Verified programming in Dafny

Will Sonnex and Sophia Drossopoulou

An introductory course to the use of Dafny for writing programs with fully verified specifications.

Acknowledgements: The development of this material has been supported by a gift from Microsoft Research, and a visit to Microsoft Research, organized and hosted by Judith Bishop, and Rustan Leino. We thank Judith Bishop for organizing the gift, and all aspects of the visit, and Rustan Leino for the tool Dafny, and for many useful discussions.

Chapter 1: Introducing Dafny

Dafny is a hybrid language, with functional and object oriented features, which can automatically check programs against specifications. By defining theorems, and using pre- and post-conditions, loop invariants, assertions, and many other constructs, you can build fully verified programs. This means that any program accepted by the Dafny verifier, is guaranteed to be totally correct – ie terminate (it will never crash or loop infinitely), and partially correct (if it terminates then it satisfies its specification). .

Dafny uses a very powerful theorem prover called Z3. This allows it to prove the correctness of some, but not all, programs. For the cases where the theorem prover cannot do the proofs unaided, it may require some help from you.

In this course, we shall study how to write Dafny programs, how to write specifications, and how to augment Dafny code with more help for the theorem prover. We will not cover all features of Dafny.

NOTE: The way you prove a program in Dafny is similar, but not identical, to the way you write by hand the proof for a program in the course Reasoning about Programs. In particular, Dafny can do many steps automatically, and your proofs in Dafny will shorter than those for our course. Moreover, Dafny uses a different syntax, and proofs in Dafny look, in some way, like programs.

An Example: Binary Search

As an example, here is a fully verified implementation of the binary search algorithm:
The parts highlighted in grey are not part of the program itself; they are the parts that are checked by the Dafny verifier.

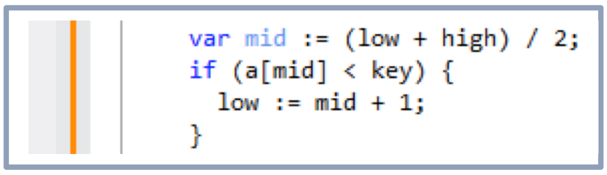
```
method BinarySearch(a: array<int>, key: int)
  returns (index: int)
  requires a != null && sorted(a);
  ensures index >= 0 ==> index < a.Length && a[index] == key;
  ensures index < 0 ==> forall k :: 0 <= k < a.Length ==> a[k] != key;
{
  var low := 0;
  var high := a.Length;
  while (low < high)
    invariant 0 <= low <= high <= a.Length;
    invariant forall i ::
      0 <= i < a.Length && !(low <= i < high) ==> a[i] != key;
  {
    var mid := (low + high) / 2;
    if (a[mid] < key) {
      low := mid + 1;
    }
    else if (key < a[mid]) {
      high := mid;
    }
    else {
      return mid;
    }
  }
  return -1;
}
```

Using Dafny from Visual Studio

Visual Studio may be upgraded so as to incorporate Dafny. To use Dafny in Visual Studio, open a .dfy file in Visual Studio. Try opening the provided “BinarySearch.dfy” file; you should see correct syntax highlighting.

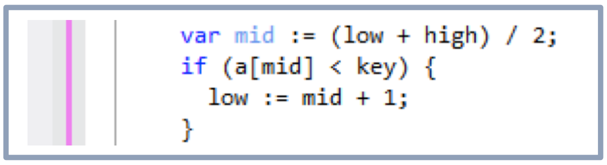
Visual studio will automatically compile and verify your code once it is loaded. If you edit a part of your code it will re-compile just that part. It uses coloured bars on the left-hand side of the editor window to let you know what phase different parts of your code are at:

An orange bar means that this your code differs from the latest version..



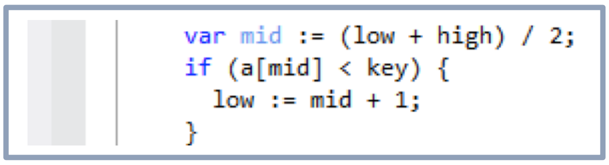
```
var mid := (low + high) / 2;
if (a[mid] < key) {
  low := mid + 1;
}
```

A purple bar means that your code is being verified by the theorem prover.



```
var mid := (low + high) / 2;
if (a[mid] < key) {
  low := mid + 1;
}
```

No bar means the checking is complete, but not necessarily that your code is correct.

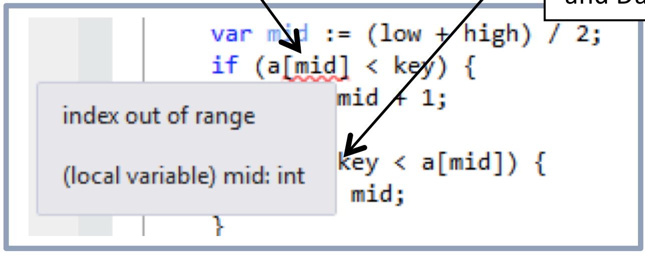


```
var mid := (low + high) / 2;
if (a[mid] < key) {
  low := mid + 1;
}
```

Squiggly underlining indicates incorrect or unverifiable code (like a spell checker in a word processor). Mouse-over the squiggly line to get a popup indicating the exact problem. Here’s an example where I edited the BinarySearch method to prompt and error:

This squiggly underline shows that there is a problem with this part of the code.

The mouse-over message tells us that Dafny thinks this array might be given an index outside of its bounds at this point. Array indexing must always be valid in Dafny (otherwise the program crashes, and Dafny programs cannot crash).



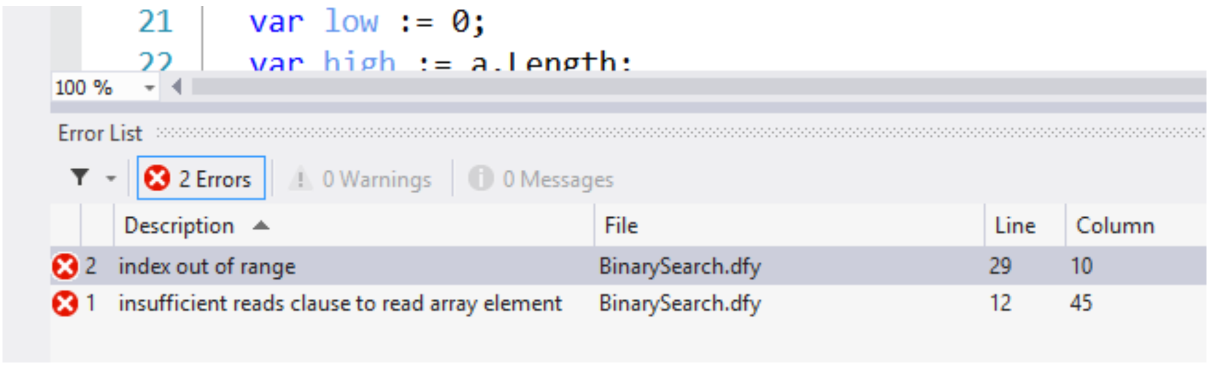
```
var mid := (low + high) / 2;
if (a[mid] < key) {
             mid + 1;

             key < a[mid]) {
              mid;
}
```

index out of range

(local variable) mid: int

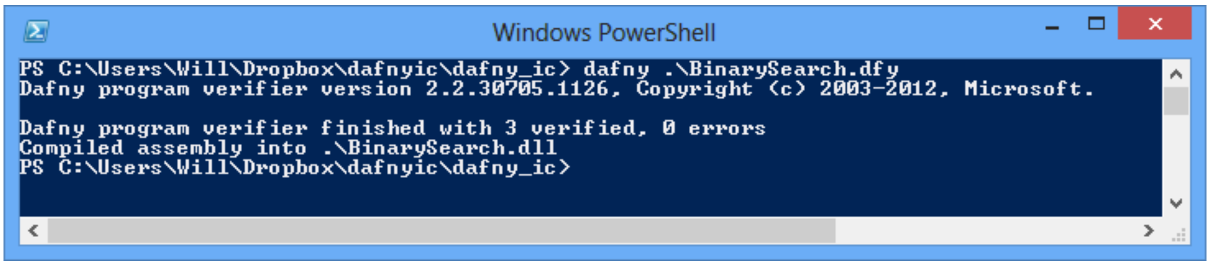
Moreover, at the bottom of the window you can see a complete list of all errors. (You might need to enable the viewing of errors, by selecting the View Menu, and then clicking Error List).


```
var low := 0;
var high := a.Length;
```

	Description	File	Line	Column
2	index out of range	BinarySearch.dfy	29	10
1	insufficient reads clause to read array element	BinarySearch.dfy	12	45

