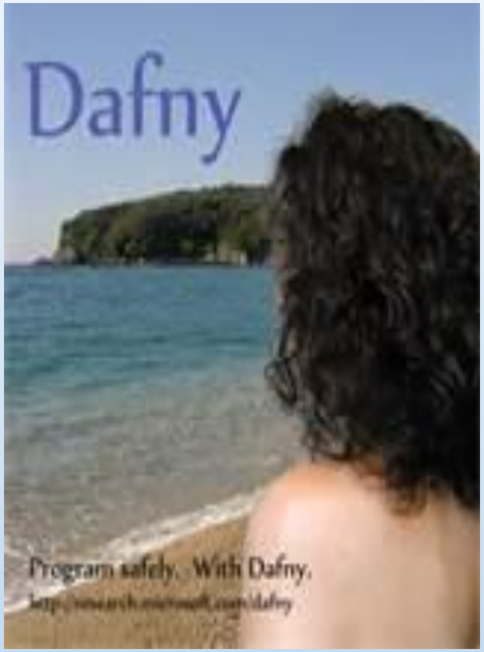


Dafny support

for

Program Reasoning

at Imperial College London

funded by MSR Connections

The Program Reasoning Course

- Program Reasoning is taught in 2nd term of 1st year.
- In 1st term, 1st Yr students are taught
  - Programing in Haskell and Java,
  - First order logic and box proofs (no induction, …)
  - Architectures, continuous maths, ….
- In 2nd term, 1st Yr students are taught
  - Reasoning about Haskell programs (ie induction)
  - Reasoning about Java programs (i.e “disguised” Hoare Logic),
  - Operating systems, , ….

Dafny material for the Program Reasoning Course

- Nov-Dec 2012: developed Dafny lab for Reasoning Course
- Jan-March 2013: ran lab (very voluntary) for Reasoning Course
- April-May 2013: refined material
- Jan-March 2014: run lab again, this time better integrated into course

Why use Dafny?

- Students like using tools.
- Tools make the area feel more relevant; in 2012/13, several students asked whether they would be using Dafny in their careers.
- Immediate feedback and soundness of the tool – i.e. great when proof has been validated by tool.
- Dafny integrates the programming and proving.
- Dafny allows proof checking, but does not insist on proof scripts.

Why *not* use Dafny?

- It sometimes does too much:
  it is a theorem prover, not a proof checker.

- It sometimes does too little:
  as all theorem provers, it sometimes
  gets stuck at awkward, unpredictable places

Aims of the Reasoning Course

1. Write specs for functional code.
2. Detailed, precise proofs using induction (various flavours); explicit proof steps, justification, and precision in application of lemmas.
3. “Architecture” of the proof – discover auxiliary lemmas.
4. Specs for imperative code.
5. “Architecture” of the proof: invariants, assertions, specs of auxiliary code.
6. Detailed Hoare-logic proofs, using, essentially, the SSA equivalent.
7. Notation (use, develop new).
8. No objects, no framing *– perhaps this should change.*
9. No concurrency.

Dafny Support for Aims of Course

1. Write specs for functional code -- yes
2. Detailed, precise proofs using induction (various); explicit proof steps, justification, and precision in application of lemmas. -- partly
3. "Architecture" of the proof – discover auxiliary lemmas – yes
4. Specs for imperative code -- yes
5. "Architecture" of the proof: invariants, assertions, specs of auxiliary code. -- yes
6. Detailed Hoare-logic proofs, using, essentially the SSA equivalent -- partly
7. Notation (use, develop new). -- partly

The remainder of this talk outlines the material we developed, and discusses in how far it supports the aims of the course.

All material available on
http://www.doc.ic.ac.uk/~scd/Dafny_Material

And there is a Verification Corner episode.

Material we Developed

1. From Haskell to Dafny (functional flavour)
2. Specifications, Lemmas, and Proofs.
3. Incremental Proof Development (auxiliary Lemmas)
4. From Java to Dafny (imperative flavour)
5. Verification of Imperative code
6. Incremental Program Development

Disclaimer:
Dafny MSR site offers Dafny tutorial and very extensive suite of exercises. And several VC episodes.

What we offer is a systematic teaching for students who are less familiar with how to write proofs, and verify programs.

1. From Haskell to Dafny

- Syntactic similarities and differences,
- Termination and totality through preconditions,
- Datatypes,
- Pattern matching.

1. From Haskell to Dafny – syntactic differences

Haskell

askell, we declare the
e of a function separately
n its definition.

