Dafny

An Automatic Program Verifier for Functional Correctness

Holzinger Jan-Michael

Seminar Formal Methods WS 2015/2016

18. November 2015

What is Dafny?

> *„Dafny is a programming language with built-in specification constructs. The Dafny static program verifier can be used to verify the functional correctness of programs.“*[8]

Among other programming languages, these and their concepts had big influence on Dafny:

What is Dafny?

> *„Dafny is a programming language with built-in specification constructs. The Dafny static program verifier can be used to verify the functional correctness of programs.“*[8]

Among other programming languages, these and their concepts had big influence on Dafny:

- Java, C# (classes, traits, syntax for functions)

What is Dafny?

> *„Dafny is a programming language with built-in specification constructs. The Dafny static program verifier can be used to verify the functional correctness of programs.“*[8]

Among other programming languages, these and their concepts had big influence on Dafny:

- Java, C# (classes, traits, syntax for functions)
- Eiffel (Pre- & Postconditions, Invariants, Asserts)

What is Dafny?

> *„Dafny is a programming language with built-in specification constructs. The Dafny static program verifier can be used to verify the functional correctness of programs.“*[8]

Among other programming languages, these and their concepts had big influence on Dafny:

- Java, C# (classes, traits, syntax for functions)
- Eiffel (Pre- & Postconditions, Invariants, Asserts)
- ML (inductive Datatypes)

What is Dafny?

> *„Dafny is a programming language with built-in specification constructs. The Dafny static program verifier can be used to verify the functional correctness of programs.“*[8]

Among other programming languages, these and their concepts had big influence on Dafny:

- Java, C# (classes, traits, syntax for functions)
- Eiffel (Pre- & Postconditions, Invariants, Asserts)
- ML (inductive Datatypes)
- ...

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

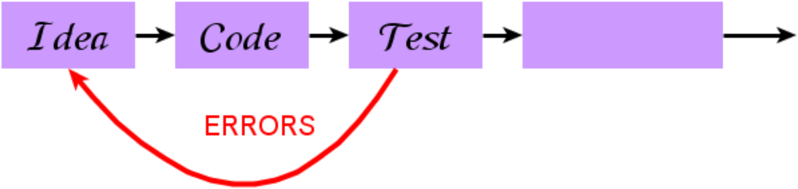
Why should one use Verification Tools?

Why should one use Verification Tools?

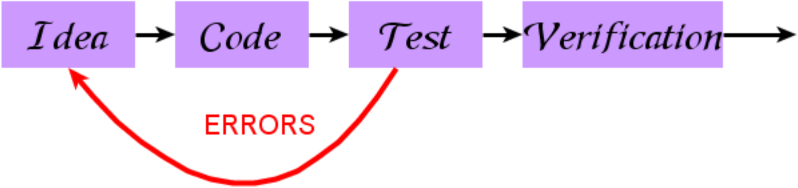
Why should one use Verification Tools?

Idea → *Code* → *Test* →

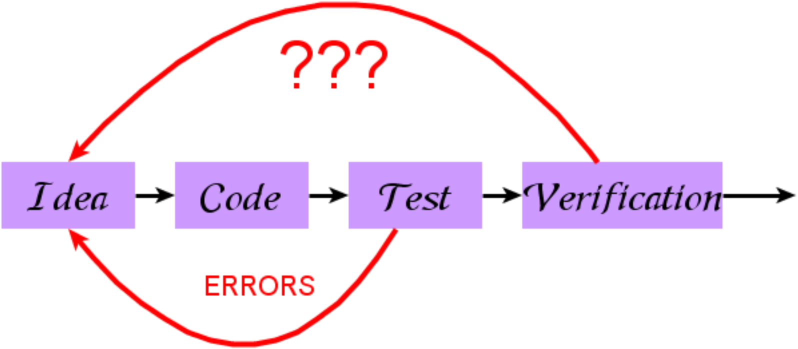
Why should one use Verification Tools?



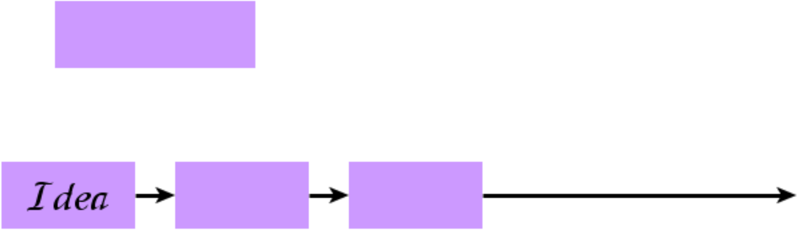
Why should one use Verification Tools?



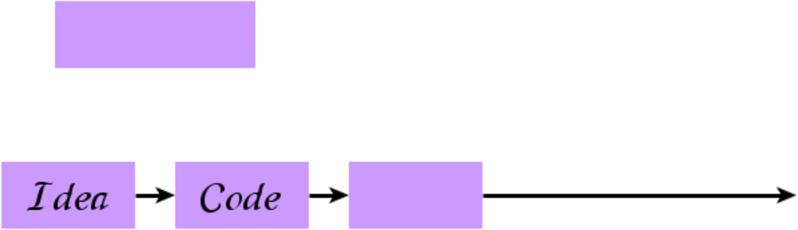
Why should one use Verification Tools?