Using Dafny from the command line

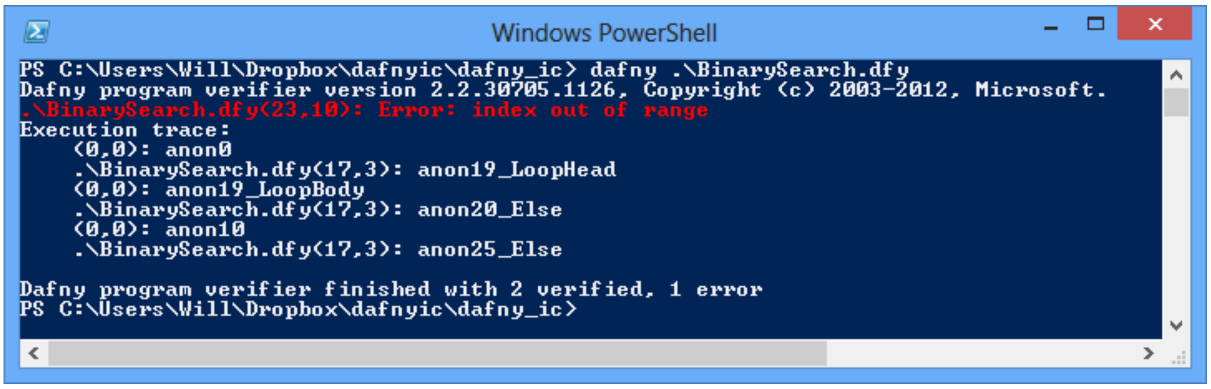
While we recommend using the Visual Studio integration under Windows (because it's cooler); you can also run Dafny from the command line in both Windows and Linux. For example, here is the compilation of "BinarySearch.dfy"



```
PS C:\Users\Will\Dropbox\dafnyic\dafny_ic> dafny .\BinarySearch.dfy
Dafny program verifier version 2.2.30705.1126, Copyright (c) 2003-2012, Microsoft.

Dafny program verifier finished with 3 verified, 0 errors
Compiled assembly into .\BinarySearch.dll
PS C:\Users\Will\Dropbox\dafnyic\dafny_ic>
```

Here is the compilation of the same file, with the error we demonstrated earlier:



```
PS C:\Users\Will\Dropbox\dafnyic\dafny_ic> dafny .\BinarySearch.dfy
Dafny program verifier version 2.2.30705.1126, Copyright (c) 2003-2012, Microsoft.
.\BinarySearch.dfy(23,10): Error: index out of range
Execution trace:
    (0,0): anon0
    .\BinarySearch.dfy(17,3): anon19_LoopHead
    (0,0): anon19_LoopBody
    .\BinarySearch.dfy(17,3): anon20_Else
    (0,0): anon10
    .\BinarySearch.dfy(17,3): anon25_Else

Dafny program verifier finished with 2 verified, 1 error
PS C:\Users\Will\Dropbox\dafnyic\dafny_ic>
```

Installing Dafny on a home machine

To download Dafny visit http://dafny.codeplex.com and follow the installation instructions. You will also need to install Z3 (the theorem prover Dafny uses), from http://z3.codeplex.com. Once downloaded you can execute “Dafny.exe” from the command line, or if you have Visual Studio run “DafnyLanguageService.vsix” to install the Visual Studio integration.

One thing they neglect to mention for the Visual Studio integration is that you need to copy the contents of the Z3 binary directory (e.g. “C:\Program Files\Z3\bin”) into the Dafny Visual Studio language extension directory (e.g. “C:\Users\...\AppData\Local\Microsoft\VisualStudio\10.0\Extensions\Microsoft Research\DafnyLanguageMode\1.0”) after installation.

Chapter 2
Functional programming in Dafny

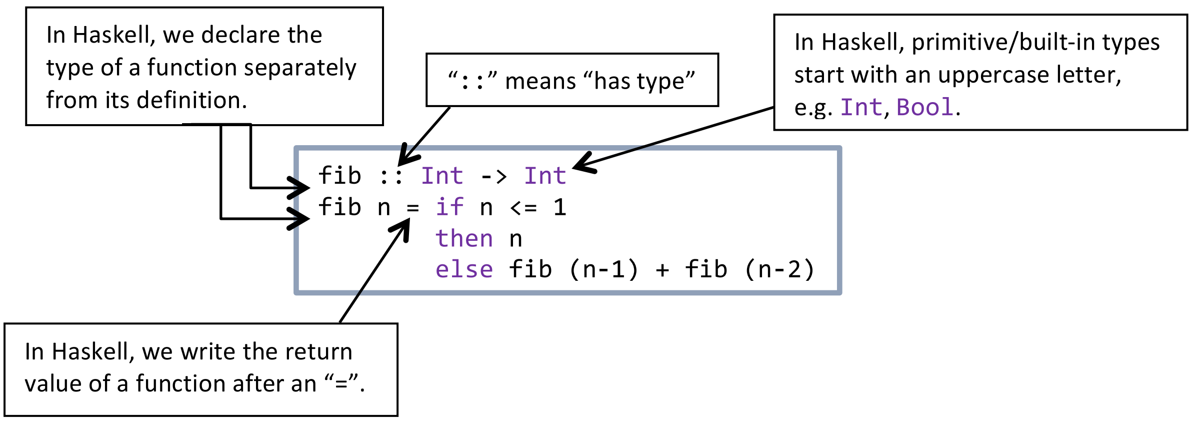
We will now study how to write functional programs in Dafny, how to write their specification, and how to help Dafny prove that the programs satisfy their specifications.

1. From Haskell to Dafny - the little differences

The functional part of Dafny is similar to a subset of Haskell. We shall study two examples to illustrate some small cosmetic differences between the two languages.

1.1 Fibonacci in Haskell and in Dafny

Haskell



In Haskell, we declare the type of a function separately from its definition.

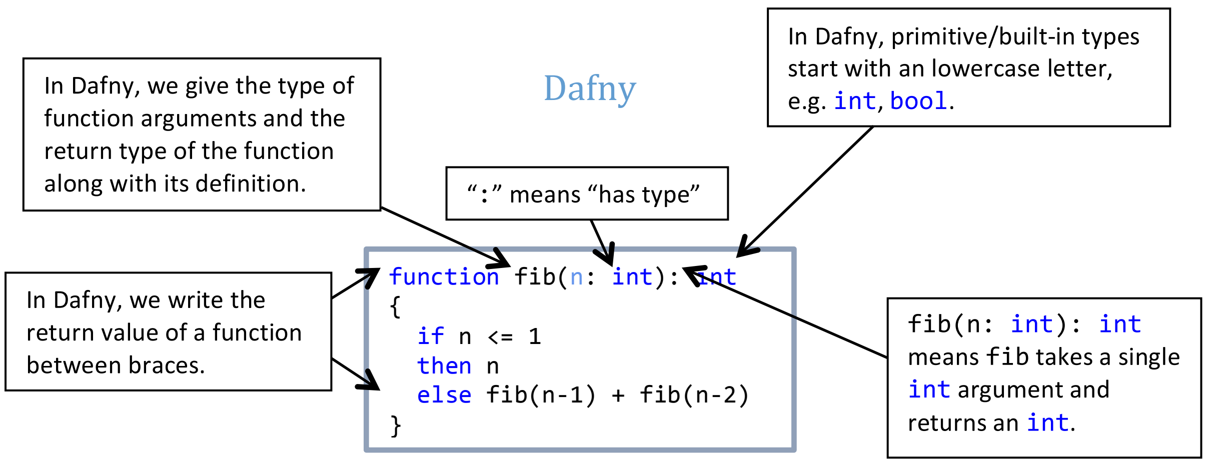
"::" means "has type"

In Haskell, primitive/built-in types 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)
```

In Haskell, we write the return value of a function after an "=".

Dafny



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

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

":" means "has type"

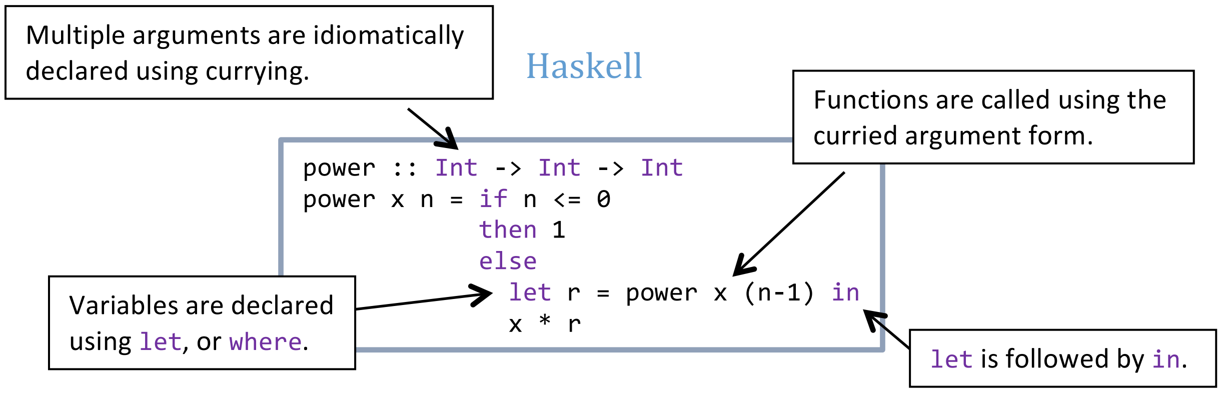
In Dafny, we write the return value of a function between braces.