“::” means “has type”

In Haskell, primitive/built-in
start with an uppercase lette
e.g. `Int`, `Bool`.

```
fib :: Int -> Int
fib n = if n <= 1
        then n
        else fib (n-1) + fib (n-2)
```

kell, we write the return
of a function after an “=”.

1. From Haskell to Dafny - syntactic differences

Haskell

laskell, we declare the
e of a function separately
n its definition.

“::” means “has type”

In Haskell, primitive/built-in t
start with an uppercase letter
e.g. Int, Bool.

```
fib :: Int -> Int
fib n = if n <= 1
        then n
        else fib (n-1) + fib (n-2)
```

skell, we write t
of a function af

Dafny

In Dafny, primitive/built-in types start with an lowercase letter, e.g. int, bool.

In Dafny, we give the type of function arguments and the return type of the function along with its definition.

“:” means “has type”

In Dafny, we write the return value of a function between braces.

```
function fib(n: int): int
{
  if n <= 1
  then n
  else fib(n-1) + fib(n-2)
}
```

fib(n: int): int means fib takes a single int argument and returns an int.

2. Specifications, Lemmas, Proofs

Stating the lemma

$$\forall x, y : \mathbb{N}. Succ(x + y) = x + Succ(y)$$

through a ghost method declaration:

```
ghost method prop_add_Succ(x: natr, y: natr)
  ensures Succ(add(x, y)) == add(x, Succ(y));
```

2. Specifications, Lemmas, Proofs

Proving the lemma

$$\forall x, y : \mathbb{N}. Succ(x + y) = x + Succ(y)$$

through a ghost method with (empty) body

```
ghost method prop_add_Succ(x: natr, y: natr)
  ensures Succ(add(x, y)) == add(x, Succ(y));
{ }
```

2. Specifications, Lemmas, Proofs

Proving the lemma

$$\forall x, y : \mathbb{N}. Succ(x + y) = x + Succ(y)$$

through a ghost method with (empty) body

```
ghost method prop_add_Succ(x: natr, y: natr)
  ensures Succ(add(x, y)) == add(x, Succ(y));
{ }
```

Notice the automatic proof!

2. Specifications, Lemmas, Proofs

From a “handwritten” proof, to a Dafny, verified, proof.

We want to show:

$$\forall x, y : \mathbb{N}. x + y = y + x$$

“handwritten” proof

Base case:

To Show: $\forall y : natr.\, add(Zero, y) = add(y, Zero)$

Take an arbitrary $y : natr$

$add(Zero, y)$

$= y$ (by definition of add)

$= add(y, Zero)$ (by prop_add_Zero)

Inductive step:

Take an arbitrary $x : natr$

Inductive Hypothesis: $\forall y : natr.\, add(x, y) = add(y, x)$

To Show: $\forall y : natr.\, add(suc(x), y) = add(y, suc(x))$

Take an arbitrary y: $natr$

$add(succ(x), y)$

$= Succ(add(x, y))$ (by definition of add)

$= Succ(add(y, x))$ (by IH)

$= add(y, Succ(x))$ (by prop_add_Succ)

“handwritten” to Dafny -- base case

Base case:

To Show: $\forall y : natr.\, add(Zero, y) = add(y, Zero)$

Take an arbitrary $y : natr$

$add(Zero, y)$

$= y$ (by definition of add)

$= add(y, Zero)$ (by prop_add_Zero)

Inductive step:

“handwritten” to Dafny – base case

Base case:

To Show: $\forall y : natr.\, add(Zero, y) = add(y, Zero)$

Take an arbitrary $y : natr$

$add(Zero, y)$

$= y$ (by definition of add)

$= add(y, Zero)$ (by prop_add_Zero)

Inductive step:

```
ghost method prop_add_comm(x: natr, y: natr)
  ensures add(x, y) == add(y, x);
{
  match x {
    case Zero =>
      calc {
        add(Zero, y);
                   ==  // definition of add
         y;
                   == { prop_add_Zero(y); }
         add(y, Zero);
      }
```

“handwritten” to Dafny – ind.step

Inductive step:
Take an arbitrary $x : natr$
Inductive Hypothesis: $\forall y : natr.\, add(x, y) = add(y, x)$
To Show: $\forall y : natr.\, add(suc(x), y) = add(y, suc(x))$

Take an arbitrary $y : natr$
$add(succ(x), y)$
$= Succ\big(add(x, y)\big)$ (by definition of add)
$= Succ\big(add(y, x)\big)$ (by IH)
$= add\big(y, Succ(x)\big)$ (by prop_add_Succ)