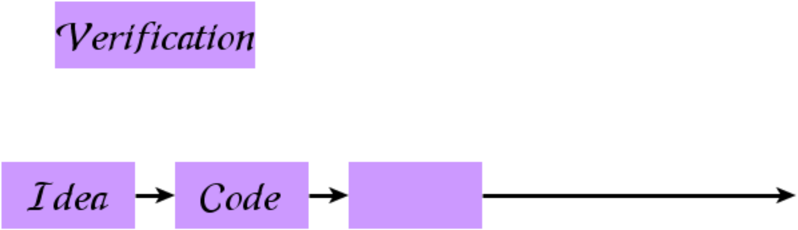
Why should one use Verification Tools?



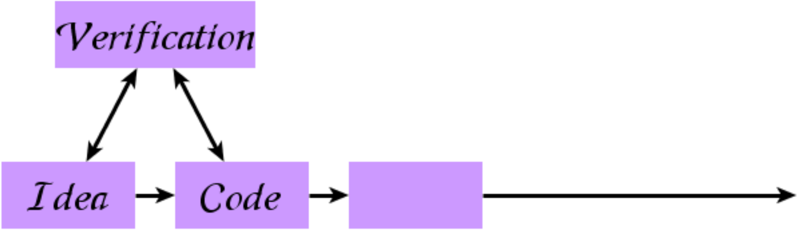
Why should one use Verification Tools?



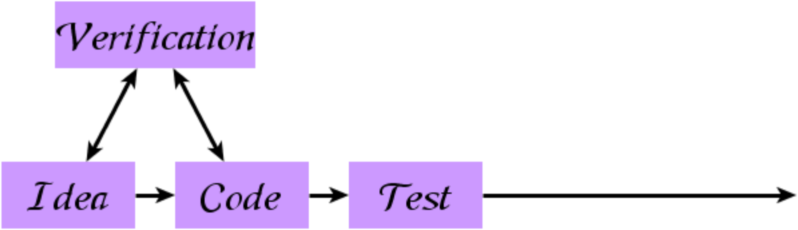
Why should one use Verification Tools?



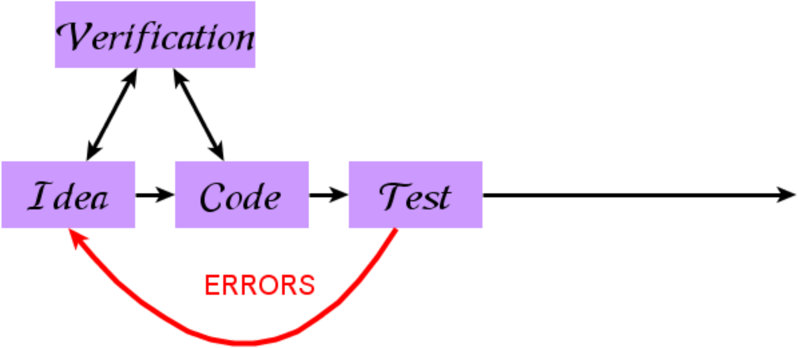
Why should one use Verification Tools?



Why should one use Verification Tools?



Why should one use Verification Tools?



Development

Dafny was developed by Microsoft Research (MSR) by K. Rustan M. Leino (shorter: Rustan Leino) and others[2] in 2008.
Founded in 1991, MSR researches among other subjects on the following:

- Algorithms
- Social Computing
- Software Development
- Hardware Development

Some well known experts as Michael Freedman (Mathematician, awarded with Fields-Medal) or Leslie Lamport work or had worked there.
Some developments of MSR are C# and the Windows Sidebar.[7]

Development

> *„Dafny started as a little language and verifier to experiment with a certain style of specifications (known as "dynamic frames") for programs that operate on a mutable heap. Since then, Dafny has become a full program verifier with both imperative (assignments, loops, classes, etc.) and functional (datatypes, co-datatypes, higher-order functions, etc.) programming constructs as well as proof authoring facilities (lemmas, proof calculations, refinement, etc.). It has been used for some systems projects, including ExpressOS [ASPLOS 2013], Ironclad Apps [OSDI 2014], and IronFleet [SOSP 2015], and it has been used in various forms of teaching at more than 30 universities.“ - Rustan Leino*

Development

I would like to call Dafny a living programming language, as there are still features that are planned to evolve, to be changed, added or left out completely.[5]

Development

I would like to call Dafny a living programming language, as there are still features that are planned to evolve, to be changed, added or left out completely.[5]

The verification process, (very) briefly summarized:

Development

I would like to call Dafny a living programming language, as there are still features that are planned to evolve, to be changed, added or left out completely.[5]

The verification process, (very) briefly summarized:

- ▷ Dafny translates code into Boogie (intermediate verification language)

Development

I would like to call Dafny a living programming language, as there are still features that are planned to evolve, to be changed, added or left out completely.[5]

The verification process, (very) briefly summarized:

- Dafny translates code into Boogie (intermediate verification language)
- The Boogie code is then verified using the SMT proofer Z3

Rustan Leino

Principal Researcher, Microsoft Research

Visiting Professor, Department of Computing, Imperial College London

My research has centered around tools for software engineers, including programming languages, programming tools, programming systems, program verification, and program design technologies.