```
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.

1.2 Power function in Haskell and in Dafny

Here is a function calculating the n-th power of 2 expressed in Haskell and Dafny. This example illustrates multiple function arguments, and the declaration of variables:



Multiple arguments are idiomatically declared using currying.

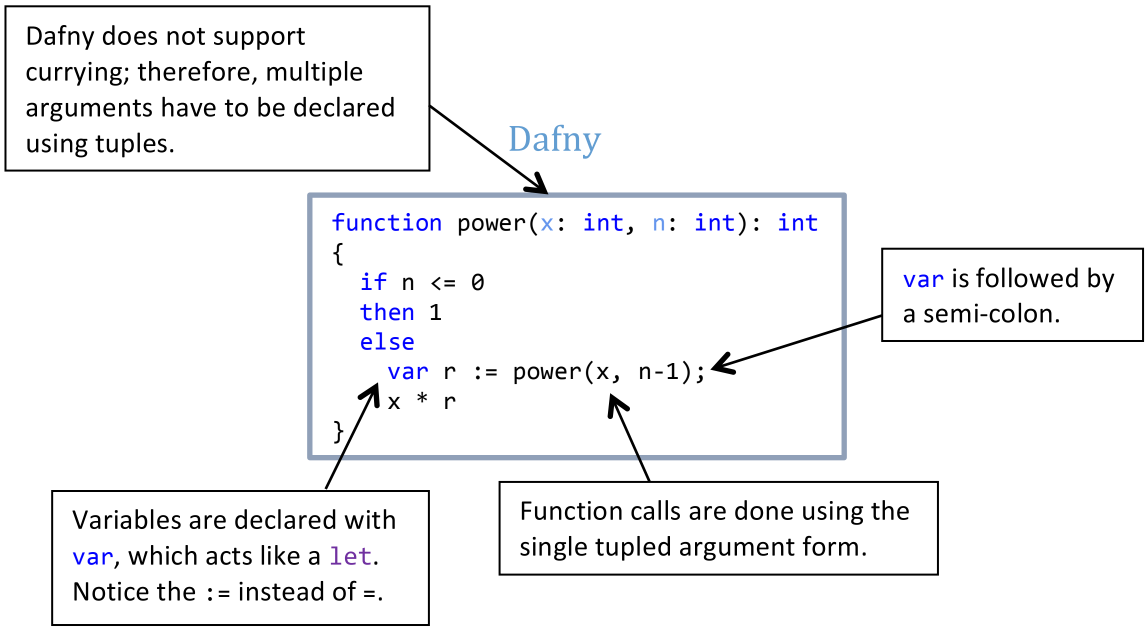
Haskell

Functions are called using the curried argument form.

```
power :: Int -> Int -> Int
power x n = if n <= 0
            then 1
            else
              let r = power x (n-1) in
              x * r
```

Variables are declared using let, or where.

let is followed by in.



Dafny does not support currying; therefore, multiple arguments have to be declared using tuples.

Dafny

```
function power(x: int, n: int): int
{
  if n <= 0
  then 1
  else
    var r := power(x, n-1);
    x * r
}
```

var is followed by a semi-colon.

Variables are declared with var, which acts like a let. Notice the := instead of =.

Function calls are done using the single tupled argument form.

2. User defined datatypes and pattern matching

We shall now show how to define algebraic datatypes in Dafny (the `data` declarations in Haskell), and then how to perform pattern matching on these types.

2.1 Defining a binary tree

Haskell

```haskell
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

In Haskell, type parameters are just written after the type name, in Dafny they are given in angle brackets.

In Haskell we give constructor argument types delimited by spaces, in Dafny they are parenthesised and comma delimited.

Dafny

```dafny
datatype tree<T> = Leaf(T) | Node(tree<T>, tree<T>)
```

In Haskell, datatype names must start with an uppercase letter. In Dafny we can use either.

2.2 Defining a mirror function

This example demonstrates pattern matching on our recursive Tree data-type.

Haskell

In Haskell, the two definitions of mirror are equivalent. The “equational” style of the first definition does not exist in Dafny however. You can only use the second style.

```haskell
mirror :: Tree a -> Tree a
mirror (Leaf x) = Leaf x
mirror (Node l r) = Node (mirror r) (mirror l)
```

```haskell
mirror :: Tree a -> Tree a
mirror t = case t of
           Leaf x -> Leaf x
           Node l r -> Node (mirror r) (mirror l)
```

In Dafny, any type parameters used may be given after the function name too. The function is also seen as being parameterised by this type.

Dafny

```dafny
function mirror<T>(t: tree<T>): tree<T>
{
  match t
  case Leaf(x) => Leaf(x)
  case Node(l, r) => Node(mirror(r), mirror(l))
}
```

2.3 Implicit Polymorphism in Dafny

Note that Dafny can implicitly introduce type missing type parameters into function declarations. Therefore, the following form compiles

```
datatype tree<T> = Leaf(T) | Node(tree<T>, tree<T>);

function mirror(t: tree): tree
{
  match t
  case Leaf(x) => Leaf(x)
  case Node(l, r) => Node(mirror(r), mirror(l))
}
```

3. Pre-conditions and total functions

Dafny requires that functions should never crash, and not loop for ever. We will now illustrate how we can stop functions from crashing or looping, by using pre-conditions which restrict their inputs.

3.1 Fixing our Fibonacci function

Here is the Dafny Fibonacci function we defined earlier:

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

This is not a correct definition however, as `fib(-1)` does not equal 1.
The function should be undefined for n < 0.

In Haskell we can define `fib` like this:

```
fib :: Int -> Int
fib n = if n == 0 || n == 1
        then 1
        else fib (n-1) + fib (n-2)
```

This function is no longer defined at for n < 0, it will loop instead.

However, try to compile this in Dafny:

```
function fib(n: int): int
{
  if (n == 0 || n == 1)
  then 1
  else fib(n-1) + fib(n-2)
}
```

You will get:

```
Error: decreases expression must be bounded below by 0
```

This is because Dafny only accepts definitions if it is able to prove that they terminate. Therefore, we need to define the function fib in such a way, that it is never called with an n < 0.

We can add pre-conditions as comments to Haskell functions , for example:

Such comments help programmers, but mean nothing to the compiler, which will accept any value from `Int` as an input to `fib`.

```
fib :: Int -> Int
-- pre: n >= 0
fib n = if n == 0 || n == 1
        then 1
        else fib (n-1) + fib (n-2)
```

In Dafny, we can add pre-conditions to a function using "`requires`":

Now this function only accepts arguments greater than or equal to zero.

```
function fib(n: int): int
  requires n >= 0;
{
  if (n == 0 || n == 1)
  then 1
  else fib(n-1) + fib(n-2)
}
```

In effect, we have refined the input type of `fib` from "all integers" to "all integers ≥ 0". Every time `fib` is called Dafny will check to make sure its input is ≥ 0.[1]

3.2 Defining list destructors

The head and `tail` functions from Haskell are known as the list "destructors", as they break a list up into the parts that the cons constructor put together.

These are not total functions, as they are undefined (will crash) when given an empty list as input.

```
head :: [a] -> a
head (x:xs) = x

tail :: [a] -> [a]
tail (x:xs) = xs
```

Hence we cannot define head and `tail` without further thought in Dafny; the functions below will not compile:

The error we get is:
    missing case in case statement: Nil

We are missing the "`case Nil => …`"
    in both of these pattern matches

"`requires`" to the rescue again:
we need to make sure that head/tail

```
datatype list<A> = Nil | Cons(A, list<A>);

function head<A>(xs: list<A>): A
{
  match xs
  case Cons(y, ys) => y
}

function tail<A>(xs: list<A>): list<A>
{
  match xs
  case Cons(y, ys) => ys
}
```

[1] To observe this, try compiling the following
function g1(n: int) : int { fib(-4) }

function g2(n: int) : int { fib(n) }

function g3(n: int) : int requires n >= 0; { fib(n) } are only given non-empty lists.

These definitions will compile:

```
function head<A>(xs: list<A>): A
  requires xs != Nil;
{
  match xs
  case Cons(y, ys) => y
}

function tail<A>(xs: list<A>): list<A>
  requires xs != Nil;
{
  match xs
  case Cons(y, ys) => ys
}
```

4. Proving function properties

In this section we will demonstrate how to specify and prove properties of functions within Dafny.

4.1 Specifying natural numbers, and Even and Odd predicates, and addition

Here is the standard definition of ℕ as a recursive data-type in Dafny, with a recursive function add:

Building on this definition, we can proceed and define predicates for Odd and Even numbers. Note that the predicates are inductively defined.

