of this for writing programs for automated testing of code that is wrapped in appropriate assertions. We call this assertive coding. An assertive version of a function f is a function f' that behaves exactly like f as long as f complies with its specification, and aborts with error otherwise. This is a much stronger sense of self-testing than what is called self-testing code (code with built-in tests) in test driven development.

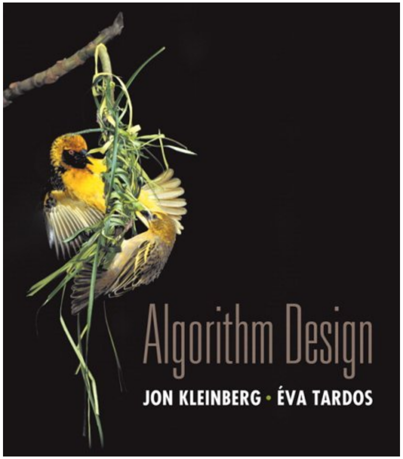
The lecture gives examples of how to use (inefficient) specification code to test (efficient) implementation code, and how to turn assertive code into production code by replacing the self-testing versions of the assertion wrappers by self-documenting versions that skip the assertion tests.

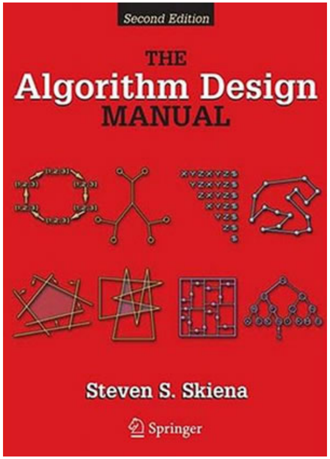
We end with a demonstration of the use of formal methods in developing a sudoku solver.

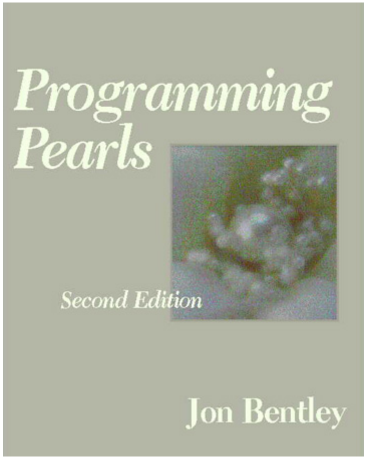
Module Declaration

```
module Week5

where

import Data.List
import Week4
```

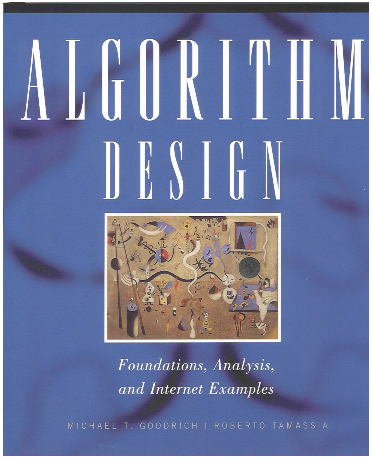
Algorithm Design and Specification: Some excellent books

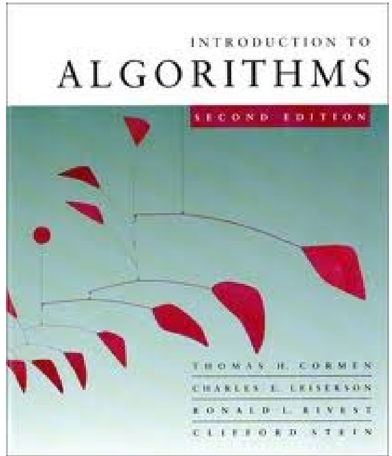


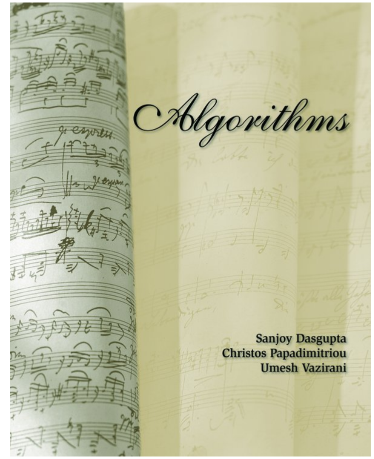




And some more:

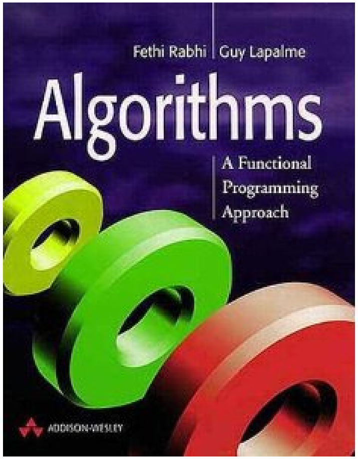


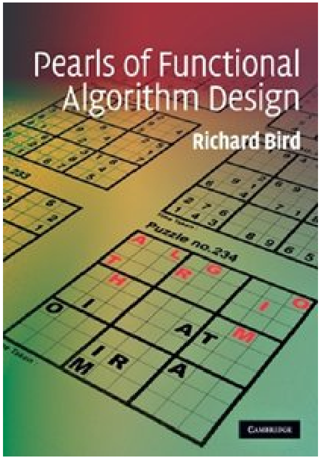


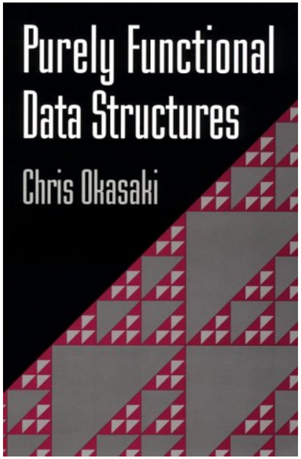


Functional Imperative Algorithm Specification

This course will teach you a purely functional way to look at algorithms as they are designed, presented and analyzed in these books. This complements the approach of [4] and [2], which propose to give 'functional' solutions for 'classical' algorithmic problems. Instead, this course will show that classical algorithmic problems plus their classical solutions can be presented in a purely functional way.