Rustan Leino

Principal Researcher, Microsoft Research

Visiting Professor, Department of Computing, Imperial College London

My research has centered around tools for software engineers, including programming languages, programming tools, programming systems, program verification, and program design technologies.

▶ Verification Corner:

`https://www.youtube.com/channel/UCP2eLEql4tROYmIYm5mA27A`

Rustan Leino

Principal Researcher, Microsoft Research

Visiting Professor, Department of Computing, Imperial College London

My research has centered around tools for software engineers, including programming languages, programming tools, programming systems, program verification, and program design technologies.

Verification Corner:
`https://www.youtube.com/channel/UCP2eLEql4tROYmIYm5mA27A`
Homepage:
`http://research.microsoft.com/en-us/um/people/leino/`

Where can I get Dafny?

Dafny can be tried out at `http://rise4fun.com/Dafny/`,where one also can find a very helpful tutorial `http://rise4fun.com/Dafny/tutorial` and examples.
If you want to get more serious, you can download it from `http://dafny.codeplex.com/` and run it from the command line, or, what I actually recommend, running it in Microsoft Visual Studio (2012). I will give a demo of that at the end of this presentation.

Table of Contents

Postcondition and Precondition

Typically, every algorithm has specific Input and Output Conditions! These concept is realized in Dafny by using requires and ensures.

Skeleton of a Method

```
method Algorithm1(x:int,y:real) returns (z:nat)
```

Postcondition and Precondition

Typically, every algorithm has specific Input and Output Conditions! These concept is realized in Dafny by using requires and ensures.

Skeleton of a Method

```
method Algorithm1(x:int,y:real) returns (z:nat)
//Comment
```

Postcondition and Precondition

Typically, every algorithm has specific Input and Output Conditions! These concept is realized in Dafny by using requires and ensures.

Skeleton of a Method

```
method Algorithm1(x:int,y:real) returns (z:nat)
//Comment
  requires <Precondition1>
  requires <Precondition2>;
  ...
```

Postcondition and Precondition

Typically, every algorithm has specific Input and Output Conditions! These concept is realized in Dafny by using requires and ensures.

Skeleton of a Method

```
method Algorithm1(x:int,y:real) returns (z:nat)
//Comment
  requires <Precondition1>
  requires <Precondition2>;
  ...
  ensures <Postcondition1>
  ensures <Postcondition2>;
  ...
```

Postcondition and Precondition

Typically, every algorithm has specific Input and Output Conditions! These concept is realized in Dafny by using requires and ensures.

Skeleton of a Method

```
method Algorithm1(x:int,y:real) returns (z:nat)
//Comment
  requires <Precondition1>
  requires <Precondition2>;
  ...
  ensures <Postcondition1>
  ensures <Postcondition2>;
  ...
{
  ...
}
```

Postcondition and Precondition

Typically, every algorithm has specific Input and Output Conditions! These concept is realized in Dafny by using requires and ensures.

Skeleton of a Method

```
method Algorithm1(x:int,y:real) returns (z:nat)
//Comment
  requires <Precondition1>  ← Callers responsibility
  requires <Precondition2>;
  ...
  ensures <Postcondition1>
  ensures <Postcondition2>;
  ...
{
  ...
}
```

Postcondition and Precondition

Typically, every algorithm has specific Input and Output Conditions! These concept is realized in Dafny by using requires and ensures.

Skeleton of a Method

```
method Algorithm1(x:int,y:real) returns (z:nat)
//Comment
  requires <Precondition1>  ← Callers responsibility
  requires <Precondition2>;
  ...
  ensures <Postcondition1>  ← Programmers responsibility
  ensures <Postcondition2>;
  ...
{
  ...
}
```

Postcondition and Precondition

Typically, every algorithm has specific Input and Output Conditions! These concept is realized in Dafny by using requires and ensures.

Skeleton of a Method

```
method Algorithm1(x:int,y:real) returns (z:nat)
//Comment
  requires <Precondition1>  ← Callers responsibility
  requires <Precondition2>;
  ...
  ensures <Postcondition1>  ← Programmers responsibility
  ensures <Postcondition2>;
  ...
{
  ...
}
```

Asserts

Whereas requires and ensures are verified at the beginning resp. at the end of a Method/Function, it is sometimes needed/useful to check if a certain assertion holds at a certain place or time in the programm.

For this purpose, one can place

Asserts

Whereas requires and ensures are verified at the beginning resp. at the end of a Method/Function, it is sometimes needed/useful to check if a certain assertion holds at a certain place or time in the programm.
For this purpose, one can place

```
method ComputeTesseract(x:int) returns (z:int)
  requires -100<=x<=100
  ensures z>=0
{
  z:=x*x*x*x;
  assert z>=x;
}
```

Asserts

Whereas requires and ensures are verified at the beginning resp. at the end of a Method/Function, it is sometimes needed/useful to check if a certain assertion holds at a certain place or time in the programm.
For this purpose, one can place

```
method ComputeTesseract(x:int) returns (z:int)
  requires -100<=x<=100
  ensures z>=0
{
  z:=x*x*x*x;
  assert z>=x;
}
```

everywhere in the Function or Method body. The assertion must then hold where it is placed.