```
datatype natr = Zero | Succ(natr);

predicate Odd(z: natr)
{
   match x
      case Zero => false
      case Succ(Zero) => true
      case Succ(z) => Odd(z)
}

predicate Even(z: natr)
{
   match x
      case Zero => true
    case Succ(Zero) => false
      case Succ(z) => Even(z)
}
```

Note: Rather than defining natr, we could use Dafny's built-in `nat` type, defined as `int` ≥ 0, but defining our own datatype and proving properties will be a useful example.

And we can also specify a recursive function add:

```
function add(x: natr, y: natr): natr
{
  match x
  case Zero => y
  case Succ(x') => Succ(add(x', y))
}
```

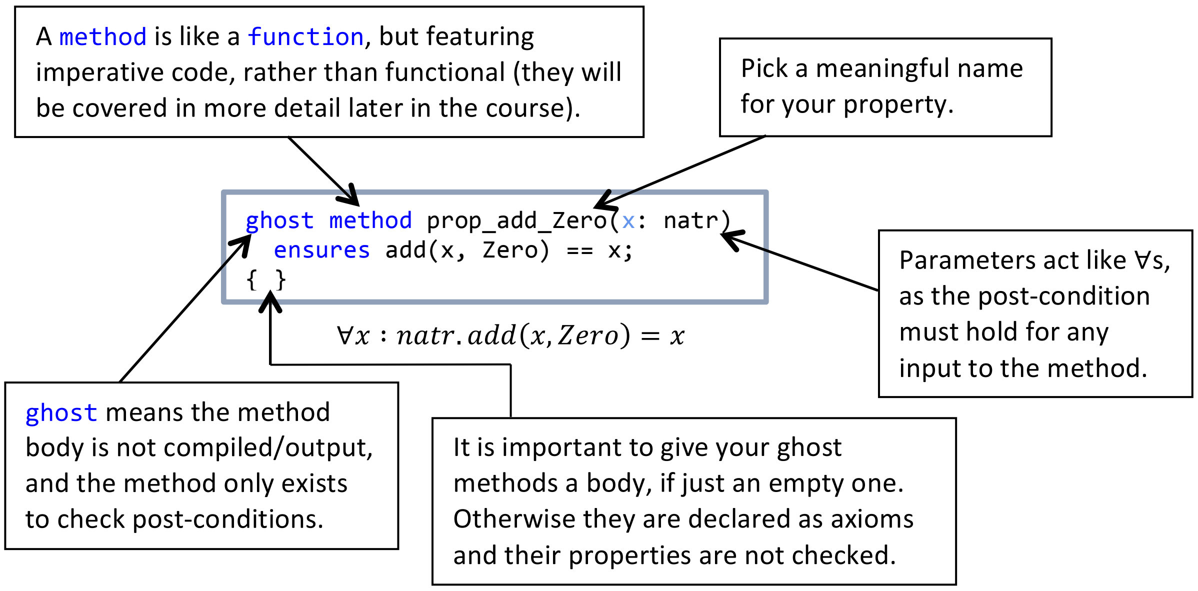
One simple property we would like to be able to prove is $\forall x : \mathbb{N}.\, x + 0 = x$.
That is to say:

$$\forall x : natr.\, add(x, Zero) = x$$

4.2 Using ghost methods to express properties

We declare properties by using post-conditions on "`ghost method`"s. Ghost methods are code that is used by the Dafny verifier, but are not part of the executable programme. Post-conditions are declared using the "`ensures`" keyword; they specify properties that should hold after the method executes.

4.2.a Property `prop_add_Zero`: $\forall x : natr.\, add(x, Zero) = x$



A `method` is like a `function`, but featuring imperative code, rather than functional (they will be covered in more detail later in the course).

Pick a meaningful name for your property.

```
ghost method prop_add_Zero(x: natr)
  ensures add(x, Zero) == x;
{ }
```

$\forall x : natr.\, add(x, Zero) = x$

Parameters act like ∀s, as the post-condition must hold for any input to the method.

`ghost` means the method body is not compiled/output, and the method only exists to check post-conditions.

It is important to give your ghost methods a body, if just an empty one. Otherwise they are declared as axioms and their properties are not checked.

Start a new Dafny file and input the definitions for `natr`, `add` and `prop_add_Zero`, making sure it compiles.
Notice that Dafny can verify `prop_add_Zero` *automatically*, even though the hand-proof requires induction!

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

Any property that requires a single structural induction step and no auxiliary lemmas Dafny should be able to prove automatically. Here is another such property of $+$ which you should input:

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

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

4.3 Verification when proofs require auxiliary lemmas

Now we will specify and prove that our add function is commutative, i.e.

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

Input prop_add_comm and re-run Dafny.

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

You will see that Dafny fails to automatically prove prop_add_comm.:

Error: A postcondition might not hold on this return path.

As we saw earlier, Dafny will apply function definitions and inductive hypotheses automatically, hence why it can prove prop_add_Zero and prop_add_Succ by itself, as this is all these proofs require. However, Dafny does not apply lemmas automatically, and prop_add_comm requires two.

To see how we verify this in Dafny we first look at the hand-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: $natr$
$add(succ(x), y)$
$= Succ\big(add(x, y)\big)$ (by definition of add)
$= Succ\big(add(y, x)\big)$ (by IH)
$= add\big(y, Succ(x)\big)$ (by prop_add_Succ)

We can now write the corresponding proof in Dafny. We use a case analysis to reflect the Base case, and the inductive step. We use `calc`-expressions, to justify each step. The application of the induction hypothesis corresponds to calling the ghost method itself (here `prop_add_comm`) with "smaller" arguments:

"`match x`" indicates proof by induction on x.

Base Case

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);
      }
    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'));
      }
  }
}
```

Justifications for each of the steps.

Lemmas indicated by "calling" the corresponding ghost method.

The Induction Hypothesis is indicated through "calling" the ghost method with the appropriate parameters.

Because we are in a `method`, `match` requires braces – a syntactic difference between `method`s and `function`s.

Dafny can do several of the above steps implicitly, and therefore the following will also verify:

```
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'));
      }
  }
}
```

Dafny can also verify the property just by calling the appropriate ghost methods (lemmas) in the base case and inductive step. Therefore, the following version also verifies:

```
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');
  }
}
```

Calling a ghost method (or any method) adds its post-conditions as usable properties for the Dafny checker.

By calling `prop_add_Succ(y, x')` we have added:

`Succ(add(y, x’)) == add(y, Succ(x’))`

to the facts which Dafny knows at this point, and which it can use to complete the verification.

4.3 Expressing Implications

In order to express an implication in a ghost lemma, we can use the requires clause to express the condition, and the ensures clause, to express the conclusion. For example, we can encode

$$\forall x, y : \mathbb{N}.\, Odd(x) \wedge Odd(y) \;\Rightarrow Even(x+y)$$

through

```
ghost method add_Odd_Odd_is_Even(u: natr, v: natr)
   requires Odd(u) && Odd(v);
   ensures Even(add(u,v));
{ }
```

4.4 Conclusions

Declaring a ghost method == Expressing a lemma.

Calling a ghost method == Using a lemma.

Body of a ghost method == Proving a lemma.

5. Exercises

You can now proceed to do the exercises titled `Exercises_Lists` and `Exercises_Change`.

Chapter 3
More on Proof Development

We will now discuss the construction of larger proofs in Dafny, and the tools Dafny provides for the development of proofs in a "top-down" manner, and which allows us to concentrate on the "architecture" of the proof, before going on to the detailed proof steps. Dafny offers two features:

- `assume` statements, which allow us to assume, and use an assertion without proving it,
- the declaration of ghost methods which stand for lemmas, but without supplying the ghost method's body; this amounts to a lemma which has not yet been proven, but which can be used in other proofs – similar to the declaration of a function/method, which can be called even before the appropriate method/function body is available.

Of course, a proof is not complete until all proof obligations have been discharged, *i.e.*, all `assume` statements have been replaced, and all ghost methods have been supplied with appropriate bodies.

In this chapter, we will also show how hand-written proofs can guide the development of the Dafny proof, while the Dafny tool can give us confidence that all proven properties hold.

