# Assertive Coding

Jan van Eijck
CWI & ILLC, Amsterdam

ESSLLI, Opole, August 8, 2012

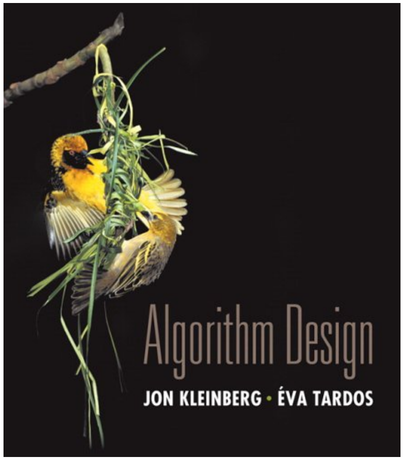
## Abstract

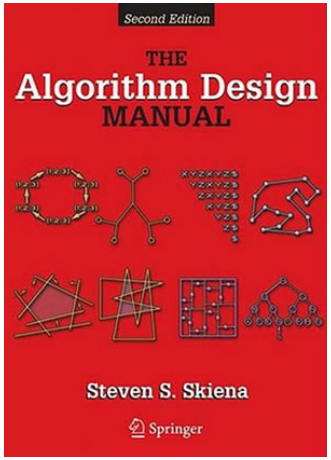
We show how to specify preconditions, postconditions, assertions and invariants, and how to wrap these around functional code or functional imperative code. We illustrate the use of this for writing programs for automated testing of code that is wrapped in appropriate assertions. We call this assertive 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 chapter 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.

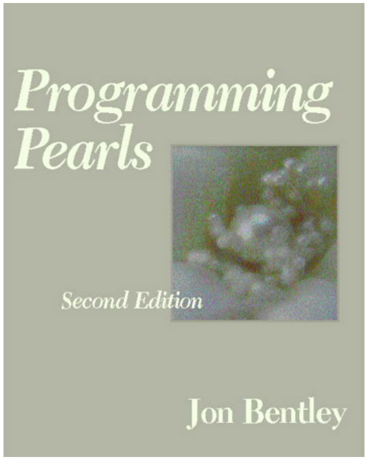
## Module Declaration

```
module AssertiveCoding

where

import Data.List
import FunctionalImperative hiding (g)
```

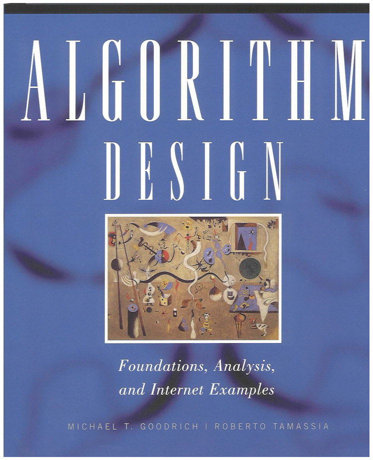
## Algorithm Design and Specification: Some excellent books