Asserts

Whereas requires and ensures are verified at the beginning resp. at the end of a Method/Function, it is sometimes needed/useful to check if a certain assertion holds at a certain place or time in the programm.
For this purpose, one can place

```
method ComputeTesseract(x:int) returns (z:int)
  requires -100<=x<=100
  ensures z>=0
{
  z:=x*x*x*x;
  assert z>=x;    ← The semikolon here is needed.
}
```

everywhere in the Function or Method body. The assertion must then hold where it is placed.

Functions

Skeleton of a Function

```
function square(x:int): nat
```

Functions

Skeleton of a Function

```
function square(x:int): nat
//Returns the square of the Integer value
ensures x*x>=0
```

Functions

Skeleton of a Function

```
function square(x:int): nat
//Returns the square of the Integer value
ensures x*x>=0
{
  x*x
}
```

Functions

Skeleton of a Function

```
function square(x:int): nat
//Returns the square of the Integer value
ensures x*x>=0
{
  x*x
}
```

Sometimes, you might see predicate used instead of function, and the „return type“ missing, as a Predicate is a Function that returns bool. So somehow this is just a shorter way of writing a Function, that corresponds to an attribute. Also, sometimes you might see function method used, that just means that this is a Function as well as a Method, so you can use the best of both worlds.

Functions

On first view, one might think, that Functions are very limited, compared to Methods. Functions can only hold one statement, the code of a Function is not compiled and they can only be used in annotations.

Functions

On first view, one might think, that Functions are very limited, compared to Methods. Functions can only hold one statement, the code of a Function is not compiled and they can only be used in annotations.

```
method Pythagoras(a:int,b:int) returns(c:int)

{

}
```

Functions

On first view, one might think, that Functions are very limited, compared to Methods. Functions can only hold one statement, the code of a Function is not compiled and they can only be used in annotations.

```
method Pythagoras(a:int,b:int) returns(c:int)


{
  var c:=square(a)+square(b);
}
```

Functions

On first view, one might think, that Functions are very limited, compared to Methods. Functions can only hold one statement, the code of a Function is not compiled and they can only be used in annotations.

```
method Pythagoras(a:int,b:int) returns(c:int)


{
  var c:=square(a)+square(b);    ← This will not work
}
```

Functions

On first view, one might think, that Functions are very limited, compared to Methods. Functions can only hold one statement, the code of a Function is not compiled and they can only be used in annotations. But, the power of Function comes from the fact that they can be used in annotations, while Methods cannot.

```
method Pythagoras(a:int,b:int) returns(c:int)


{
  var c:=square(a)+square(b);    ← This will not work
}
```

Functions

On first view, one might think, that Functions are very limited, compared to Methods. Functions can only hold one statement, the code of a Function is not compiled and they can only be used in annotations. But, the power of Function comes from the fact that they can be used in annotations, while Methods cannot.

```
method Pythagoras(a:int,b:int) returns(c:int)
  ensures square(a)>=0
{
  var c:=square(a)+square(b);   ← This will not work
}
```

Functions

On first view, one might think, that Functions are very limited, compared to Methods. Functions can only hold one statement, the code of a Function is not compiled and they can only be used in annotations.
But, the power of Function comes from the fact that they can be used in annotations, while Methods cannot.

```
method Pythagoras(a:int,b:int) returns(c:int)
  ensures square(a)>=0  ← This will work
{
  var c:=square(a)+square(b);    ← This will not work
}
```

Functions

On first view, one might think, that Functions are very limited, compared to Methods. Functions can only hold one statement, the code of a Function is not compiled and they can only be used in annotations.
But, the power of Function comes from the fact that they can be used in annotations, while Methods cannot.

```
method Pythagoras(a:int,b:int) returns(c:int)
  ensures square(a)>=0  ← This will work
{
  var c:=square(a)+square(b);    ← This will not work
}
```

This means, one can use Functions to prove correctness of a Method.

Functions

On first view, one might think, that Functions are very limited, compared to Methods. Functions can only hold one statement, the code of a Function is not compiled and they can only be used in annotations.
But, the power of Function comes from the fact that they can be used in annotations, while Methods cannot.

```
method Pythagoras(a:int,b:int) returns(c:int)


{


}
```

This means, one can use Functions to prove correctness of a Method.

Functions

On first view, one might think, that Functions are very limited, compared to Methods. Functions can only hold one statement, the code of a Function is not compiled and they can only be used in annotations.
But, the power of Function comes from the fact that they can be used in annotations, while Methods cannot.

```
method Pythagoras(a:int,b:int) returns(c:int)
  ensures c==square(a)+square(b)
{
  c:=a*a+b*b;
}
```

This means, one can use Functions to prove correctness of a Method.