As an example, we will study the verification of the function quick-sort ($qsort$), invented by Tony Hoare in 1960 (http://en.wikipedia.org/wiki/Quicksort), and which sorts an array or list of n elements by applying O(*n* log *n*) comparisons. In this chapter we will look at the functional version of the function, and we shall prove:

$$\forall xs: list\langle\mathbb{Z}\rangle.\, is\_sorted\big(qsort(xs)\big) \;\wedge\; perm(xs, qsort(xs))$$

Definitions are provided in "Quicksort.dfy". The function `qsort(xs)` sorts a list `xs`, the predicate `is_sorted(x)` says that its argument `xs` is sorted, and the predicate `perm(xs,ys)` expresses that `xs` is a permutation of `ys`. Moreover, the functions `take_le(y,ys)`, resp. `take_gt(y,ys)` return the sublist of `ys` containing all elements less-or equal to, respectively greater than, `y`.

1. Termination of the function qsort

Here is the body of the function qsort

```
function qsort(xs: list<int>): list<int>
{
  match xs
  case Nil => Nil
  case Cons(y, ys) =>
    var sle := qsort(take_le(y, ys));
    vars gt := qsort(take_gt(y, ys));
    app(sle, Cons(y, sgt))
}
```

The two recursive calls with the terms `take_le(y, ys)` and `take_gt(y, ys)` as arguments.

Dafny requires that every function terminates for every valid input, and will not accept a function definition unless it can prove this. It will attempt to prove termination automatically, by trying to show that any recursive calls take place with arguments which are *structurally* smaller than the original arguments. However, in the body of qsort, the recurisve calls are made with the arguments `take_le(y, ys)` and `take_gt(y, ys, which are not necessarily` *structurally* decreasing. Therefore, the function as defined above does not compile in Dafny.

However, we can give Dafny further information to help it prove termination. Namely, we know that that the *length* of the input decreases with each nested call. Therefore, we update the definition of qsort through a `decreases` clause as follows:

To show termination, we supply a term whose value decreases with every recursive call. In this case the length of the input list.

```
function qsort(xs: list<int>): list<int>
  decreases len(xs);
{
  match xs
  case Nil => Nil
  case Cons(y, ys) =>
    var sle := qsort(take_le(y, ys));
    var sgt := qsort(take_gt(y, ys));
    app(sle, Cons(y, sgt))
}
```

In general, a `decrease`s clause provides some "measure" that decreases on every recursive call to the function within the function body.

Note: The argument that qsort terminates because every recursive call takes place on parameters which are smaller in terms of a decreasing measure corresponds to well-founded induction. Namely, we define the following well-founded ordering ($\prec$):

$$ys \prec xs \Leftrightarrow len(ys) < len(xs)$$

i.e. ys is "smaller than" xs iff the length of ys is less than the length of ys. And we then prove the lemma

$$\forall xs: list\langle\mathbb{Z}\rangle.\ \ qsort(xs)\quad terminates.$$

using the well-founded ordering on xs.
An equivalent view is, that we prove that

$$\forall n: \mathbb{N}. \forall xs: list\langle\mathbb{Z}\rangle.\ \ length(xs) = n \Rightarrow\ \ qsort(xs)\quad terminates.$$

in which case the proof goes by mathematical induction.

However, even though we have provided the decreases-measure, the definition of `qsort` still does not compile, and Dafny complains: “failure to decrease termination measure”. In order to find out why, we consider the proof of termination of `qsort` by well founded ordering.

Take a $xs: list\langle\mathbb{Z}\rangle$ arbitrary
Inductive Hypothesis $\forall zs: list\langle\mathbb{Z}\rangle. zs \prec xs \Rightarrow qsort(zs) \quad terminates.$
To Show: $qsort(xs)$ terminates

1st Case: $xs=Nil$
To Show: $qsort(Nil)$ terminates
by definition of $qsort$

2nd Case: $xs \neq Nil$. Therefore $xs=Cons(y,ys)$ for some $y:\mathbb{Z}$ and $ys: list\langle\mathbb{Z}\rangle$.
To Show: $qsort(Nil)$ terminates

(1)	$take\_le(y,ys)$ terminates	by body of take_le (recursive calls on structurally smaller arguments)
(2)	$take\_gt(y,ys)$ terminates	similar argument to (1)
(3)	$take\_gt(y,ys) \prec ys$	by Auxiliary Lemma *1*
(4)	$qsort(take\_le(y,ys))$ terminates	by (1), (2), (3) and IH
(5)	$qsort(take\_gt(y,ys))$ terminates	similar to (1)-(4), using Auxiliary Lemma 2
(6)	$qsort(xs)$ terminates	by (1), (2), by body of $qsort$,

Auxiliary Lemma 1: $\forall y: \mathbb{Z}. \forall ys: list\langle\mathbb{Z}\rangle. \quad take\_le(y, ys) \prec ys$

Auxiliary Lemma 2: $\forall y: \mathbb{Z}. \forall ys: list\langle\mathbb{Z}\rangle. \quad take\_gt(y, ys) \prec ys$

The important observation is, that in order to prove termination of `qsort`, we need the two auxiliary lemmas from above.

We now turn back to the argument about termination of `qsort` in Dafny. Remember, that Dafny requires any function defined to be terminating, and therefore, proof of termination cannot be separated from the definition of the function (as we did in the hand-written proof above). Instead, the definition of the function, the function body, and the definition the functions used within the function body, need to provide sufficient information for Dafny to be able to “convince itself” that the function terminates.

For this reason, we make the two auxiliary lemmas, i.e. Auxiliary Lemma 1 and Auxialliary Lemma 2, part of the definition of `take_le` and `take_gt`:

```
function take_le(n: int, xs: list<int>): list<int>
  ensures len(take_le(n, xs)) <= len(xs);
```

```
function take_gt(n: int, xs: list<int>): list<int>
  ensures len(take_gt(n, xs)) <= len(xs);
```

These post-conditions, along with "`decreases len(xs);`" in the definition of `qsort`, allow Dafny to prove that the `qsort` function terminates. Compare these three lines with of Dafny with our longer, hand-written proof. As we said earlier, Dafny is often able to find proof steps implicitly.

2 First stab at Partial correctness: structural vs well-founded induction

We will now move onto proving the correctness of our `qsort` function.

The technique used in this sheet is "top-down" proof, where you start with the eventual goal, and iteratively break it down into simpler and simpler sub-properties.

Here is our eventual goal:

$$\forall xs{:}\, list\langle \mathbb{Z}\rangle .\, is\_sorted\big(qsort(xs)\big) \quad \wedge \quad perm(xs, qsort(xs))$$

2.1 First attempt: Structural Induction.

To simplify the discussion, we concentrate first on the following, simpler proof obligation:

$$\forall xs{:}\, list\langle \mathbb{Z}\rangle .\, is\_sorted\big(qsort(xs)\big)$$

We define the proof goal in the ghost method `prop_qsort`.

```
ghost method prop_qsort(xs: list<int>)
  ensures is_sorted(qsort(xs));
{
  assume is_sorted(qsort(xs));
}
```

The `assume` keyword introduces an assumption which is not checked.

Since we have `assume`d the goal, the file will compile and be verified. It is important to recompile after each step, so as to make sure that we haven't made any mistakes.

However, our proof is not complete, and we need to replace the `assume` clause with appropriate statements. To do this, we break our property down into simpler sub-properties. We first case-split on `xs`, and since the base case is trivial, we expect Dafny to be able to prove it without further assistance. Hence, the following code compiles:

```
ghost method prop_qsort(xs: list<int>)
  ensures is_sorted(qsort(xs)) ;
{
  match xs {
  case Nil =>
  case Cons(y, ys) =>
    var sle := qsort(take_le(y, ys));
    var sgt := qsort(take_gt(y, ys));
    assume is_sorted(app(sle, Cons(y, sgt)));
  }
}
```

We will now try to prove the inductive step (case where `xs` is `Cons(y,ys)`). To do that, we expect to be able to apply the inductive hypothesis, and obtain, that `is_sorted(sgt) and that is_sorted(sgt).` To try this out, we can type

```
ghost method prop_qsort(xs: list<int>)
  ensures is_sorted(qsort(xs)) ;
{
  match xs {
  case Nil =>
  case Cons(y, ys) =>
    var sle := qsort(take_le(y, ys));
    var sgt := qsort(take_gt(y, ys));
    assert is_sorted(sle) && is_sorted(sgt);
    assume is_sorted(app(sle, Cons(y, sgt)));
  }
}
```

However, the ghost method from above does not compile, and Dafny complains: `assertion violation`.

Similarly, the following ghost method does not compile:

```
ghost method prop_qsort(xs: list<int>)
  ensures is_sorted(qsort(xs)) ;
{
  match xs {
  case Nil =>
  case Cons(y, ys) =>
    var sle := qsort(take_le(y, ys));
    var sgt := qsort(take_gt(y, ys));
    prop_qsort(sle);
    …
  }
}
```

And Dafny gives the error message: cannot prove termination, try supplying a decreases clause.