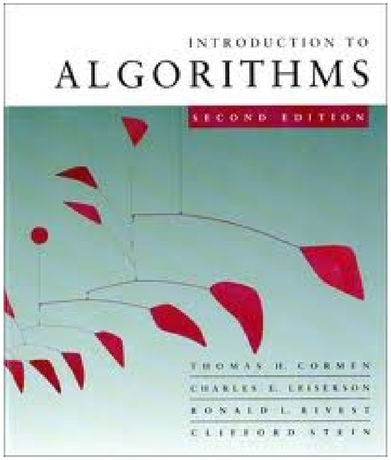


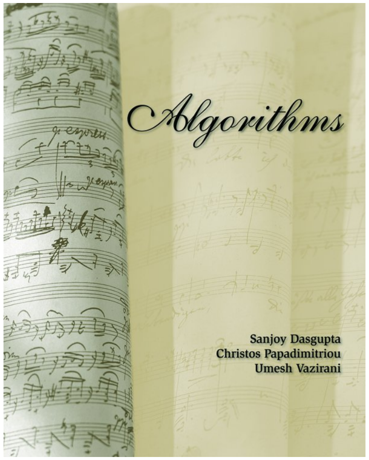




## And some more:

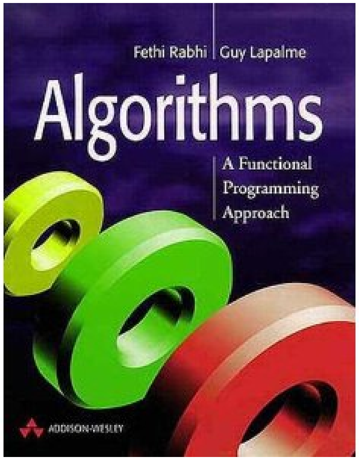

ALGORITHM DESIGN
Foundations, Analysis, and Internet Examples
MICHAEL T. GOODRICH | ROBERTO TAMASSIA

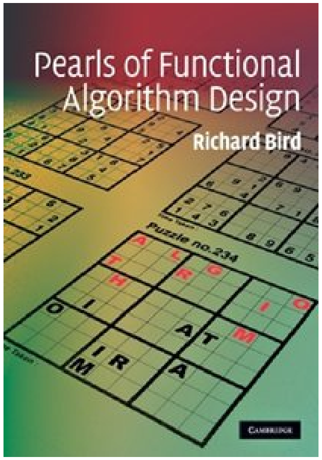


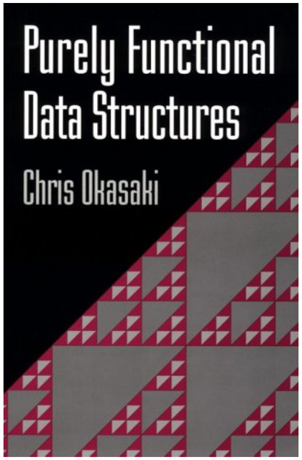


## Functional Imperative Algorithm Specification

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

This can be used to specify the expected behaviour of a function.

Consider the function $g$:

```
g = while1 even (`div` 2)
```

The $g$ function should always output an odd integer:

```
godd = post1 odd g
```

Note that `godd` has the same type as `g`, and that the two functions compute the same value whenever `godd` is defined (i.e., whenever the output value of `g` 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:

```
gA = assert1 (\ i o -> signum i == signum o) g
```

Note that `gA` has the same type as `g`. Indeed, as long as the assertion holds, `gA` and `g` 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:

$$\{\text{C}\}\ \text{Program}\ \{\text{C}\}$$

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 around $g$:

```
gsign = invar1 (>0) g
```

Another example:

```
gsign' = invar1 (<0) g
```

## Use of Invariant Inside While Loop

We can also use the invariant inside a while loop:

```
g3' =  while1 (\x -> even x)
              (invar1 (>0) (\x -> x `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]
```

## Assertive Versions of the GCD Algorithm

A precondition of the GCD algorithm is that its arguments are positive integers. This can be expressed as follows:

```
euclid = assert2 (\ m n _ -> m > 0 && n > 0)
                 euclidGCD
```

This function has the same type as `euclidGCD`. Both `euclid` and `euclidGCD` are partial functions; in fact they are the same partial function. The difference is that `euclid` aborts where `euclidGCD` diverges.

## Testing Euclid with Specific Input

If we want to check that `euclid` behaves correctly, we can test by using very specific input. E.g., if $p$ and $q$ are different prime numbers, we know that the GCD of $p$ and $q$ equals 1. Here is a test of Euclid's algorithm based on this knowledge.

```
testEuclid1 :: Int -> Bool
testEuclid1 k = let
    primes = take k (filter prime [2..])
  in
    and [ euclid p q == 1 |
          p <- primes, q <- primes, p /= q ]
```

## Prime

This uses the following implementation for the property of being a prime number:

```
prime :: Integer -> Bool
prime n =
  n > 1 && all (\ x -> rem n x /= 0) xs
  where xs = takeWhile (\ y -> y^2 <= n) [2..]
```

## Testing with Assertions

We can also test using assertions. An example assertion for Euclid's algorithm is that the result value is the greatest common divisor of the two inputs. To express that, we implement a bit of logic.