“handwritten” to Dafny – ind.step

Inductive step:
Take an arbitrary $x : natr$
Inductive Hypothesis: $\forall y : natr.\, add(x,y) = add(y,x)$
To Show: $\forall y : natr.\, add(suc(x),y) = add(y,suc(x))$

Take an arbitrary y: $natr$
$add(succ(x),y)$
$= Succ\big(add(x,y)\big)$ (by definition of add)
$= Succ\big(add(y,x)\big)$ (by IH)
$= add\big(y, Succ(x)\big)$ (by prop_add_Succ)

```
case Succ(x') =>
      calc {
         add(x, y);
      == { assert x == Succ(x'); }
         add(Succ(x'), y);
      == // definition of add
         Succ(add(x', y));
      == { prop_add_comm(x', y); }
          // Induction Hypothesis
         Succ(add(y, x'));
      == {  prop_add_Succ(y, x');  }
         add(y, Succ(x'));
      }
} }
```

The Proof more succinctly

```
ghost method prop_add_comm(x: natr, y: natr)
  ensures add(x, y) == add(y, x);
{
  match x {
    case Zero =>
       calc {
           add(Zero, y)
        == { prop_add_Zero(y); }
           add(y, Zero);
     }
    case Succ(x') =>
      calc {
           add(x, y);
        == {  prop_add_Succ(y, x');  }
           add(y, Succ(x'));
      }
  }
}
```

The Proof *even* more succinctly

```
ghost method prop_add_comm(x: natr, y: natr)
  ensures add(x, y) == add(y, x);
{
  match x {
  case Zero =>
    prop_add_Zero(y);
  case Succ(x') =>
    prop_add_Succ(y, x');
  }
}
```

Dafny as an oracle

Does the following property hold?

$$\forall c\colon int,\ cs\colon list\langle int\rangle,\ n\colon nat,\ chg\colon list\langle int\rangle.$$
$$elem(c, cs) \wedge correct\_change(cs, n, chg) \Longrightarrow elem(c, chg)$$

Dafny as an oracle

Does the following property hold?

$$\forall c\colon int,\ cs\colon list\langle int\rangle,\ n\colon nat,\ chg\colon list\langle int\rangle.$$
$$elem(c, cs) \wedge correct\_change(cs,n,chg) \Longrightarrow elem(c, chg)$$

```
ghost method partA(c:nat,cs:list<int>,chg: list<int> )
    requires elem(c,cs) && correct_change(cs, c, chg);
    ensures elem(c,chg);
  { }
```

Dafny as an oracle

Does the following property hold? **NO!**

$$\forall c\text{: } int\text{, } cs\text{: } list\langle int\rangle\text{, } n\text{: } nat\text{, } chg\text{:} list\langle int\rangle.$$
$$elem(c, cs) \wedge correct\_change(cs,n,chg) \Longrightarrow elem(c, chg)$$

```
ghost method partA(c:nat,cs:list<int>,chg: list<int> )
    requires elem(c,cs) && correct_change(cs, c, chg);
    ensures elem(c,chg);
  { }
```

		Description
⊗	1	A postcondition might not hold on this return path.
	2	This is the postcondition that might not hold.

Dafny for counterexamples

Give a counterexample for

$$\forall c\colon int,\ cs\colon list\langle int\rangle,\ n\colon nat,\ chg\colon list\langle int\rangle.$$
$$elem(c,\ cs) \land correct\_change(cs, n, chg) \Longrightarrow elem(c,\ chg)$$

Dafny for counterexamples

Give a counterexample for

$$\forall c\colon int,\ cs\colon list\langle int\rangle,\ n\colon nat,\ chg\colon list\langle int\rangle.$$
$$elem(c,\ cs) \land correct\_change(cs,n,chg) \Longrightarrow elem(c,\ chg)$$

```
ghost method partACountExample( )
{
      var coins : list<int> := Cons(1,Cons(6,Cons(5,Nil)));
      var change : list<int> := Cons(5,Nil);
      assert ! (elem(6, coins) && correct_change(coins, 5,change) && elem(6,change));
}
```

Dafny for counterexamples

Give a counterexample for

$$\forall c\!: int,\ cs\!: list\langle int\rangle,\ n\!: nat,\ chg\!: list\langle int\rangle.$$
$$elem(c,\ cs) \land correct\_change(cs,n,chg) \Longrightarrow elem(c,\ chg)$$

```
ghost method partACountExample( )
{
      var coins : list<int> := Cons(1,Cons(6,Cons(5,Nil)));
      var change : list<int> := Cons(5,Nil);
      assert ! (elem(6, coins) && correct_change(coins, 5,change) && elem(6,change));
}
```

```
Dafny program verifier finished with 11 verified, 0 errors
```

samples about Dafny - A language and program verifier for functional correctness

More such exercises

- On numbers, oodd and even
- On lists, concatenation, length, reverse
- On making change out of coins

3. Incremental Proof Development

- Proof development top-down.
- Discover, express and use auxiliary lemmas.
- Use `assume` statements to assume a property, and delay its proof.