To understand why this is the case, we need to be clear as to what our inductive principle is. In the method `prop_qsort` as defined above, the inductive principle is structural induction (because Dafny applies structural induction unless asked otherwise. And since `sle` and `sgt` are not necessarily structurally smaller than `xs`.

We can understand the above, if we look into the attempt to hand-prove the property by structural induction:

Base Case
To Show: $is\_sorted(qsort(Nil))$
by definition of $qsort$

Inductive Step Take arbitrary $y{:}\,\mathbb{Z}, ys{:}\,list\langle\mathbb{Z}\rangle$
Inductive Hypothesis $is\_sorted(qsort(ys))$
To Show: $is\_sorted(qsort(y :: ys))$

The induction hypothesis **cannot** be used to gives us that $is\_sorted(qsort(take\_le(y, ys)))$, nor that $is\_sorted(qsort(take\_gt(y, ys)))$

Therefore, we cannot prove correctness of quicksort by structural induction.

We can, however, prove it by well-founded induction, similar to what we did with the proof of termination in the previous section.

2.2 Second attempt: Well-founded induction

We will be using the well founded ordering $\prec$, defines as

$$ys \prec xs \Leftrightarrow len(ys) < len(xs)$$

Here is the structure of the proof:

Take a $xs: list\langle\mathbb{Z}\rangle$ arbitrary
Inductive Hypothesis $\forall zs: list\langle\mathbb{Z}\rangle.\ zs \prec xs \ \Rightarrow \ is\_sorted(qsort(zs))$.
To Show: $is\_sorted(qsort(xs))$

1st Case: $xs{=}Nil$
To Show: $is\_sorted(qsort(Nil))$
by definition of $qsort$

2nd Case: $xs \neq Nil$. Therefore $xs{=}Cons(y,ys)$ for some $y{:}\mathbb{Z}$ and $ys: list\langle\mathbb{Z}\rangle$.
To Show: $is\_sorted(qsort(zs))$

(1) $is\_sorted(qsort(take\_le(y,ys)))$ by Auxiliary Lemma 1, and IH
(2) $is\_sorted(qsort(take\_gt(y,ys)))$ by Auxiliary Lemma 2, and IH
(3) $is\_sorted(app(qsort(take\_le(y,ys))), Cons(y, qsort(take\_gt(y,ys)))$
by (1), (2), and further Auxiliary Lemmas

The proof above shows how to obtain that $is\_sorted(qsort(take\_le(y,ys)))$ and $is\_sorted(qsort(take\_le(y,ys)))$, but leaves open how to use these results to obtain that $is\_sorted(app(qsort(take\_le(y,ys))), Cons(y, qsort(take\_gt(y,ys)))$.

We can reflect the handwritten proof into Dafny as follows:

Since we are defining a method we must prove it terminates. Our recursive calls are the same as those to qsort, so we can use the same

```
ghost method prop_qsort(xs: list<int>)
  ensures is_sorted(qsort(xs));
  decreases len(xs);
{
  match xs {
  case Nil =>
  case Cons(y, ys) =>
    var le := qsort(take_le(y, ys));
    assert is_sorted(le);

    var gt := qsort(take_gt(y, ys));
    assert is_sorted(gt);

    assume is_sorted(app(le, Cons(y, gt)));
  }
}
```

Again, we recompile, to check that this is a valid step.

Our proof is still incomplete because of the assume clause.

3. Partial correctness

We now turn our attention to our full proof aim

$$\forall xs: list\langle\mathbb{Z}\rangle.\, is\_sorted\big(qsort(xs)\big) \land \;\; perm(xs, qsort(xs))$$

and re-organize the code of `prop_qsort` slightly, to make it easier to work on the proof, as follows:

```
ghost method prop_qsort(xs: list<int>)
    ensures is_sorted(qsort(xs)) && perm(xs, qsort(xs));
    decreases len(xs);
{
  match xs {
  case Nil =>
  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);
  }
}
```

In order to complete the proof, we now think about the reasons that `qsort` is correct:

(1)	le[0..le.length) <= y < gt[0..gt.length)	properties take_le, take_gt
(2)	le++[y]++gt is perm. of xs	properties take_le, take_gt
(3)	sle is perm. of le,	(1), and IH
(4)	sle is sorted	IH
(5)	sgt is perm. of gt	(2), and IH
(6)	sgt is sorted	IH
(7)	sle[0..le.length) <= y < sgt[0..gt.length)	(1), (2), (3), (5), properties of perm
(8)	`app(sle, Cons(y, sgt))` is sorted	(4), (6), (7) and properties of ++
(9)	`app(sle, Cons(y, sgt))` is perm. xs	(3), (5), (2) and properties of ++, perm

The argument from above mentions further properties which we will need to prove wither through adding postconditions to our functions, or through the provision of further ghost methods, and their call within the body of prop_qsort.

In order to be able to discharge (1), we add the following postcondition to take_le and take_gt:

```
function take_le(n: int, xs: list<int>): list<int>
   ensures len(take_le(n, xs)) <= len(xs);
   ensures forall z:: elem(z,take_le(n, xs)) ==> z<= n;
```

```
function take_gt(n: int, xs: list<int>): list<int>
  ensures len(take_gt(n, xs)) <= len(xs);
  ensures forall z:: elem(z,take_gt(n, xs)) ==> z> n;
```

3.1 Formulating the necessary sublemmas

We now further refine the proof as follows:

```
ghost method prop_qsort(xs: list<int>)
    ensures is_sorted(qsort(xs))  && perm(xs, qsort(xs));
    decreases len(xs);
{
  match xs {
  case Nil =>
  case Cons(y, ys) =>

    var le := take_le(y, ys);
    var gt := take_gt(y, ys);
    assert forall z:: elem(z,le) ==> z<=y;    // (1)
    assert forall z:: elem(z,gt) ==> z>y;     // (1)

    assume perm(ys,app(le,gt));                // (2)

    var sle := qsort(le);
    assert is_sorted(sle) && perm(sle, le );   // (3), (4)
    assume forall z:: elem(z,sle) ==> z<=y;    // (7)


    var sgt := qsort(gt);
    assert is_sorted(sgt) && perm(sgt, gt );    // (5), (6)
    assume forall z:: elem(z,sle) ==> z>y;      // (7)

    var res := app(sle, Cons(y, sgt)) ;
    prop_sorted_app(sle, y, sgt);                // (8)
    assert is_sorted(res);

    prop_app_cons_perm(xs,y,le,sle,gt,sgt);      // (9)
    assert perm( xs, res);
  }
}
```

In the proof above, we `assume`d the properties from steps (2) and (7), and also, we defined two auxiliary lemmas which will be used in steps (8) and (9).

```
ghost method prop_sorted_app(xs: list<int>, y: int, ys: list<int>)
  requires is_sorted(xs);
  requires is_sorted(ys);
  requires forall z:: elem(z,xs) ==> z<=y;
  requires forall z:: elem(z,ys) ==> z>y;
  ensures is_sorted(app(xs, Cons(y, ys)));
```

```
ghost method prop_app_cons_perm<T>
   (xs: list<T>, y: T, us: list<T>, us': list<T>,
    vs: list<T>, vs': list<T>)
    requires perm(xs, Cons(y, app(us,vs)))
          && perm(us,us') && perm(vs,vs');
    ensures  perm(xs, app(us', Cons(y,vs')));
```

We now refine our proof:

In order the eliminate the assume clause in step (2), we need a further auxiliary lemma which states

```
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)) ) ;
```

To eliminate the assume clauses for step (7), we inline the necessary proofs in the code.