Preconditions, Postconditions, Assertions and Invariants

A (Hoare) assertion about an imperative program [3] has the form

$$\{\text{Pre}\}\ \text{Program}\ \{\text{Post}\}$$

where Pre and Post are conditions on states.

This Hoare statement is true in state s if truth of Pre in s guarantees truth of Post in any state s' that is a result state of performing Program in state s.

One way to write assertions for functional code is as wrappers around functions. This results in a much stronger sense of self-testing than what is called self-testing code (code with built-in tests) in test driven development [1].

Precondition Wrappers

The precondition of a function is a condition on its input parameter(s), the postcondition is a condition on its value.

Here is a precondion wrapper for functions with one argument. The wrapper takes a precondition property and a function and produces a new function that behaves as the old one, provided the precondition is satisfied,

```
pre1 :: (a -> Bool) -> (a -> b) -> a -> b
pre1 p f x = if p x then f x
             else error "pre1"
```

Postcondition Wrappers

A postcondition wrapper for functions with one argument.

```
post1 :: (b -> Bool) -> (a -> b) -> a -> b
post1 p f x = if p (f x) then f x
              else error "post1"
```

Example

```
decomp :: Integer -> (Integer,Integer)
decomp n = decomp' (0,n) where
  decomp' = while1 (\ (_,m) -> even m)
                   (\ (k,m) -> (k+1,m `div` 2))
```

Here is an equivalent recursive version:

```
decomp :: Integer -> (Integer,Integer)
decomp n = decomp' (0,n) where
  decomp' (k,m) | odd m = (k,m)
                | otherwise = decomp' (k+1,m `div` 2)
```

Specifying a Postcondition

The `decomp` function, when applied to n, should compute a pair (k, m) with $n = 2^k \times m$, with m odd.

Put the second requirement in the postcondition:

```
decompPost = post1 (\ (_,m) -> odd m) decomp
```

Note that `decompPost` has the same type as `decomp`, and that the two functions compute the same number pair whenever `decompPost` is defined (i.e., whenever the output value of `decomp` satisfies the postcondition).

Assertions

More generally, an assertion is a condition that may relate input parameters to the computed value. Here is an assertion wrapper for functions with one argument. The wrapper wraps a binary relation expressing a condition on input and output around a function and produces a new function that behaves as the old one, provided that the relation holds.

```
assert1 :: (a -> b -> Bool) -> (a -> b) -> a -> b
assert1 p f x = if p x (f x) then f x
                else error "assert1"
```

Example use:

```
decompA = assert1 (\ n (k,m) -> n == 2^k*m) decomp
```

Note that `decompA` has the same type as `decomp`. Indeed, as long as the assertion holds, `decompA` and `decomp` compute the same value.

Invariants

An invariant of a program P in a state s is a condition C with the property that if C holds in s then C will also hold in any state that results from execution of P in s. Thus, invariants are Hoare assertions of the form:

$$\{\mathrm{C}\}\ \mathrm{Program}\ \{\mathrm{C}\}$$

If you wrap an invariant around a step function in a loop, the invariant documents the expected behaviour of the loop.

Invariant Wrappers

First an invariant wrapper for the case of a function with a single parameter: a function `f :: a -> a` fails an invariant `p :: a -> Bool` if the input of the function satisfies p but the output does not:

```
invar1 :: (a -> Bool) -> (a -> a) -> a -> a
invar1 p f x =
  let
    x' = f x
  in
  if p x && not (p x') then error "invar1"
  else x'
```

Two Examples

Example of an invariant wrap:

```
gSign = invar1 (>0) (while1 even (`div` 2))
```

Another example:

```
gSign' = invar1 (<0) (while1 even (`div` 2))
```

Use of Invariant Inside While Loop

We can also use the invariant inside a while loop:

```
decompInvar :: Integer -> (Integer,Integer)
decompInvar n = decomp' (0,n) where
  decomp' = while1 (\ (_,m) -> even m)
                   (invar1
                     (\ (i,j) -> 2^i*j == n)
                     (\ (k,m) -> (k+1,m `div` 2)))
```

An Assertive List Merge Algorithm

Consider the problem of merging two sorted lists into a result list that is also sorted, and that contains the two original lists as sublists.