Proving quicksort

```
ghost method prop_qsort(xs: list<int>)
    ensures is_sorted(qsort(xs))  && perm(xs, qsort(xs));
    decreases len(xs);
{
  match xs{
   ...
  case Cons(y, ys) =>
       var le := take_le(y, ys);   var gt := take_gt(y, ys);
       var sle := qsort(le);        var sgt := qsort(gt);
       var res := app(sle, Cons(y, sgt)) ;
       assume is_sorted(res) && perm( xs, res);
  }
}
```

Proving quicksort incrementally

```
ghost method prop_qsort(xs: list<int>)
    ensures is_sorted(qsort(xs))  && perm(xs, qsort(xs));
    decreases len(xs);
{
   match xs{
    …
   case Cons(y, ys) =>
        var le := take_le(y, ys);   var gt := take_gt(y, ys);
        var sle := qsort(le);        var sgt := qsort(gt);
        var res := app(sle, Cons(y, sgt)) ;
        assume is_sorted(res) && perm( xs, res);
   }
}
```

Proof Structure (liberal notation)

1. le <= y   &&   gt > y
2. perm(ys,le++gt)
3. sorted(sle) && perm(sle,le)

   sorted(sgt) && perm(sgt,gt)
4. sle <= y && sgt > y
5. sorted(sle++y++sgt)
6. perm(xs,sle++y++xs)

Proving quicksort incrementally

```
ghost method prop_qsort(xs: list<int>)
     ensures is_sorted(qsort(xs))  && perm(xs, qsort(xs));
     decreases len(xs);
{
   match xs{
    …
   case Cons(y, ys) =>
        var le := take_le(y, ys);   var gt := take_gt(y, ys);
        var sle := qsort(le);        var sgt := qsort(gt);
        var res := app(sle, Cons(y, sgt)) ;
        assume is_sorted(res) && perm( xs, res);
   }
}
```

Proof Structure (liberal notation)

1. le <= y && gt > y
2. perm(ys,le++gt)
3. sorted(sle) && perm(sle,le)

   sorted(sgt) && perm(sgt,gt)
4. sle <= y && sgt > y
5. sorted(sle++y++sgt)
6. perm(xs,sle++y++xs)

In general, for each "proof target", we apply the following steps

1) Assume the target,
2) Formulate a respective lemma, and assert the target,
3) Prove the respective lemma

Proving quicksort incrementally

```
ghost method prop_qsort(xs: list<int>)
      ensures is_sorted(qsort(xs))  && perm(xs, qsort(xs));
      decreases len(xs);
{
   match xs{
    …
   case Cons(y, ys) =>
         var le := take_le(y, ys);   var gt := take_gt(y, ys);
         assume perm(ys, app(le,gt);
         var sle := qsort(le);         var sgt := qsort(gt);
         var res := app(sle, Cons(y, sgt)) ;
         ….
   }
}
```

Proof Structure (liberal notation)

1. le <= y && gt > y
2. **perm(ys,le++gt)**
3. sorted(sle) && perm(sle,le)

   sorted(sgt) && perm(sgt,gt)
4. sle <= y && sgt > y
5. sorted(sle++y++sgt)
6. perm(xs,sle++y++xs)

In general, for each "proof target", we apply the following steps

1) Assume the target,
2) Formulate a respective lemma, and assert the target,
3) Prove the respective lemma

Proving quicksort incrementally

```
ghost method prop_qsort(xs: list<int>)
    ensures is_sorted(qsort(xs))  && perm(xs, qsort(xs));
    decreases len(xs);
{
  match xs{
   …
  case Cons(y, ys) =>
      var le := take_le(y, ys);   var gt := take_gt(y, ys);
      prop_take_perm(y,ys,le,gt);
      assert perm(ys, app(le,gt);
      var sle := qsort(le);         var sgt := qsort(gt);
      var res := app(sle, Cons(y, sgt)) ;
      assume is_sorted(res) && perm( xs, res);
  }
}
```