Therefore, we obtain:

```
ghost method prop_qsort (xs: list<int>)
    ensures is_sorted(qsort(xs))  && perm(xs, qsort(xs));
    decreases len(xs);
{
  match xs {
  case Nil =>
  case Cons(y, ys) =>

    var le := take_le(y, ys);
    var gt := take_gt(y, ys);
    assert forall z:: elem(z,le) ==> z<=y;        // (1)
    assert forall z:: elem(z,gt) ==> z>y;         // (1)

    prop_take_perm(y,ys,le,gt);
    assert perm(ys,app(le,gt));                   // (2)

    var sle := qsort(le);
    assert is_sorted(sle) && perm(sle, le );      // (3), (4)
    forall (z:int) ensures elem(z,sle) ==> z<=y;  // (7)
         { calc{ elem(z,sle );
                 ==> {  prop_perm_elem(z,sle,le) ;  }
                 elem(z,le);
                 ==>
                 z <=y;  }
          }

    var sgt := qsort(gt);
    assert is_sorted(sgt) && perm(sgt, gt );      // (5), (6)
    forall (z:int) ensures elem(z,sgt) ==> z>y;   // (7)
         { calc{ elem(z,sgt );
                 ==> {   prop_perm_elem(z,sgt,gt) ;  }
                 elem(z,gt);
                 ==>
                 z > y;  }
          }

    var res := app(sle, Cons(y, sgt)) ;
    prop_sorted_app(sle, y, sgt);                  // (8)
    assert is_sorted(res);

    prop_app_cons_perm(xs,y,le,sle,gt,sgt);        // (9)
    assert perm( xs, res);
  }
}
```

Notice, that in order to prove step (7), we formulated a further auxiliary lemma, which guarantees that any x which appears in a list ys, also appears in any permutation of ys:

```
ghost method prop_perm_elem<T>(x: T, ys: list<T>, zs: list<T>)
     requires elem(x,ys) && perm(ys,zs);
     ensures elem(x,zs);
```

3.2 Remaining proof obligations

The proof of the lemma `prop_qsort` as given in the previous page is incomplete, as it uses auxiliary lemmas which we have not yet proven. We still need to prove the following lemmas:

```
ghost method prop_sorted_app(xs: list<int>, y: int, ys: list<int>)
  requires is_sorted(xs);
  requires is_sorted(ys);
  requires forall z:: elem(z,xs) ==> z<=y;
  requires forall z:: elem(z,ys) ==> z>y;
  ensures is_sorted(app(xs, Cons(y, ys)));
```

```
ghost method prop_app_cons_perm<T>
   (xs: list<T>, y: T, us: list<T>, us': list<T>,
    vs: list<T>, vs': list<T>)
    requires perm(xs, Cons(y, app(us,vs)))
          && perm(us,us') && perm(vs,vs');
    ensures  perm(xs, app(us', Cons(y,vs')));
```

```
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)) ) ;
```

```
ghost method prop_perm_elem<T>(x: T, ys: list<T>, zs: list<T>)
     requires elem(x,ys) && perm(ys,zs);
     ensures elem(x,zs);
```

We leave the proof of these lemmas as an exercise. Some of these lemmas may require further the proof of further auxiliary lemmas.

3.3 A more succinct version of prop_qsort

The proof of the lemma `prop_qsort` as given in section 3.1 was written with the aim to provide some explanations as to how the proof works. However some of the clauses are unnecessary for Dafny, as Dafny can make several reasoning steps implicitly. In the following we show a shorter, perhaps less informative version of the proof, which, nevertheless also verifies:

```
ghost method prop_qsort3 (xs: list<int>)
// the "minimal version"
    ensures is_sorted(qsort(xs))  && perm(xs, qsort(xs));
    decreases len(xs);
{
  match xs {
  case Nil =>
  case Cons(y, ys) =>

    var le := take_le(y, ys);
    var gt := take_gt(y, ys);

    prop_take_perm(y,ys,le,gt);

    var sle := qsort(le);
    forall (z:int) ensures elem(z,sle) ==> z<=y;  // (7)
         { calc{ elem(z,sle );
                 ==> {  prop_perm_elem(z,sle,le) ;  }
                 elem(z,le);
                 ==>
                 z <=y;  }  }

    var sgt := qsort(gt);
    forall (z:int) ensures elem(z,sgt) ==> z>y;   // (7)
         { calc{ elem(z,sgt );
                 ==> {   prop_perm_elem(z,sgt,gt) ;  }
                 elem(z,gt);
                 ==>
                 z > y;  }  }

    prop_sorted_app(sle, y, sgt);                  // (8)
    prop_app_cons_perm(xs,y,le,sle,gt,sgt);        // (9)
   }
}
```

How much detail you give in a Dafny proof is, of course a matter of personal style. We advocate the use of assert-clauses to indicate what properties have been established so far.

In contrast, in handwritten proofs we will be following the style proposed in the lectures, breaking the proof into assertion-steps, and justifying each step.

4. Exercises

You can now proceed to do the exercises titled `Exercises_Tail_Recursion`. and `Exercises_Flattening`.

Chapter 4
Imperative programming in Dafny

In this lecture shall study how to write imperative programs in Dafny, how to write their specification, and how to help Dafny prove the specifications. We will use pre-conditions and post-conditions to *specify* our code. We will use loop invariants and assertions to *verify* our code.

1. From Java to Dafny - the little differences

The imperative part of Dafny is similar to a subset of Java. We shall study an example to illustrate some small cosmetic differences between the two languages.

Here is the method "`Find`", which returns the index of a given element in an array if it exists within the array, and the length of the array otherwise.

Array types have special syntax.

The return type of a method goes before its name.

Java

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

Variable declarations must be explicitly typed.

Type parameters are themselves parameterised by the operators they have to support. Here we say that `T` has to support "==".

Array types use the regular parameterised type syntax.

Dafny

Return values are declared after the method's parameters, and must be named, like parameters. "`Find`" returns an `int`, referred to as "`r`".

```
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;
}
```

Recall that pre-conditions are declared using "`requires`". This method would crash if given a `null` array. Because methods cannot crash in Dafny, we must add this pre-condition for the code to compile.

Dafny can infer the types of variables.

No "++" operator in Dafny.

2. Method Specifications

As for functions, we can use method pre-conditions and post-conditions to describe the method's requirements and its beahaviour.

Let's write a specification for our "`Find`" method.

The first post-condition we want to verify is: $r < a.Length \Longrightarrow a[r] = x$

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

Naming our return value is useful, as we can then refer to it in method post-conditions.

However, if we try to compile the method as above, we get:

Error: index out of range

The error message is referring to `a[r]` in the post-condition; namely, Dafny does not know that $0 \leq r$.

So, we add `ensures 0 <= r;` to our post-conditions,

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

and it compiles. This means that Dafny has been able to establish automatically that this postcondition is satisfied without further help from the programmer.

Note that the order in which we write the assertions matters. For example, the following

```
method Find<T(==)>(x: T, a: array<T>)  returns (r : int)
  requires a != null;
  ensures r < a.Length ==> a[r] == x;
  ensures 0 <=r;
{  ….  }
```

does not compile, and throws: Error: index out of range

3. Verification

3.1 Partial Correctness and Loop Invariants

In the general case, when verifying functional properties of methods which contain loops, we will need to supply invariants. To demonstrate this, we consider another post-condition for `Find`:

$$\forall j. 0 \leq j < r \Longrightarrow a[j] \neq x$$

We add this postcondition to our code:

```
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)
  {
    i := i + 1;
  }
  return i;
}
```

Unfortunately, Dafny can verify neither $r \leq a.Length$ nor $\forall j. 0 \leq j < r \Longrightarrow a[j] \neq x$.
Therefore, it creates the error message:

Input(..) Error BP5003 A postcondition might not hold on this return path.

We need to give more help to Dafny and specifically, we need to add invariants to the loop, as already discussed in the course.

The loop invariants are

$$i \leq a.Length \quad \text{and} \quad \forall j. 0 \leq j < i \Longrightarrow a[j] \neq x$$

```
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;
}
```

We add invariants using `invariant`.

This code compiles successfully and hence all post-conditions and invariants are verified. In particular, Dafny also was able to prove termination of the loop, as it was able to automatically infer the loop variant.

3.2 Assertions

We can use assertions, to check whether Dafny can prove that certain properties hold at specific points in program execution.

```
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;
  {
       assert a[i] != x;
       i := i + 1;
       assert a[i-1] != x;
  }
  r := i;
}
```

Assertions describe the state before and after the assignment to i

The code above compiles, which means that the assertions hold.

However, not all assertions are successful. For example << PUT AN INTERESTING EXAMPLE HERE>>>

3.3 Termination and decreases Clauses

Remember that all methods in Dafny must be proven to terminate. Therefore, loops also need to be provan to terminate. To do this, Dafny tries to infer the loop variant, and tries to prove that the variant decreases for every loop iteration, and that it is bouded. In the `Find` method as given earlier, Dafny could infer the variant “`a.Length - i`”, and therefore could automatically prove termination of the loop.

In the cases where Dafny cannot infer the loop variant, we supply it ourselves using the `decreases` keyword. In the example below, we have McCarthy’s 91 function. It returns 91 for any input $x < 100$, and $x - 10$ for any higher input.

```
method McCarthy91(x: nat)  returns (n: nat)
{
  n := x;
  var i := 1;
  while (true)
    decreases (10 * i - n) + 90, n;
  {
    if (n > 100) {
      if (i == 1) {
        return n - 10;
      }
      else {
        n := n - 10;
        i := i - 1;
      }
    }
    else {
      n := n + 11;
      i := i + 1;
    }
  }
}
```

The loop variant is the pair
((10*i-n)+90, n).
The ordering on these pairs is the lexicographic ordering.

Note that Dafny implicitly choses the lexicographic ordering on the pairs.

4. Exercises

You can now proceed to do the exercises titled `Exercises_Arithm_Terms_Loops` and `Exercises_Max_Product`.