Implication Operator

For writing specifications an operator for Boolean implication is good to have.

```
infix 1 ==>

(==>) :: Bool -> Bool -> Bool
p ==> q = (not p) || q
```

Sorted Property

The specification for merge uses the following property:

```
sortedProp :: Ord a => [a] -> [a] -> [a] -> Bool
sortedProp xs ys zs =
  (sorted xs && sorted ys) ==> sorted zs

sorted :: Ord a => [a] -> Bool
sorted [] = True
sorted [_] = True
sorted (x:y:zs) = x <= y && sorted (y:zs)
```

Sublist Property

Each list should occur as a sublist in the merge:

```
sublistProp :: Eq a => [a] -> [a] -> [a] -> Bool
sublistProp xs ys zs =
  sublist xs zs && sublist ys zs

sublist :: Eq a => [a] -> [a] -> Bool
sublist [] _ = True
sublist (x:xs) ys =
  elem x ys && sublist xs (ys \\ [x])
```

Assertion wrapper for functions with two parameters

```
assert2 ::  (a -> b -> c -> Bool)
             -> (a -> b -> c) -> a -> b -> c
assert2 p f x y =
  if p x y (f x y) then f x y
  else error "assert2"
```

A merge function

```
merge :: Ord a => [a] -> [a] -> [a]
merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys) = if x <= y
                         then x : merge xs (y:ys)
                         else y : merge (x:xs) ys
```

And an assertive version of the `merge` function:

```
mergeA :: Ord a => [a] -> [a] -> [a]
mergeA = assert2 sortedProp
             $ assert2 sublistProp merge
```

Wrap Arond Wrap

We have wrapped an assertion around a wrap of an assertion around a function. This cause no problems, for the wrap of an assertion around a function has the same type as the original function.

Note that `sortedProp` is an implication. If we apply test-merge to a list that is not sorted, the property still holds:

```
*AssertiveCoding> mergeA [2,1] [3..10]
[2,1,3,4,5,6,7,8,9,10]
```

Definition of GCD and the ‘divides’ relation

The definition of GCD is given in terms of the divides relation. An integer n divides another integer m if there is an integer k with $nk = m$, in other words, if the process of dividing m by n leaves a remainder 0.

```
divides :: Integer -> Integer -> Bool
divides n m = rem m n == 0
```

The GCD Definition Implemented

An integer n is the GCD of k and m if n divides both k and m, and every divisor of k and m also divides n.

```
isGCD :: Integer -> Integer -> Integer -> Bool
isGCD k m n = divides n k && divides n m &&
              forall [1..min k m]
              (\ x -> (divides x k && divides x m)
                       ==> divides x n)
```

The Extended GCD Algorithm

Euclid's GCD algorithm (see slides of last week) computes the GCD of two integers.

The extended GCD algoritm extends the Euclidean algorithm, as follows. Instead of finding the GCD of two (positive) integers M and N it finds two integers x and y satisfying the so-called Bézout identity (or: Bézout equality):

$$xM + yN = \mathrm{gcd}(M, N).$$

For example, for arguments $M = 12$ and $N = 26$, the extended GCD algorithm gives the pair $x = -2$ and $y = 1$. And indeed, $-2 * 12 + 26 = 2$, which is the GCD of 12 and 26.

GCD Lemma

If $D \mid M$ and $D \mid N$ and $D = xM + yN$ (with $x, y \in \mathbb{Z}$), then $D = \gcd(M, N)$.

Proof:

1. From $D \mid M$ and $D \mid N$ we get that $D \leq \gcd(M, N)$.

2. From $\gcd(M, N) \mid xM$ and $\gcd(M, N) \mid yN$ it follows that

$$\gcd(M, N) \mid xM + yN,$$

i.e., $\gcd(M, N) \mid D$. Therefore, $\gcd(M, N) \leq D$.

Combining (1) and (2) we get $\gcd(M, N) = D$.

Importance of this:

If we can find x and y with the property that $xM + yN$ divides both M and N then we know, by the lemma, that we have found the GCD of M and N.

Imperative (iterative) version of the algorithm

Extended GCD algorithm

1. Let positive integers a and b be given.
2. $x := 0$;
3. $\text{lastx} := 1$;
4. $y := 1$;
5. $\text{lasty} := 0$;
6. while $b \neq 0$ do
   (a) $(q, r) := \text{quotRem}(a, b)$;
   (b) $(a, b) := (b, r)$;
   (c) $(x, \text{lastx}) := (\text{lastx} - q * x, x)$;
   (d) $(y, \text{lasty}) := (\text{lasty} - q * y, y)$.
7. Return $(\text{lastx}, \text{lasty})$.

Functional imperative version, in Haskell:

```
ext_gcd :: Integer -> Integer -> (Integer,Integer)
ext_gcd a b = let
   x = 0
   y = 1
   lastx = 1
   lasty = 0
  in ext_gcd' a b x y (lastx,lasty)

ext_gcd' = while5 (\ _ b _ _ _ ->  b /= 0)
                  (\ a b x y (lastx,lasty) -> let
                    (q,r)   = quotRem a b
                    (x',lastx') = (lastx-q*x,x)
                    (y',lasty') = (lasty-q*y,y)
                 in (b,r,x',y',(lastx',lasty')))
```

While5

This uses a `while5` loop:

```
while5 :: (a -> b -> c -> d -> e -> Bool)
       -> (a -> b -> c -> d -> e -> (a,b,c,d,e))
       -> a -> b -> c -> d -> e -> e
while5 p f x y z v w
  | p x y z v w = let
                    (x',y',z',v',w') = f x y z v w
                  in while5 p f x' y' z' v' w'
  | otherwise = w
```

Correctness of the Extended GCD Algorithm

Study Section 8.2 of The Haskell Road.

Mathematical Importance of Extended GCD Algorithm

Key to the Fundamental Theorem of Arithmetic:

> Every natural number greater than 1 has a unique prime factorization.

Practical Importance of Extended GCD Algorithm

Building block for the RSA algorithm for public key cryptography (topic of next week).