Proof Structure (liberal notation)

1. le <= y   &&   gt > y
2. **perm(ys,le++gt)**
3. sorted(sle) && perm(sle,le)

   sorted(sgt) && perm(sgt,gt)
4. sle <= y && sgt > y
5. sorted(sle++y++sgt)
6. perm(xs,sle++y++xs)

In general, for each “proof target”, we apply the following steps

1) Assume the target,
2) Formulate a respective lemma, and assert the target,
3) Prove the respective lemma

```
ghost method prop_take_perm(n: int, xs: list<int>, le: list<int>, gt: list<int>)
    ensures perm( xs, app(take_le(n,xs), take_gt(n,xs)) ) ;
```

Proving quicksort incrementally

```
ghost method prop_qsort(xs: list<int>)
     ensures is_sorted(qsort(xs))  && perm(xs, qsort(xs));
     decreases len(xs);
{
   match xs{
    …
   case Cons(y, ys) =>
         var le := take_le(y, ys);   var gt := take_gt(y, ys);
         prop_take_perm(y,ys,le,gt);
         assert perm(ys, app(le,gt);
         var sle := qsort(le);         var sgt := qsort(gt);
         var res := app(sle, Cons(y, sgt)) ;
         assume is_sorted(res) && perm( xs, res);
   }
}
```

Proof Structure (liberal notation)

1. le <= y && gt > y
2. **perm(ys,le++gt)**
3. sorted(sle) && perm(sle,le)

   sorted(sgt) && perm(sgt,gt)
4. sle <= y && sgt > y
5. sorted(sle++y++sgt)
6. perm(xs,sle++y++xs)

In general, for each “proof target”, we apply the following steps

1) Assume the target,
2) Formulate a respective lemma, and assert the target,
3) Prove the respective lemma

```
ghost method prop_take_perm(n: int, xs: list<int>, le: list<int>, gt: list<int>)
     ensures perm( xs, app(take_le(n,xs), take_gt(n,xs)) )
{ … }
```

More such exercises

- On flatttening and reconstructing trees (VC episode)
- Several tailrecursive functions

4. From Java to Dafny

•

4. From Java to Dafny

Java

```
int Find<T>(T x, T[] a)
{
  int i = 0;
  while (i < a.length && a[i] != x)
  {
    i++;
  }
  return i;
}
```

4. From Java to Dafny

Java

```
int Find<T>(T x, T[] a)
{
  int i = 0;
  while (i < a.length && a[i] != x)
  {
    i++;
  }
  return i;
}
```

Dafny

```
method Find<T(==)>(x: T, a: array<T>)
   returns (r: int)
  requires a != null;
{
  var i := 0;
  while (i < a.Length
           && a[i] != x)
  { i := i + 1; }
  return i;
}
```

Loop invariants

```
method Find<T(==)>(a: array<T>, x: T)
  returns (r : int)
  requires a != null;
  ensures 0 <= r <= a.Length;
  ensures r < a.Length ==> a[r] == x;
  ensures forall j :: 0 <= j < r ==> a[j] != x;
{
  var i := 0;
  while (i < a.Length && a[i] != x)
    invariant i <= a.Length;
    invariant forall j :: 0 <= j < i ==> a[j] != x;
  {
    i := i + 1;
  }
  return i;
}
```

More such exercises

- Single loop: gcd, and sum through incr/decr
- Nested loops: arithm. opers through incr/decr
- Triple nested loops
- Consecutive loops
- Maximum in array, product, insertion sort
- McCarthy91

5. Incremental Development of imperative programs

- Similar to incremental proof development

Incremental code development

a. Find loop INVariant so that
   INV && TERM → POST
b. Incomplete loop body, but assume INV
c. Write and call an empty auxiliary method which preserves INV
d. Develop body of auxiliary method
e. Inline auxiliary method in the loop