Chapter 5: Incremental Development of imperative programs

In this chapter we shall study how to develop imperative Dafny programs Dafny in an incremental manner. The approach mirrors that of functional programs as discussed in chapter 3. We will use pre-conditions and post-conditions to *specify* our code, `assume` statements to allow ourselves to skip some of the details to be added for later, and assert `statements` to ask Dafny to confirm that some expected properties do, indeed, hold. We will also see how to inline method calls.

We will use a similar example as we did for functional programming, namely sorting. And because we have imperative programming, we will be sorting an array.

The Dafny definitions for the functions and predicates from this chapter can be found in the file `Arrays_BubbleSort.dfy`.

Step 1: Preliminaries: predicates and the sequences in Dafny

We will first define what it means for an array a to be sorted:

$$Sorted(a) \quad \Leftrightarrow \quad \forall i, j{:}\, nat.\, 0 \le i < j < a.Length \;\rightarrow\; a[i] \le a[j]$$

$$SortedBetween(a, from, to)$$
$$\Leftrightarrow$$
$$SortedBetween(a, 0, a.Length).$$

A definition of Sorted in Dafny is given below. Notice the `reads` clause, which indicates that the predicate depends on the contents of the whole array.

```
predicate sorted_between(a: array<int>, from:int, to:int)
      requires a!= null;
      reads a;
{
      reads forall i,j :: from<=i<j<to && 0<=i<j<a.Length ==> a[i]<=a[j]
}

predicate sorted(a: array<int>)
      requires a!= null;
      reads a;
{
      sorted_between(a,0,a.Length)
```

Must declare that the function depends on contents of the whole array.

In order to define whether two arrays are permutations of each other, we use the concept of *sequences.* These are introduced in http://rise4fun.com/Dafny/tutorialcontent/Sequences in detail. In summary, a sequence s of type `seq<T>` is an ordered list. Sequences are immutable. The length of a sequence is written as `|s|`. The slice notation `s[i..k]`, where `0<=j<=k<|s|` has k-`j` elements, starting at the `i`-th element, and continuing until the k-1st element, unless the result is empty. Notice that Dafnyonly supports closed-open intervals, and therefore the Dafny notation `s[i..j]` corresponds to our course's notation `s[i..j)`. The notation `s[k..]` is a shorthand for `s[k..|s|]`, while `[t]` is a singleton `seq<T>`, where `t` is a value of type T, and + is the concatenation operator.

We now define what it means for an array to be a permutation of another array.

$$Count(a,m) \quad \Leftrightarrow \quad card\{\, i \mid a[i] = m \,\}$$

$$Perm(a,b) \quad \Leftrightarrow \quad \forall m.\, Count(a,m) = Count(b,m)$$

The Dafny definitions are as follows:

```
function count<T>(t:T, seq<T>): nat
{      if (|s| == 0)
       then 0
       else  if x == s[0]
             then 1 + count(x, s[1..])
             else count(x, s[1..])
}

predicate perm<T>(a: seq<T>, b: seq<T>)
{
  forall t :: count(t, a) == count(t, b)
}
```

To obtain a sequence out of an array a, we can use the slice notation, `a[..]`.

Step 2:. bubble Sort, Specification

We give the specification of the sorting function. Notice that in the pre-condition (requires clause) any mention of the formal parameters (here a) refers to the contents of the parameters upon entry to the method call, while in the pre-condition (ensures clause) any mention of the formal parameters refers to their contents of the parameters upon exit from the method call. In the ensures clause we indicate the value of a term t upon entry, through the notation old(t). Therefore, old(a) is the value of the array pointer upon entry, while old(a[..]) is the value of the contents of the array upon entry.

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

Must declare that it modifies the contents of the array.

Therefore, in the above, the ensures clause requires that upon exit from the function call, the contents of the array is a permutation of its contents upon entry, and moreover, that it will be sorted.

Step 3:. bubbleSort, sketching the body

We now take a first stab at the method body for bubbleSort . We can assume that we will be sorting the array gradually from left to right, while maintaining the property that the array is a permutation of the original array:

```
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[..]));
  }
}
```

We have not yet specified *how* we will sort the array..

Note that we have an assume clause in the method body. This means that the method cannot be compiled. Instead, we need to replace the assume clause by a piece of code which will provide an implementation for the guarantee given by the assume clause.

Step 4:. Introducing an auxiliary method

We now replace the `assume` clause by a call to a method which is supposed to achieve the difference that is required by each loop iteration. This is done by the method `pushToRight`:

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

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[..]);
```

We have not yet given a method body for `pushToRight`

The method as given above does not verify. In fact, Dafny gives the error message that it cannot prove that the invariant "`perm(old(a[..]),a[..])`" is preserved by the loop body. We therefore need to give some help to Dafny to make this proof. The logical argument that underpins that the loop body does, indeed, preserve the invariant is, that at the beginning of the loop body, and before the call to pushToRight, the value of `a[..]` is a permutation of `old(a[..])`, and, because of the specification of pushToRight, after the call, `a[..]` is a permutation of what `a[..]` was before the call. Therefore, because the permutation relation is transitive, we also have that at the end of the loop body, the value of `a[..]` is a permutation of the value of `old(a[..])`.

In order to express this argument, we need to a) introduce ghost variables for the value of a[..] at the start and at the end of the loop body, and b) prove a lemma that guarantees that permutation is a transitive relation. Note that ghost variables are indicated by the keyword `ghost`. They form part of the correctness argument and may appear in the call of ghost methods, but not in the normal calculations of a method.

The lemma that guarantees that permutation is a transitive relation is given here:

```
ghost method perm_trans<T>(a: seq<T>, b: seq<T>, c: seq<T>)
    requires perm(a,b) && perm(b,c);
    ensures  perm(a,c);
{    }
```

We now complete the argument for the preservation of the invariant within the loop body. We introduce ghost variables a_before and a_after to distinguish the contents of a[..] before and after the call of pushToRight.

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

The method as defined above now verifies in Dafny. We can also make a shorter version:

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

Step 5:. The body of method `pushToRight`

We now give a method body for `pushToRight`. This method, in its turn, calls method `swap(_,_,_)`, which swaps the contents of array a at indices `j-1` and `j`.

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

method swap<T>(a: array<T>, i: nat, j:nat)
   // swaps a[i] and a[j] in the array a
   requires a!=null && 0<=i<a.Length &&  0<=j<a.Length ;
   modifies a;
   ensures  swapped(old(a[..]),a[..],i,j);

predicate swapped<T>(a:seq<T>, b:seq<T>, i: nat, j:nat)
    requires |a|==|b|  && 0<=i<|a|  && 0<=j<|a|;
{ ( forall k:: 0<=k<|a|  &&  k!=i && k!=j ==> a[k]==b[k] )
              &&
  ( b[j]==a[i] )
              &&
  ( b[i]==a[j] )  }
```

Notice that Dafny can independently infer the variant, and we do not need to supply an explicit decreases clause.

However, Dafny cannot prove the correctness of method `pushToRight`, and in particular, it cannot prove that the method body preserves “`perm(old(a[..]),a[..]);`”. The argument here hinges on the fact that swapping elements in sequences creates permutations, as stated in lemma `swap_implies_perm` below. As in step 4, we need to introduce a ghost variable to “remember” the contents of `a[..]` before swapping and use this variable when calling the lemma.

The following version of the method body verifies in Dafny:

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

ghost method swap_implies_perm<T>(a:seq<T>, b:seq<T>, i: nat, j:nat)
     requires|a|==|b| && 0<=i<|a| && 0<=j<|a| && swapped(a,b,i,j);
     ensures perm(a,b);
```

Step 6:. The method body of swap

We now write the method body for swap, which verifies in Dafny without further need for justification:

```
method swap<T>(a: array<T>, i: nat, j:nat)
   // swaps a[i] and a[j] in the array a
   requires a!=null && 0<=i<a.Length &&  0<=j<a.Length ;
   modifies a;
   ensures  swapped(old(a[..]),a[..],i,j);
{
            var temp : T := a[i];
            a[i]:=a[j];
            a[j]:=temp;
}
```

Step 7:. Inlining the body of method pushToRight within method bubbleSort

We can now replace the method call to swap inside body of bubbleSort by its body, and we can replace the call to pushToRight inside the loop body within method bubbleSort, by its body. We will also inline the invariants accordingly:

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

  while i < a.Length
    invariant 0 < i <= a.Length;
    invariant perm(a[..], old(a[..])) && sortedBetween(a, 0, i);
    decreases a.Length - i;
  {  var j: nat := i;

     while j > 0 && a[j-1] > a[j]
        invariant 0 <= j <= i;
        invariant perm(a[..], old(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;
  }
 }
```

Exercises

You can now proceed to do the exercises titled Exercises_Insertion_Sort..