Bézout's identity

Bézout's identity is turned into an assertion, as follows:

```
bezout :: Integer -> Integer
          -> (Integer,Integer) -> Bool
bezout m n (x,y) = x*m + y*n == gcd m n
```

Use of this to produce assertive code for the extended algorithm:

```
ext_gcdA = assert2 bezout ext_gcd
```

Extended Euclidean Algorithm, Functional Version

A functional (recursive) version of the extended Euclidean algorithm:

```
fct_gcd :: Integer -> Integer -> (Integer,Integer)
fct_gcd a b =
  if b == 0
  then (1,0)
  else
     let
       (q,r) = quotRem a b
       (s,t) = fct_gcd b r
     in (t, s - q*t)
```

Testing for the GCD property

```
gcd_property :: Integer -> Integer
                -> (Integer,Integer) -> Bool
gcd_property = \ m n (x,y) -> let
    d = x*m + y*n
  in
    divides d m && divides d n
```

And use the property to define an assertive version of `fct_gcd`:

```
fct_gcdA = assert2 gcd_property fct_gcd
```

Assertion wrapper for functions with three arguments

```
assert3 :: (a -> b -> c -> d -> Bool) ->
           (a -> b -> c -> d) ->
            a -> b -> c -> d
assert3 p f x y z =
  if p x y z (f x y z) then f x y z
  else error "assert3"
```

Invariant wrapper for step functions with three arguments

```
invar3 :: (a -> b -> c -> Bool) ->
          (a -> b -> c -> (a,b,c)) ->
           a -> b -> c -> (a,b,c)
invar3 p f x y z =
  let
    (x',y',z') = f x y z
  in
   if p x y z && not (p x' y' z') then error "invar3"
   else (x',y',z')
```

Assertion wrapper for functions with four arguments

```
assert4 ::  (a -> b -> c -> d -> e -> Bool)
             -> (a -> b -> c -> d -> e)
             -> a -> b -> c -> d -> e
assert4 p f x y z u =
  if p x y z u (f x y z u) then f x y z u
  else error "assert4"
```

Invariant wrapper for step functions with four arguments

```
invar4 :: (a -> b -> c -> d -> Bool) ->
          (a -> b -> c -> d -> (a,b,c,d)) ->
           a -> b -> c -> d -> (a,b,c,d)
invar4 p f x y z u =
  let
    (x',y',z',u') = f x y z u
  in
   if p x y z u && not (p x' y' z' u')
   then error "invar4"
   else (x',y',z',u')
```

Assertion wrapper for functions with five arguments

```
assert5 ::  (a -> b -> c -> d -> e -> f -> Bool)
             -> (a -> b -> c -> d -> e -> f)
             -> a -> b -> c -> d -> e -> f
assert5 p f x y z u v =
  if p x y z u v (f x y z u v) then f x y z u v
  else error "assert5"
```

Invariant wrapper for step functions with five arguments

```
invar5 :: (a -> b -> c -> d -> e -> Bool) ->
          (a -> b -> c -> d -> e -> (a,b,c,d,e)) ->
           a -> b -> c -> d -> e -> (a,b,c,d,e)
invar5 p f x y z u v =
  let
    (x',y',z',u',v') = f x y z u v
  in
   if p x y z u v && not (p x' y' z' u' v')
   then error "invar5"
   else (x',y',z',u',v')
```

Assertive Code is Efficient Self-Documenting Code

More often than not, an assertive version of a function is much less efficient than the regular version: the assertions are inefficient specification algorithms to test the behaviour of efficient functions.

But this does not matter. To turn assertive code into self-documenting production code, all you have to do is load a module with alternative definitions of the assertion and invariant wrappers.

Take the definition of `assert1`. This is replaced by:

```
assert1 :: (a -> b -> Bool) -> (a -> b) -> a -> b
assert1 _ = id
```

And so on for the other wrappers. See module `AssertDoc` on the Course Website.

The assertions are still in the code, but instead of being executed they now serve as documentation. The assertive version of a function executes exactly as the version without the assertion. Assertive code comes with absolutely no efficiency penalty.

What Are We Testing?

Suppose a program (implemented function) fails its implemented assertion. What should we conclude? This is a pertinent question, for the assertion itself is a piece of code too, in the same programming language as the function that we want to test.

So what are we testing:

- the correctness of the code?
- the correctness of the implemememented specification for the code?

We Are Testing Both

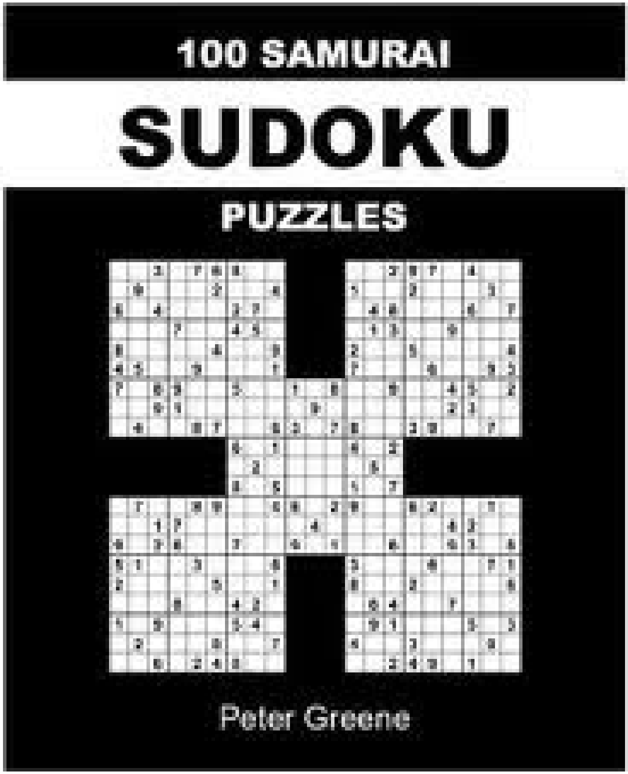
In fact, we are testing both at the same time. Therefore, the failure of a test can mean either of two things, and we should be careful to find out wat our situation is:

1. There is something wrong with the program.
2. There is something wrong with the specification of the assertion for the program.