Bubble-sort incremental

```
method bubbleSort(a: array<int>)
  requires a != null;
  modifies a;
  ensures perm(a[..],old(a[..])) && sorted(a);
{




    . . .




}
```

a. INV && TERM → POST

```
method bubbleSort(a: array<int>)
  requires a != null;
  modifies a;
  ensures perm(a[..],old(a[..])) && sorted(a);
{
  var i: nat := 0;
  while i < a.Length
    invariant 0 <= i <= a.Length;
    invariant sortedBetween(a, 0, i) && perm(a[..],old(a[..]));
    decreases a.Length - i;
  {
        …
        …
  }
}
```

b. Incomplete loop body assume INV

```
method bubbleSort(a: array<int>)
  requires a != null;
  modifies a;
  ensures perm(a[..],old(a[..])) && sorted(a);
{
  var i: nat := 0;
  while i < a.Length
    invariant 0 <= i <= a.Length;
    invariant sortedBetween(a, 0, i) && perm(a[..],old(a[..]));
    decreases a.Length - i;
  {
        i:= i+1;
        assume sortedBetween(a, 0, i) && perm(a[..],old(a[..]));
   }
}
```

c. Write and call empty aux method which preserves INV

```
method bubbleSort(a: array<int>)
  …
{
  var i: nat := 0;
  while i < a.Length
    invariant 0 <= i <= a.Length;
    invariant sortedBetween(a, 0, i) && perm(a[..],old(a[..]));
    decreases a.Length - i;
  {
       pushToRight(a, i);
       i:= i+1;
   }
}

method pushToRight(a: array<int>, i: nat)
   requires a!=null && 0<i<=a.Length && sortedBetween(a,0,i-1);
   modifies a;
   ensures sortedBetween(a,0,i) && perm(old(a[..]),a[..]);
```

d. Develop body for aux method

```
method pushToRight(a: array<int>, i: nat)
   requires a!=null && 0<i<=a.Length && sortedBetween(a,0,i-1);
   modifies a;
   ensures sortedBetween(a,0,i) && perm(old(a[..]),a[..]);
{
  var j: nat := i;

  while j > 0 && a[j - 1] > a[j]
    invariant 0 <= j <= i;
    invariant perm(old(a[..]),a[..]);
    invariant sortedBetween(a,0,j) && sortedBetween(a,j,i+1);
    invariant forall k, k' :: 0 <= k < j && j + 1 <= k' < i + 1
                                             ==> a[k] <= a[k'];
  {
    swap(a,j-1,j);
    j := j - 1;
  }
}
```

e. Inline aux method in loop body

```
method pushToRight(a: array<int>, i: nat)
   requires a!=null && 0<i<=a.Length && sortedBetween(a,0,i-1);
   modifies a;
   ensures sortedBetween(a,0,i) && perm(old(a[..]),a[..]);
{
  var j: nat := i;

  while j > 0 && a[j - 1] > a[j]
    invariant 0 <= j <= I && perm(old(a[..]),a[..]);
    invariant sortedBetween(a,0,j) && sortedBetween(a,j,i+1);
    invariant forall k, k' :: 0 <= k < j && j + 1 <= k' < i + 1
                                           ==> a[k] <= a[k'];
    {   ghost var a_before := a[..];
        var temp: int := a[j - 1];  a[j - 1] := a[j];  a[j] := temp;
        ghost var a_after := a[..];
        swap_implies_perm(a_before,a_after,j-1,j);
        j := j-1;
    }
   i := i + 1;
 }
}
```

More such exercises

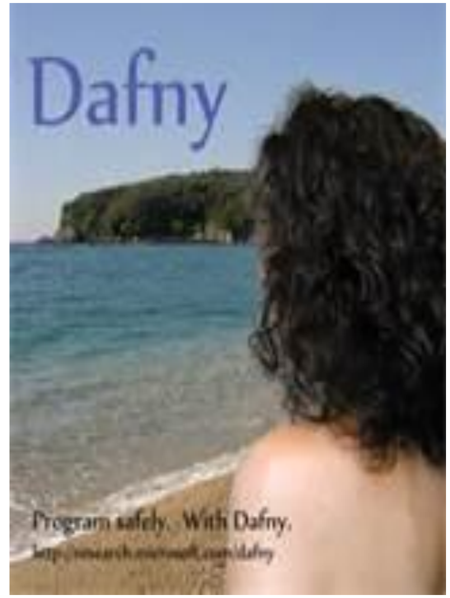
- insertionsort

Summary of material

1. From Haskell to Dafny (functional flavour)
2. Specifications, Lemmas, and Proofs.
3. Incremental Proof Development (auxiliary Lemmas)
4. From Java to Dafny (imperative flavour)
5. Verification of Imperative code
6. Incremental Program Development

Overall motto: Declarative proofs, handwritten proof first

Reasoning course using



- Nice syntax,
- Beautiful interface,
- Supports incremental proof/code development,
- Supports both declarative and imperative proofs,
- Biggest worry: will it confuse students as to nature of a proof?
- Evolves fast.
- Looking forward to applying it.

Thank you to Judith & Rustan