

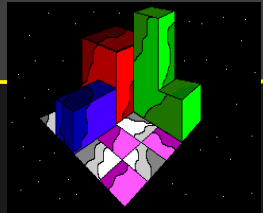
# Language Design meets Verifying Compilers

**David J. Pearce**

*ConsenSys*

@WhileyDave  
whileydave.com

```
class sprlib {  
public:  
    sprlib(char *,int = 0);  
    ~sprlib();  
    void setscreenptr(word);  
    void drawspr(int,int,int);  
    void xchgspr(int,int,int,int);  
    ...  
};
```



(Circa 1995)

# History

Friday, 24th, June

Checking a large routine. by Dr. A. Turing.

How can one check a routine in the sense of making sure that it is right?

In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily follows.

—Alan Turing, 1949

Stanford Verification Group  
Report No. 11

March 1979  
Edition 1

Computer Science Department  
Report No. STAN-CS-79-73 1

# STANFORD PASCAL VERIFIER USER MANUAL

by

STANFORD VERIFICATION GROUP

*“it was the **Stanford Pascal Verifier** project that produced the first verification system to target a real programming language”*

—Ireland'04

## The Verifying Compiler: A Grand Challenge for Computing Research

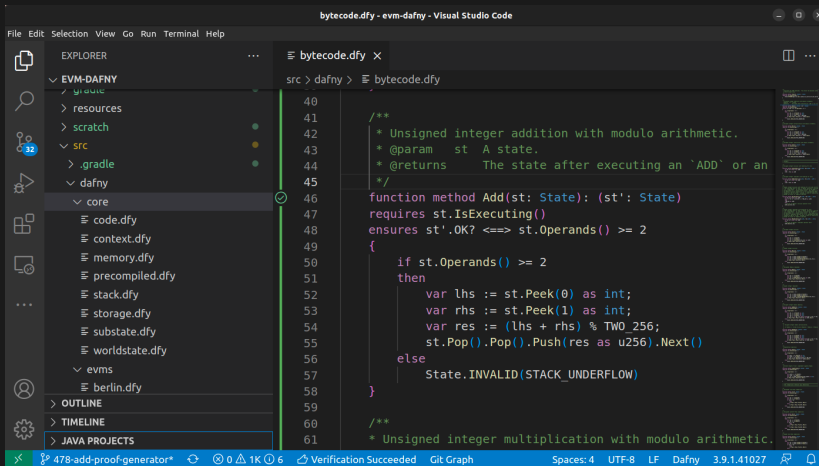
**Abstract.** This contribution proposes a set of criteria that distinguish a grand challenge in science or engineering from the many other kinds of short-term or long-term research problems that engage the interest of scientists and engineers. As an example drawn from Computer Science, it revives an old challenge: **the construction and application of a verifying compiler that guarantees correctness of a program before running it.**

—Hoare'03

```
deposit(...)
{
    while C1 {
        if C2 { return; }
        ...
    }
    // As the loop should always end prematurely with the 'return'
    // statement, this code should be unreachable. We assert 'false'
    // just to be safe.
    assert (false);
}
```

–Cassez, et al., FM'21

*(contract currently holds around 9million ETH)*





## Dafny

```
function abs(x:int) : (r:int)
  ensures r >= 0
  ensures (x == r) || (-x == r) {
    if x >= 0 then x else -x
  }
```

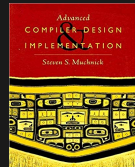
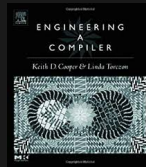
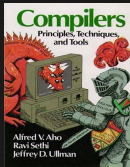
## Whiley

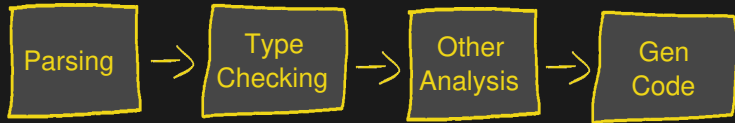
```
function abs(int x) -> (int r)
  ensures r >= 0
  ensures (r == x) || (r == -x):
    if x >= 0:
      return x
    else:
      return -x
```

# **(Verifying) Compilers**

“In computing, a **compiler** is a computer program that translates computer code written in one programming language (the source language) into another language (the target language).”

—Wikipedia









## Dafny

```
datatype Option = Some(val:int) | None

method unboxer(x:int, p:Option) returns (r:int)
requires x >= 0 ==> p.Some? {
  if x >= 0 {
    return p.val;
  } else {
    return x;
  }
}
```

## Whiley

```
type Some is {int val}
type Option is Some | null

function unboxer(int x, Option p) -> (int r)
requires (x >= 0) ==> (p is Some):
    if x >= 0:
        // Error!
        return p.val
    else:
        return x
```



## Whiley

```
type Some is {int val}
type Option is Some | null

function unboxer(int x, Option p) -> (int r)
requires (x >= 0) ==> (p is Some):
  if x >= 0:
    assert p is Some
    return p.val
  else:
    return x
```

## Dafny

```
method maxer(x:int, y:int) returns (r:int)
requires x != y
ensures (r == x) || (r == y) {
    if x > y {
        return x;
    } else if x < y {
        return y;
    }
}
```

Whiley

```
function maxer(int x, int y) -> (int r)
requires x != y:
  if x > y:
    return 1
  else if x < y:
    return 0
  else:
    fail
```

# Flow Typing

“In programming language theory, flow-sensitive typing (also called **flow typing** or occurrence typing) is a type system where the type of an expression depends on its position in the control flow.”