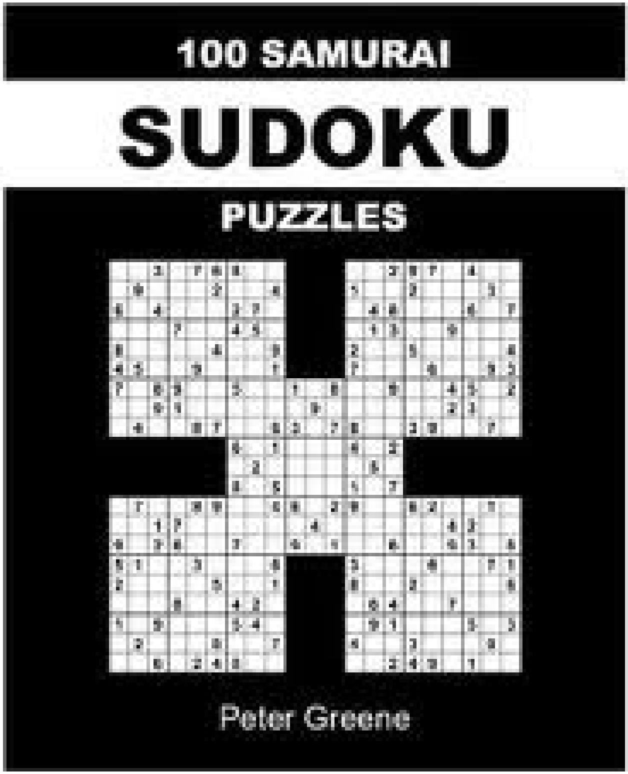
It is up to us to find out which case we are in.

In both cases it is important to find out where the problem resides. In the first case, we have to fix a code defect, and we are in a good position to do so because we have the specification as a yardstick. In the second case, we are not ready to fix code defects. First and foremost, we have to fix a defect in our understanding of what our program is supposed to do. Without that growth in understanding, it will be very hard indeed to detect and fix possible defects in the code itself.

Example: What is a Sudoku Problem?



Example: What is a Sudoku Problem?



- If declarative specification is to be taken seriously, all there is to solving sudokus is specifying what a sudoku problem is.

- If declarative specification is to be taken seriously, all there is to solving sudokus is specifying what a sudoku problem is.
- A sudoku is a 9×9 matrix of numbers in $\{1, \ldots, 9\}$ satisfying a number of constraints:

- If declarative specification is to be taken seriously, all there is to solving sudokus is specifying what a sudoku problem is.
- A sudoku is a 9×9 matrix of numbers in $\{1, \ldots, 9\}$ satisfying a number of constraints:
  - Every row should contain each number in $\{1, \ldots, 9\}$

- If declarative specification is to be taken seriously, all there is to solving sudokus is specifying what a sudoku problem is.
- A sudoku is a 9×9 matrix of numbers in $\{1, \ldots, 9\}$ satisfying a number of constraints:
  - Every row should contain each number in $\{1, \ldots, 9\}$
  - Every column should contain each number in $\{1, \ldots, 9\}$

- If declarative specification is to be taken seriously, all there is to solving sudokus is specifying what a sudoku problem is.

- A sudoku is a 9×9 matrix of numbers in $\{1, \ldots, 9\}$ satisfying a number of constraints:

  - Every row should contain each number in $\{1, \ldots, 9\}$
  - Every column should contain each number in $\{1, \ldots, 9\}$
  - Every subgrid $[i, j]$ with i, j ranging over $1..3$, $4..6$ and $7..9$ should contain each number in $\{1, \ldots, 9\}$.

- If declarative specification is to be taken seriously, all there is to solving sudokus is specifying what a sudoku problem is.
- A sudoku is a 9×9 matrix of numbers in $\{1, \ldots, 9\}$ satisfying a number of constraints:
  - Every row should contain each number in $\{1, \ldots, 9\}$
  - Every column should contain each number in $\{1, \ldots, 9\}$
  - Every subgrid $[i, j]$ with i, j ranging over $1..3$, $4..6$ and $7..9$ should contain each number in $\{1, \ldots, 9\}$.
- A sudoku problem is a partial sudoku matrix (a list of values in the matrix).

- If declarative specification is to be taken seriously, all there is to solving sudokus is specifying what a sudoku problem is.

- A sudoku is a 9×9 matrix of numbers in $\{1, \ldots, 9\}$ satisfying a number of constraints:

  – Every row should contain each number in $\{1, \ldots, 9\}$

  – Every column should contain each number in $\{1, \ldots, 9\}$

  – Every subgrid $[i, j]$ with i, j ranging over $1..3$, $4..6$ and $7..9$ should contain each number in $\{1, \ldots, 9\}$.