Functions

So for the Tesseract Method we can use:

Functions

So for the Tesseract Method we can use:

```
method ComputeTesseract(x:int) returns (z:int)
  requires -100<=x<=100
  ensures z>=0

{
  z:=x*x*x*x;
  assert z>=x;
}
```

Functions

So for the Tesseract Method we can use:

```
method ComputeTesseract(x:int) returns (z:int)
  requires -100<=x<=100
  ensures z>=0
  ensures z==square(square(x))
{
  z:=x*x*x*x;
  assert z>=x;
}
```

Framing

For every variable, or object, that is allocated on the heap or accessed through references, Dafny needs to know what parts of the memory are accessed or even modified by a Function or Method.

Framing

For every variable, or object, that is allocated on the heap or accessed through references, Dafny needs to know what parts of the memory are accessed or even modified by a Function or Method. A Method is allowed to read everything, but it has to be specified with the modifies annotation what heap allocated memory areas it is allowed to change.

Framing

For every variable, or object, that is allocated on the heap or accessed through references, Dafny needs to know what parts of the memory are accessed or even modified by a Function or Method. A Method is allowed to read everything, but it has to be specified with the modifies annotation what heap allocated memory areas it is allowed to change.
A Function must be allowed to access heap allocated memory with the reads annotation. By definition, a Function is not allowed to modify anything, so we don't need to specify that.

Framing

For every variable, or object, that is allocated on the heap or accessed through references, Dafny needs to know what parts of the memory are accessed or even modified by a Function or Method. A Method is allowed to read everything, but it has to be specified with the modifies annotation what heap allocated memory areas it is allowed to change.
A Function must be allowed to access heap allocated memory with the reads annotation. By definition, a Function is not allowed to modify anything, so we don't need to specify that.
Bear in mind, that local variables, sets, sequences and multisets - that are treated like local variables or integers - cannot/need not to be mentioned in such annotations.

Framing

For every variable, or object, that is allocated on the heap or accessed through references, Dafny needs to know what parts of the memory are accessed or even modified by a Function or Method. A Method is allowed to read everything, but it has to be specified with the modifies annotation what heap allocated memory areas it is allowed to change.
A Function must be allowed to access heap allocated memory with the reads annotation. By definition, a Function is not allowed to modify anything, so we don't need to specify that.
Bear in mind, that local variables, sets, sequences and multisets - that are treated like local variables or integers - cannot/need not to be mentioned in such annotations.
I will give some examples later.

Loop Invariants

Dafny faces a real problem when it reaches the beginning of a loop or a recursion. It has to prove correctness, that means to „go all possible ways“, of a loop. One can easily imagine that this is impossible in general. What it does to solve this problem is, to view the loop as a black box. It is then the programmers duty to provide loop invariants. Dafny then tries to proof that the invariant holds at entering and at every execution of the loop.

Loop Invariants

Dafny faces a real problem when it reaches the beginning of a loop or a recursion. It has to prove correctness, that means to „go all possible ways“, of a loop. One can easily imagine that this is impossible in general. What it does to solve this problem is, to view the loop as a black box. It is then the programmers duty to provide loop invariants. Dafny then tries to proof that the invariant holds at entering and at every execution of the loop.

```
var i: int :=0;
while i<n
  invariant 0<=i<=n
{
  i:=i+1;
}
```

Loop Invariants

Dafny faces a real problem when it reaches the beginning of a loop or a recursion. It has to prove correctness, that means to „go all possible ways“, of a loop. One can easily imagine that this is impossible in general. What it does to solve this problem is, to view the loop as a black box. It is then the programmers duty to provide loop invariants. Dafny then tries to proof that the invariant holds at entering and at every execution of the loop.

```
var i: int :=0;
while i<n
  invariant 0<=i<=n
{
  i:=i+1;
}
```

```
var i, a, b := 1,0,1;
while i<n
  invariant 0<=i<=n
  invariant a==fib(i-1)
  invariant b==fib(i)
{
  a, b:=b,a+b;
  i:=i+1;
}
```

Loop Invariants

To find the right Invariants is very difficult,

Loop Invariants

To find the right Invariants is very difficult,a good tactics is to work from the postcondition upwards.

Loop Invariants

To find the right Invariants is very difficult,a good tactics is to work from the postcondition upwards.
As an example of the amount of invariants needed should the Schorr-Waite Algorithm (Written in Dafny)[3] serve.

Loop Invariants

To find the right Invariants is very difficult,a good tactics is to work from the postcondition upwards.
As an example of the amount of invariants needed should the Schorr-Waite Algorithm (Written in Dafny)[3] serve.

> about $\frac{1}{4}$ of the whole code (including comment),

Loop Invariants

To find the right Invariants is very difficult,a good tactics is to work from the postcondition upwards.
As an example of the amount of invariants needed should the Schorr-Waite Algorithm (Written in Dafny)[3] serve.

about $\frac{1}{4}$ of the whole code (including comment),

about $\frac{1}{3}$ of the „core Method“ and

Loop Invariants

To find the right Invariants is very difficult,a good tactics is to work from the postcondition upwards.
As an example of the amount of invariants needed should the Schorr-Waite Algorithm (Written in Dafny)[3] serve.

about $\frac{1}{4}$ of the whole code (including comment),
about $\frac{1}{3}$ of the „core Method“ and
more than $\frac{1}{2}$ of the core loop are code lines for invariants/decreases.