—Wikipedia

## Dafny

```
method iof(xs:seq<int>, x:nat) returns (r:Option)
  // If valid index returned, element matches item
  ensures r.Some? ==> (r.val<|xs| && xs[r.val] == x)
```

## Whiley

```
method iof(int[] xs, int x) -> (Option r)
  // If valid index returned, element matches item
  ensures (r is Some) ==> (xs[r.val] == x)
```

# Functional Purity

“To be functionally pure, a method must satisfy two critical properties: **First**, it must have no side effects. ... The **second** property is functional determinism.”

—Finifter *et al.*, 2008



## Dafny

```
function max(x:int, y:int) : (r:int)
ensures (r == x) || (r == y)
ensures (r >= x) && (r >= y) {
    ...
}
```

## Whiley

```
function max(int x, int y) -> (int r)
ensures (r == x) || (r == y)
ensures (r >= x) && (r >= y):
    ...
```

“Unlike pure functional programming, however, **mutable value semantics** allows part-wise in-place mutation, thereby eliminating the memory traffic usually associated with functional updates of immutable data”

—Racordon *et al.*, 2022

## Dafny

```
function fill(xs:seq<int>, n:nat, x:int) : seq<int>
requires n <= |xs|
{
    if n == 0 then xs
    else [x] + fill(xs[1..],n-1,x)
}
```

## Whiley

```
function fill(int[] xs, uint n, int x) -> (int[] rs)
requires n <= |xs|:
    for i in 0..n:
        xs[i] = x
    return xs
```

# **(Un)interpreted Functions**

“Normally function bodies are transparent and available for constructing proofs of assertions that use those functions. However, sometimes it is helpful to mark a function `{:opaque}` and treat it as an **uninterpreted function**, whose properties are just its specifications.”

—Dafny Reference Manual

“In mathematical logic, an **uninterpreted function** or function symbol is one that has no other property than its name and n-ary form.”

—Wikipedia

## Dafny

```
function zero_f(xs:seq<int>, n:nat) : (r:seq<int>)
requires n <= |xs| { ... }

method zero_m(xs:seq<int>,n:nat) returns(r:seq<int>)
requires n <= |xs| { ... }

-----

assert zero_f([1,2,3],2) == [0,0,3];

var r := zero_m([1,2,3],2);
assert r == [0,0,3];
```

```
property zero_p(int[] xs, uint n) -> (int[] rs)
requires n <= |xs|:
```

```
...
```

```
function zero_f(int[] xs, uint n) -> (int[] rs)
requires n <= |xs|:
```

```
...
```

```
method zero_m(int[] xs, uint n) -> (int[] rs)
requires n <= |xs|:
```

```
...
```

---

```
assert zero_p([1,2,3],2) == [0,0,3]
assert zero_f([1,2,3],2) == [0,0,3]
int[] rs = zero_m([1,2,3],2)
assert rs == [0,0,3]
```



# Inference

```
function contains(xs:seq<int>, x:int) : bool {
    exists k:nat | k < |xs| :: xs[k] == x
}

method find(xs:seq<int>, x:int) returns (r:nat)
requires contains(xs,x)
ensures xs[r] == x
{
    for i := 0 to |xs|
    invariant contains(xs[i..],x) {
        if xs[i] == x { return i; }
    }
    assert false;
}
```

```
function contains(xs:seq<int>, x:int) : bool {  
    exists k:nat | k < |xs| :: xs[k] == x  
}  
  
method find(xs:seq<int>, x:int) returns (r:nat)  
requires contains(xs,x)  
ensures r < |xs| && xs[r] == x  
{  
    for i := 0 to |xs|  
    invariant contains(xs[i..],x) {  
        if xs[i] == x { return i; }  
    }  
    assert false;  
}
```

## Dafny

```
method indexOf(xs: seq<int>, x:int) returns (r:nat)
ensures r <= |xs|
ensures (r < |xs|) ==> (xs[r] == x) {
    var i : int := 0;

    while i < |xs|
    invariant i <= |xs| {
        if xs[i] == x { break; }
        i := i + 1;
    }
    return i;
}
```

## Whiley

```
function indexOf(int[] xs, int x) -> (int r)
ensures (r != |xs|) ==> (xs[r] == x):
    int i = 0

    while i < |xs|
    where i >= 0 && i <= |xs|:
        if xs[i] == x:
            break
        i = i + 1
    return i
```



**ShriramKrishnamurthi**

@ShriramKMurthi

...

10/ The next generation of computing problems will not be about writing 80s style 5-line for-loops. It'll be about properties, specification, reasoning, verification, prompt eng, synthesis, etc. How will we get there?

And no, I will not be taking questions.

(-:

- **Checking a Large Routine**, Turing. In Report of a Conference on High Speed Automatic Calculating Machines, 1949.
- **A Practical Perspective on the Verifying Compiler Proposal**, Ireland, GCCRC, 2004.
- **Verifiable functional purity in Java**, Finifter *et al.*, CCS, 2008.
- **Implementation Strategies for Mutable Value Semantics**, Racordon *et al*, JOT, 2022. .

<http://whiley.org>

@WhileyDave  
<http://github.com/Whiley>



**Language Features?**

Whiley

```
function copy(int[] xs, uint n) -> (int[] ys)
ensures |ys| == n:
    // Create array of given size
    ys = [0; n]
    // Copy over what we can
    for i in 0..min(n, |xs|)
    where n == |ys|:
        ys[i] = xs[i]
    // Done
    return ys
```

Whiley

```
type Box<T> is &T
```

```
method destroy(Box<T> p):
```

```
    //
```

```
    delete p
```

Whiley

```
type Box<T> is &T

method destroy(Box<T> p)
  requires #p == 1:
    //
    delete p
```

Whiley

```
type Box<T> is &T where #p == 1
```

```
method destroy(Box<T> p):
```

```
    //
```

```
    delete p
```