- A sudoku problem is a partial sudoku matrix (a list of values in the matrix).

- A solution to a sudoku problem is a complete extension of the problem, satisfying the sudoku constraints.

Example Problem, With Solution

```
+-------+-------+-------+    +-------+-------+-------+
| 5 3   |   7   |       |    | 5 3 4 | 6 7 8 | 9 1 2 |
| 6     | 1 9 5 |       |    | 6 7 2 | 1 9 5 | 3 4 8 |
|   9 8 |       |   6   |    | 1 9 8 | 3 4 2 | 5 6 7 |
+-------+-------+-------+    +-------+-------+-------+
| 8     |   6   |     3 |    | 8 5 9 | 7 6 1 | 4 2 3 |
| 4     | 8   3 |     1 |    | 4 2 6 | 8 5 3 | 7 9 1 |
| 7     |   2   |     6 |    | 7 1 3 | 9 2 4 | 8 5 6 |
+-------+-------+-------+    +-------+-------+-------+
|   6   |       | 2 8   |    | 9 6 1 | 5 3 7 | 2 8 4 |
|       | 4 1 9 |     5 |    | 2 8 7 | 4 1 9 | 6 3 5 |
|       |   8   |   7 9 |    | 3 4 5 | 2 8 6 | 1 7 9 |
+-------+-------+-------+    +-------+-------+-------+
```

Sudoku Constraints: Injectivity

Sudoku Constraints: Injectivity

- To express the sudoku constraints, we have to be able to express the property that a function is injective (or: one-to-one, or: an injection).

Sudoku Constraints: Injectivity

- To express the sudoku constraints, we have to be able to express the property that a function is injective (or: one-to-one, or: an injection).
- A function $f : X \to Y$ is an injection if it preserves distinctions: if $x_1 \neq x_2$ then $f(x_1) \neq f(x_2)$.

Sudoku Constraints: Injectivity

- To express the sudoku constraints, we have to be able to express the property that a function is injective (or: one-to-one, or: an injection).
- A function $f : X \to Y$ is an injection if it preserves distinctions: if $x_1 \neq x_2$ then $f(x_1) \neq f(x_2)$.
- Equivalently: a function $f : X \to Y$ is injective if $f(x_1) = f(x_2)$ implies that $x_1 = x_2$.

Sudoku Constraints as Injectivity Requirements

Sudoku Constraints as Injectivity Requirements

- Represent a sudoku as a function $f[i, j]$.

Sudoku Constraints as Injectivity Requirements

- Represent a sudoku as a function $f[i, j]$.
- Requirements:

Sudoku Constraints as Injectivity Requirements

- Represent a sudoku as a function $f[i,j]$.
- Requirements:
  - The members of each row should be all different.

    I.e., for every i, the function $j \mapsto f[i,j]$ should be injective (one to one).

    I.e., the list of values

    ```
    [f [i,j] | j <- [1..9] ]
    ```

    should not have duplicates.

Sudoku Constraints as Injectivity Requirements

- Represent a sudoku as a function $f[i, j]$.
- Requirements:
  - The members of each row should be all different.

    I.e., for every i, the function $j \mapsto f[i, j]$ should be injective (one to one).

    I.e., the list of values

    ```
    [f [i,j] | j <- [1..9] ]
    ```

    should not have duplicates.
  - The members of every column should be all different.

    I.e., for every j, the function $i \mapsto f[i, j]$ should be injective (one to one).

    I.e., the list of values

    ```
    [f [i,j] | i <- [1..9] ]
    ```

    should not have duplicates.

Sudoku Constraints as Injectivity Requirements

- Represent a sudoku as a function $f[i, j]$.
- Requirements:
  - The members of each row should be all different.

    I.e., for every i, the function $j \mapsto f[i, j]$ should be injective (one to one).

    I.e., the list of values

    ```
    [f [i,j] | j <- [1..9] ]
    ```

    should not have duplicates.
  - The members of every column should be all different.

    I.e., for every j, the function $i \mapsto f[i, j]$ should be injective (one to one).

    I.e., the list of values

    ```
    [f [i,j] | i <- [1..9] ]
    ```

    should not have duplicates.

– The members of every subgrid $[i, j]$ with i, j ranging over $1..3$, $4..6$ and $7..9$ should be all different.

I.e., the list of values

```
[f [i,j] | i <- [1..3], j <- [1..3] ]
```

should not have duplicates, and similarly for the other subgrids.

In Haskell . . .

```
type Row    = Int
type Column = Int
type Value  = Int
type Grid   = [[Value]]

positions, values :: [Int]
positions = [1..9]
values    = [1..9]

blocks :: [[Int]]
blocks = [[1..3],[4..6],[7..9]]
```

Showing Sudoku Stuff

Use 0 for a blank slot, so show 0 as a blank.

```
showDgt :: Value -> String
showDgt 0 = " "
showDgt d = show d
```

```
showRow :: [Value] -> IO()
showRow [a1,a2,a3,a4,a5,a6,a7,a8,a9] =
 do  putChar '|'         ; putChar ' '
     putStr (showDgt a1) ; putChar ' '
     putStr (showDgt a2) ; putChar ' '
     putStr (showDgt a3) ; putChar ' '
     putChar '|'         ; putChar ' '
     putStr (showDgt a4) ; putChar ' '
     putStr (showDgt a5) ; putChar ' '
     putStr (showDgt a6) ; putChar ' '
     putChar '|'         ; putChar ' '
     putStr (showDgt a7) ; putChar ' '
     putStr (showDgt a8) ; putChar ' '
     putStr (showDgt a9) ; putChar ' '
     putChar '|'         ; putChar '\n'
```

```
showGrid :: Grid -> IO()
showGrid [as,bs,cs,ds,es,fs,gs,hs,is] =
 do putStrLn ("+-------+-------+-------+")
    showRow as; showRow bs; showRow cs
    putStrLn ("+-------+-------+-------+")
    showRow ds; showRow es; showRow fs
    putStrLn ("+-------+-------+-------+")
    showRow gs; showRow hs; showRow is
    putStrLn ("+-------+-------+-------+")
```