Termination of Loops and Recursions

In some (easy) cases, Dafny can proof termination of loops and recursions without any help. But after programming a few easy recursive Functions or Methods, one will see that Dafny can really prove termination of very little recursions and loops. The programmer then must help Dafny with providing the decreases annotation.

Termination of Loops and Recursions

In some (easy) cases, Dafny can proof termination of loops and recursions without any help. But after programming a few easy recursive Functions or Methods, one will see that Dafny can really prove termination of very little recursions and loops. The programmer then must help Dafny with providing the decreases annotation.

```
while 0 <i
  invariant 0<=i
  decreases i
{
  i:=i-1;
}
```

Termination of Loops and Recursions

In some (easy) cases, Dafny can proof termination of loops and recursions without any help. But after programming a few easy recursive Functions or Methods, one will see that Dafny can really prove termination of very little recursions and loops. The programmer then must help Dafny with providing the decreases annotation.

```
while 0 <i
  invariant 0<=i
  decreases i
{
  i:=i-1;
}
```

While most of the time this will be an integer, every expression, that:

1. Gets smaller and
2. Is bounded

can be used.

Table of Contents

Boolean Operators

Besides the already presented Numeric Types, Dafny also provides Boolean variables. A Boolean value can either be true or false, which are also the corresponding literals.
With Boolean values and expressions one can use the following operators:

Boolean Operators

Besides the already presented Numeric Types, Dafny also provides Boolean variables. A Boolean value can either be true or false, which are also the corresponding literals.

With Boolean values and expressions one can use the following operators:

```
a && b; a || b; !a;
```

Boolean Operators

Besides the already presented Numeric Types, Dafny also provides Boolean variables. A Boolean value can either be true or false, which are also the corresponding literals.

With Boolean values and expressions one can use the following operators:

```
    a && b; a || b; !a;   // AND (conjunction), OR (disjunction), NOT (negation)
```

Boolean Operators

Besides the already presented Numeric Types, Dafny also provides Boolean variables. A Boolean value can either be true or false, which are also the corresponding literals.
With Boolean values and expressions one can use the following operators:

```
a && b; a || b; !a;   // AND (conjunction), OR (disjunction), NOT (negation)
0 <= x ==> x == y   // IMPLIES, also possible is <==
```

Boolean Operators

Besides the already presented Numeric Types, Dafny also provides Boolean variables. A Boolean value can either be true or false, which are also the corresponding literals.
With Boolean values and expressions one can use the following operators:

```
a && b; a || b; !a;   // AND (conjunction), OR (disjunction), NOT (negation)
0 <= x ==> x == y   // IMPLIES, also possible is <==
(x > 0) && (y > 0) <==> x == y   // IFF, „==> & <=="
```

Boolean Operators

Besides the already presented Numeric Types, Dafny also provides Boolean variables. A Boolean value can either be true or false, which are also the corresponding literals.
With Boolean values and expressions one can use the following operators:

```
a && b; a || b; !a;   // AND (conjunction), OR (disjunction), NOT (negation)
0 <= x ==> x == y   // IMPLIES, also possible is <==
(x > 0) && (y > 0) <==> x == y   // IFF, „==> & <=="
```

Of course one can also test equality (==) and inequality (!=) with boolean values and expressions.

Quantifiers

Like every modern program language, Dafny also allows the use of Arrays, and many other Collection Types, such as Sets, Multisets, Sequences and so on. Using such concepts, one certainly will sooner or later face the problem that one want to implement something like

Quantifiers

Like every modern program language, Dafny also allows the use of Arrays, and many other Collection Types, such as Sets, Multisets, Sequences and so on. Using such concepts, one certainly will sooner or later face the problem that one want to implement something like

- „All elements of the set should have a property p“

Quantifiers

Like every modern program language, Dafny also allows the use of Arrays, and many other Collection Types, such as Sets, Multisets, Sequences and so on. Using such concepts, one certainly will sooner or later face the problem that one want to implement something like

- „All elements of the set should have a property p“
- „No index of the Array is equal to zero“,

Quantifiers

Like every modern program language, Dafny also allows the use of Arrays, and many other Collection Types, such as Sets, Multisets, Sequences and so on. Using such concepts, one certainly will sooner or later face the problem that one want to implement something like

- „All elements of the set should have a property p“
- „No index of the Array is equal to zero“,
- or even more abstract, one might want something like „There is e1 and e2 in a set, such that x+e1=x and x*e2=x, for every x in the set“.

Quantifiers

Like every modern program language, Dafny also allows the use of Arrays, and many other Collection Types, such as Sets, Multisets, Sequences and so on. Using such concepts, one certainly will sooner or later face the problem that one want to implement something like

- „All elements of the set should have a property p“
- „No index of the Array is equal to zero“,
- or even more abstract, one might want something like „There is e1 and e2 in a set, such that x+e1=x and x*e2=x, for every x in the set“.

Dafny allows even such more abstract constructs, using quantifiers.