## Universal Quantification

The Haskell function `all` has type

`(a -> Bool) -> [a] -> Bool.`

Sometimes it is more convenient to have a universal quantifier of type

`[a] -> (a -> Bool) -> Bool.`

Here it is:

```
forall = flip all
```

## Divides

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
```

## GCD Definition

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)
```

## Assertive Version of Euclid’s GCD Function

```
euclid' :: Integer -> Integer -> Integer
euclid' = assert2 isGCD euclid
```

In case you don’t see what’s happening: the new function `euclid'` computes the same value as `euclid`, but the assertion guarantees that if it would ever compute the wrong value, an exception will be raised.

## A Test

```
testEuclid :: Integer -> Bool
testEuclid k =
  and [ (assert2 isGCD euclid) n m > 0 |
                       n <- [1..k], m <- [1..k] ]
```

The subtlety here is that the assertion allows us to define what is right and what is wrong. Imperative programs are dumb. Functional programs are dumb, too. But assertions allow us to relate what a program does to what the program is supposed to do.

The following test succeeds in under 25 seconds on my 3 Ghz dualcore machine:

```
*AssertiveCoding> testEuclid 300
True
```

## Finding An Invariant

Another thing we can do is find a suitable invariant. An invariant for Euclid's step function is that the set of common divisors does not change. The following function gives the common divisors of two integers:

```
divisors :: Integer -> Integer -> [Integer]
divisors m n = let
    k = min m n
  in [ d | d <- [2..k], divides d m, divides d n ]
```

## Assertion for the Step Function

We can use this in an assertion about the step function, as follows:

```
sameDivisors x y (x',y') =
    divisors x y == divisors x' y'
```

## Wrap the assertion around the step function

```
euclidGCD' :: Integer -> Integer -> Integer
euclidGCD' = while2
             (\ x y -> x /= y)
             (assert2 sameDivisors
             (\ x y -> if x > y
                        then (x-y,y)
                        else (x,y-x)))
```

Note that the assertion `sameDivisors` is in fact an invariant, stated as a relation between input and output of the step function.

## Invariant Wrapper for Step Functions with Two Arguments

```
invar2 :: (a -> b -> Bool) ->
          (a -> b -> (a,b)) ->
           a -> b -> (a,b)
invar2 p f x y =
  let
    (x’,y’) = f x y
  in
    if p x y && not (p x’ y’) then error "invar2"
    else (x’,y’)
```

## Example

As an example of how this is used, consider the following invariant. If $d$ divides both $x$ and $y$ before the step, then $d$ should divide $x$ and $y$ after the step.

Let's add a parameter for such a divisor $d$, and state the invariant:

```
euclidGCD'' :: Integer -> Integer
               -> Integer -> Integer
euclidGCD'' = \ d -> while2
             (\ x y -> x /= y)
             (invar2 (\ x y ->
                 divides d x && divides d y)
              (\ x y -> if x > y
                         then (x-y,y)
                         else (x,y-x)))
```

## Use of Example in Test

```
testEuclid2 :: Integer -> Bool
testEuclid2 k =
  and [ euclidGCD'' d n m >= 0 |
          n <- [1..k],
          m <- [1..k],
          d <- [2..min n m] ]
```

## The Extended GCD Algorithm

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.

Here is an 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
```

## 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 == euclid 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)
```

Again, an assertive version of this:

```
fct_gcdA = assert2 bezout 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')
```

## Asertion 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.

## Summary

State-of-the-art functional programming languages are well suited not only for implementing algorithms but also for specifying algorithms and wrap the specifications around the implementations, to produce assertive code. In fact you can do this with any language, but the cost is higher (sometimes much higher) for less abstract languages.

We have seen that imperative algorithms can be implemented directly in a functional language. The while loops that were employed above are explicit about the parameters that are used for checking loop termination and for defining the loop step. This often makes the implemented algorithms more perspicuous.

## 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.