Sudoku Type

Define a sudoku as a function from positions to values

```
type Sudoku = (Row,Column) -> Value
```

Useful conversions:

```
sud2grid :: Sudoku -> Grid
sud2grid s =
  [ [ s (r,c) | c <- [1..9] ] | r <- [1..9] ]

grid2sud :: Grid -> Sudoku
grid2sud gr = \ (r,c) -> pos gr (r,c)
  where
  pos :: [[a]] -> (Row,Column) -> a
  pos gr (r,c) = (gr !! (r-1)) !! (c-1)
```

Showing a sudoku

Show a sudoku by displaying its grid:

```
showSudoku :: Sudoku -> IO()
showSudoku = showGrid . sud2grid
```

Picking the block of a position

```
bl :: Int -> [Int]
bl x = concat $ filter (elem x) blocks
```

Picking the subgrid of a position in a sudoku.

```
subGrid :: Sudoku -> (Row,Column) -> [Value]
subGrid s (r,c) =
  [ s (r',c') | r' <- bl r, c' <- bl c ]
```

Free Values

Free values are available values at open slot positions.

```
freeInSeq :: [Value] -> [Value]
freeInSeq seq = values \\ seq

freeInRow :: Sudoku -> Row -> [Value]
freeInRow s r =
  freeInSeq [ s (r,i) | i <- positions  ]

freeInColumn :: Sudoku -> Column -> [Value]
freeInColumn s c =
  freeInSeq [ s (i,c) | i <- positions ]

freeInSubgrid :: Sudoku -> (Row,Column) -> [Value]
freeInSubgrid s (r,c) = freeInSeq (subGrid s (r,c))
```

The key notion

The available values at a position.

```
freeAtPos :: Sudoku -> (Row,Column) -> [Value]
freeAtPos s (r,c) =
  (freeInRow s r)
   `intersect` (freeInColumn s c)
   `intersect` (freeInSubgrid s (r,c))
```

Injectivity

A list of values is injective if each value occurs only once in the list:

```
injective :: Eq a => [a] -> Bool
injective xs = nub xs == xs
```

Injectivity Check for Rows, Columns, Blocks

Check (the non-zero values on) the rows, colums and subgrids for injectivity.

```
rowInjective :: Sudoku -> Row -> Bool
rowInjective s r = injective vs where
   vs = filter (/= 0) [ s (r,i) | i <- positions ]

colInjective :: Sudoku -> Column -> Bool
colInjective s c = injective vs where
   vs = filter (/= 0) [ s (i,c) | i <- positions ]

subgridInjective :: Sudoku -> (Row,Column) -> Bool
subgridInjective s (r,c) = injective vs where
   vs = filter (/= 0) (subGrid s (r,c))
```

Consistency Check

```
consistent :: Sudoku -> Bool
consistent s = and $
               [ rowInjective s r |  r <- positions ]
                ++
               [ colInjective s c |  c <- positions ]
                ++
               [ subgridInjective s (r,c) |
                    r <- [1,4,7], c <- [1,4,7]]
```

Sudoku Extension

Extend a sudoku by filling in a value in a new position

```
extend :: Sudoku -> (Row,Column,Value) -> Sudoku
extend s (r,c,v) (i,j) | (i,j) == (r,c) = v
                       | otherwise      = s (i,j)
```

The Solution Search Tree

A sudoku constraint is a list of possible values for a particular position.

```
type Constraint = (Row,Column,[Value])
```

Nodes in the search tree are pairs consisting of a sudoku and the list of all empty positions in it, together with possible values for those positions, according to the constraints imposed by the sudoku.

```
type Node = (Sudoku,[Constraint])

showNode :: Node -> IO()
showNode = showSudoku . fst
```

Solution

A sudoku is solved if there are no more empty slots.

```
solved  :: Node -> Bool
solved = null . snd
```

Successors in the Search Tree

The successors of a node are the nodes where the sudoku gets extended at the next empty slot position on the list, using the values listed in the constraint for that position.

```
extendNode :: Node -> Constraint -> [Node]
extendNode (s,constraints) (r,c,vs) =
   [(extend s (r,c,v),
     sortBy length3rd $
         prune (r,c,v) constraints) | v <- vs ]
```

`prune` removes the new value v from the relevant constraints, given that v now occupies position (r, c). The definition of `prune` is given below.