Quantifiers

Like every modern program language, Dafny also allows the use of Arrays, and many other Collection Types, such as Sets, Multisets, Sequences and so on. Using such concepts, one certainly will sooner or later face the problem that one want to implement something like

- „All elements of the set should have a property p“
- „No index of the Array is equal to zero“,
- or even more abstract, one might want something like „There is e1 and e2 in a set, such that x+e1=x and x*e2=x, for every x in the set“.

Dafny allows even such more abstract constructs, using quantifiers.

```
predicate haszero(a:set<int>)
{
  exists j :: j in a && (forall i :: i in a ==> j + i == i)
}
```

Sets and Multisets

The main difference between a Set and a Multiset is just, that a set does not count how often an element is contained, a Multiset does. So I will just give a few examples on how to specify and calculate with sets. (Note that unlike Arrays, Sets are not allocated on the heap)

Sets and Multisets

The main difference between a Set and a Multiset is just, that a set does not count how often an element is contained, a Multiset does. So I will just give a few examples on how to specify and calculate with sets. (Note that unlike Arrays, Sets are not allocated on the heap)

```
    var empty := {}; //Empty Set
```

Sets and Multisets

The main difference between a Set and a Multiset is just, that a set does not count how often an element is contained, a Multiset does. So I will just give a few examples on how to specify and calculate with sets. (Note that unlike Arrays, Sets are not allocated on the heap)

```
    var empty := {}; //Empty Set
    var ftp, stp := {2, 3}, {5, 7};
```

Sets and Multisets

The main difference between a Set and a Multiset is just, that a set does not count how often an element is contained, a Multiset does. So I will just give a few examples on how to specify and calculate with sets. (Note that unlike Arrays, Sets are not allocated on the heap)

```
var empty := {}; //Empty Set
var ftp, stp := {2, 3}, {5, 7};
var unite := ftp + stp; // {2,3,4,5}
```

Sets and Multisets

The main difference between a Set and a Multiset is just, that a set does not count how often an element is contained, a Multiset does. So I will just give a few examples on how to specify and calculate with sets. (Note that unlike Arrays, Sets are not allocated on the heap)

```
var empty := {}; //Empty Set
var ftp, stp := {2, 3}, {5, 7};
var unite := ftp + stp; // {2,3,4,5}
assert (stp * ftp == {}) && (|ftp| == 2); // This Assertion holds
```

Sets and Multisets

The main difference between a Set and a Multiset is just, that a set does not count how often an element is contained, a Multiset does. So I will just give a few examples on how to specify and calculate with sets. (Note that unlike Arrays, Sets are not allocated on the heap)

```
var empty := {}; //Empty Set
var ftp, stp := {2, 3}, {5, 7};
var unite := ftp + stp; // {2,3,4,5}
assert (stp * ftp == {}) && (|ftp| == 2); // This Assertion holds
var diff := ftp - {2}; // {3}
```

Sets and Multisets

The main difference between a Set and a Multiset is just, that a set does not count how often an element is contained, a Multiset does. So I will just give a few examples on how to specify and calculate with sets. (Note that unlike Arrays, Sets are not allocated on the heap)

```
var empty := {}; //Empty Set
var ftp, stp := {2, 3}, {5, 7};
var unite := ftp + stp; // {2,3,4,5}
assert (stp * ftp == {}) && (|ftp| == 2); // This Assertion holds
var diff := ftp - {2}; // {3}
```

U can use $\underbrace{<=,<,>,=>}_{\text{(proper) subset}}$ and $\underbrace{==,!=,!!}_{\text{(in)equality, disjointness}}$ as well as $\underbrace{\texttt{in},!\texttt{in}}_{\text{element membership}}$

Sets and Multisets

The main difference between a Set and a Multiset is just, that a set does not count how often an element is contained, a Multiset does. So I will just give a few examples on how to specify and calculate with sets. (Note that unlike Arrays, Sets are not allocated on the heap)

```
var empty := {}; //Empty Set
var ftp, stp := {2, 3}, {5, 7};
var unite := ftp + stp; // {2,3,4,5}
assert (stp * ftp == {}) && (|ftp| == 2); // This Assertion holds
var diff := ftp - {2}; // {3}
```

U can use $\underbrace{<=, <, >, =>}_{\text{(proper) subset}}$ and $\underbrace{==, != , !!}_{\text{(in)equality, disjointness}}$ as well as $\underbrace{\texttt{in}, !\texttt{in}}_{\text{element membership}}$

Another, really nice, way to generate sets is using the following construct:

Sets and Multisets

The main difference between a Set and a Multiset is just, that a set does not count how often an element is contained, a Multiset does. So I will just give a few examples on how to specify and calculate with sets. (Note that unlike Arrays, Sets are not allocated on the heap)

```
var empty := {}; //Empty Set
var ftp, stp := {2, 3}, {5, 7};
var unite := ftp + stp; // {2,3,4,5}
assert (stp * ftp == {}) && (|ftp| == 2); // This Assertion holds
var diff := ftp - {2}; // {3}
```

$$\texttt{U can use } \underbrace{<=,<,>,=>}_{\text{(proper) subset}} \texttt{ and } \underbrace{==,!=,!!}_{\text{(in)equality, disjointness}} \texttt{ as well as } \underbrace{\texttt{in},!\texttt{in}}_{\text{element membership}}$$