Put constraints that are easiest to solve first

```
length3rd :: (a,b,[c]) -> (a,b,[c]) -> Ordering
length3rd (_,_,zs) (_,_,zs') =
  compare (length zs) (length zs')
```

Pruning

Prune values that are no longer possible from constraint list, given a new guess (r, c, v) for the value of (r, c).

```
prune :: (Row,Column,Value)
      -> [Constraint] -> [Constraint]
prune _ [] = []
prune (r,c,v) ((x,y,zs):rest)
  | r == x = (x,y,zs\\[v]) : prune (r,c,v) rest
  | c == y = (x,y,zs\\[v]) : prune (r,c,v) rest
  | sameblock (r,c) (x,y) =
        (x,y,zs\\[v]) : prune (r,c,v) rest
  | otherwise = (x,y,zs) : prune (r,c,v) rest

sameblock :: (Row,Column) -> (Row,Column) -> Bool
sameblock (r,c) (x,y) = bl r == bl x && bl c == bl y
```

Initialisation

Success is indicated by return of a unit node `[n]`.

```
initNode :: Grid -> [Node]
initNode gr = let s = grid2sud gr in
              if (not . consistent) s then []
              else [(s, constraints s)]
```

The open positions of a sudoku are the positions with value 0.

```
openPositions :: Sudoku -> [(Row,Column)]
openPositions s = [ (r,c) | r <- positions,
                            c <- positions,
                            s (r,c) == 0 ]
```

Sudoku constraints, in a useful order

Put the constraints with the shortest lists of possible values first.

```
constraints :: Sudoku -> [Constraint]
constraints s = sortBy length3rd
    [(r,c, freeAtPos s (r,c)) |
                       (r,c) <- openPositions s ]
```

Depth First Search

The depth first search algorithm is completely standard. The goal property is used to end the search.

```
search :: (node -> [node])
       -> (node -> Bool) -> [node] -> [node]
search succ goal [] = []
search succ goal (x:xs)
  | goal x    = x : search succ goal xs
  | otherwise = search succ goal ((succ x) ++ xs)
```

Pursuing the Search

```
solveNs :: [Node] -> [Node]
solveNs = search succNode solved

succNode :: Node -> [Node]
succNode (s,[]) = []
succNode (s,p:ps) = extendNode (s,ps) p
```

Solving and showing the results

This uses some monad operators: `fmap` and `sequence`.

```
solveAndShow :: Grid -> IO[()]
solveAndShow gr = solveShowNs (initNode gr)

solveShowNs :: [Node] -> IO[()]
solveShowNs ns = sequence $ fmap showNode (solveNs ns)
```

Examples

```
example1 :: Grid
example1 = [[5,3,0,0,7,0,0,0,0],
            [6,0,0,1,9,5,0,0,0],
            [0,9,8,0,0,0,0,6,0],
            [8,0,0,0,6,0,0,0,3],
            [4,0,0,8,0,3,0,0,1],
            [7,0,0,0,2,0,0,0,6],
            [0,6,0,0,0,0,2,8,0],
            [0,0,0,4,1,9,0,0,5],
            [0,0,0,0,8,0,0,7,9]]
```

```
example2 :: Grid
example2 = [[0,3,0,0,7,0,0,0,0],
            [6,0,0,1,9,5,0,0,0],
            [0,9,8,0,0,0,0,6,0],
            [8,0,0,0,6,0,0,0,3],
            [4,0,0,8,0,3,0,0,1],
            [7,0,0,0,2,0,0,0,6],
            [0,6,0,0,0,0,2,8,0],
            [0,0,0,4,1,9,0,0,5],
            [0,0,0,0,8,0,0,7,9]]
```

```
example3 :: Grid
example3 = [[1,0,0,0,3,0,5,0,4],
            [0,0,0,0,0,0,0,0,3],
            [0,0,2,0,0,5,0,9,8],
            [0,0,9,0,0,0,0,3,0],
            [2,0,0,0,0,0,0,0,7],
            [8,0,3,0,9,1,0,6,0],
            [0,5,1,4,7,0,0,0,0],
            [0,0,0,3,0,0,0,0,0],
            [0,4,0,0,0,9,7,0,0]]
```

```
example4 :: Grid
example4 = [[1,2,3,4,5,6,7,8,9],
            [2,0,0,0,0,0,0,0,0],
            [3,0,0,0,0,0,0,0,0],
            [4,0,0,0,0,0,0,0,0],
            [5,0,0,0,0,0,0,0,0],
            [6,0,0,0,0,0,0,0,0],
            [7,0,0,0,0,0,0,0,0],
            [8,0,0,0,0,0,0,0,0],
            [9,0,0,0,0,0,0,0,0]]
```

```
example5 :: Grid
example5 = [[1,0,0,0,0,0,0,0,0],
            [0,2,0,0,0,0,0,0,0],
            [0,0,3,0,0,0,0,0,0],
            [0,0,0,4,0,0,0,0,0],
            [0,0,0,0,5,0,0,0,0],
            [0,0,0,0,0,6,0,0,0],
            [0,0,0,0,0,0,7,0,0],
            [0,0,0,0,0,0,0,8,0],
            [0,0,0,0,0,0,0,0,9]]
```

Next Week

More about algorithm specification, and writing assertive (self-testing) versions of algorithms.

Next week we will also demonstrate the importance of the extended GCD algorithm for public key cryptography.

You should study Section 8.2 of The Haskell Road, in order to understand the extended GCD algorithm.

References

[1] Kent Beck. Test Driven Development By Example. Addison-Wesley Longman, Boston, MA, 2002.

[2] Richard Bird. Pearls of Functional Algorithm Design. Cambridge University Press, 2010.

[3] C.A.R. Hoare. An axiomatic basis for computer programming. Communications of the ACM, 12(10):567–580, 583, 1969.

[4] F. Rabhi and G. Lapalme. Algorithms: a Functional Programming Approach. Addison-Wesley, 1999.