Another, really nice, way to generate sets is using the following construct:

$$\texttt{var s := set x: nat} \mid \texttt{x} < 100 :: \underbrace{\texttt{isprime}}_{\text{must be a function method}} \texttt{(x) == true;}$$

Sequences

Also Sequences, that represent ordered lists, are not allocated on the heap, so they are also immutable, once created. Some examples on handling and slicing Sequences:

Sequences

Also Sequences, that represent ordered lists, are not allocated on the heap, so they are also immutable, once created. Some examples on handling and slicing Sequences:

```
var empty := [ ]; //Empty Sequence
```

Sequences

Also Sequences, that represent ordered lists, are not allocated on the heap, so they are also immutable, once created. Some examples on handling and slicing Sequences:

```
var empty := [ ]; //Empty Sequence
var s := [2, 3, 5, 7, 11, 13, 17, 19];
```

Sequences

Also Sequences, that represent ordered lists, are not allocated on the heap, so they are also immutable, once created. Some examples on handling and slicing Sequences:

```
var empty := [ ]; //Empty Sequence
var s := [2, 3, 5, 7, 11, 13, 17, 19];
s[|s|-1 .. |s|];          // [19],
```

Sequences

Also Sequences, that represent ordered lists, are not allocated on the heap, so they are also immutable, once created. Some examples on handling and slicing Sequences:

```
var empty := [ ]; //Empty Sequence
var s := [2, 3, 5, 7, 11, 13, 17, 19];
s[|s|-1 .. |s|]; s[|s|-1]; // [19], 19
```

Sequences

Also Sequences, that represent ordered lists, are not allocated on the heap, so they are also immutable, once created. Some examples on handling and slicing Sequences:

```
var empty := [ ]; //Empty Sequence
var s := [2, 3, 5, 7, 11, 13, 17, 19];
s[|s|-1 .. |s|]; s[|s|-1]; // [19], 19
s[1 .. ];                  // [2, 3, 5, 7, 11, 13, 17],
```

Sequences

Also Sequences, that represent ordered lists, are not allocated on the heap, so they are also immutable, once created. Some examples on handling and slicing Sequences:

```
var empty := [ ]; //Empty Sequence
var s := [2, 3, 5, 7, 11, 13, 17, 19];
s[|s|-1 .. |s|]; s[|s|-1]; // [19], 19
s[1 .. ]; s[ .. |s|-1]; // [2, 3, 5, 7, 11, 13, 17], [3, 5, 7, 11, 13, 17,
19]
```

Sequences

Also Sequences, that represent ordered lists, are not allocated on the heap, so they are also immutable, once created. Some examples on handling and slicing Sequences:

```
var empty := [ ]; //Empty Sequence
var s := [2, 3, 5, 7, 11, 13, 17, 19];
s[|s|-1 .. |s|]; s[|s|-1]; // [19], 19
s[1 .. ]; s[ .. |s|-1]; // [2, 3, 5, 7, 11, 13, 17], [3, 5, 7, 11, 13, 17,
19]
assert s == s[0 .. ] == s[ .. |s|] == s[0 .. |s|] == s[ .. ] // This
Assertion holds
```

Sequences

Also Sequences, that represent ordered lists, are not allocated on the heap, so they are also immutable, once created. Some examples on handling and slicing Sequences:

```
var empty := [ ]; //Empty Sequence
var s := [2, 3, 5, 7, 11, 13, 17, 19];
s[|s|-1 .. |s|]; s[|s|-1]; // [19], 19
s[1 .. ]; s[ .. |s|-1]; // [2, 3, 5, 7, 11, 13, 17], [3, 5, 7, 11, 13, 17,
19]
assert s == s[0 .. ] == s[ .. |s|] == s[0 .. |s|] == s[ .. ] // This
Assertion holds
```

The reverse operation of slicing is concatenateing.

Sequences

Also Sequences, that represent ordered lists, are not allocated on the heap, so they are also immutable, once created. Some examples on handling and slicing Sequences:

```
var empty := [ ]; //Empty Sequence
var s := [2, 3, 5, 7, 11, 13, 17, 19];
s[|s|-1 .. |s|]; s[|s|-1]; // [19], 19
s[1 .. ]; s[ .. |s|-1]; // [2, 3, 5, 7, 11, 13, 17], [3, 5, 7, 11, 13, 17,
19]
assert s == s[0 .. ] == s[ .. |s|] == s[0 .. |s|] == s[ .. ] // This
Assertion holds
```

The reverse operation of slicing is concatenateing.

```
[2, 3, 5, 7,] + [11, 13, 17, 19] + [ ]// [2, 3, 5, 7, 11, 13, 17, 19]
```

Sequences

Also Sequences, that represent ordered lists, are not allocated on the heap, so they are also immutable, once created. Some examples on handling and slicing Sequences:

```
var empty := [ ]; //Empty Sequence
var s := [2, 3, 5, 7, 11, 13, 17, 19];
s[|s|-1 .. |s|]; s[|s|-1]; // [19], 19
s[1 .. ]; s[ .. |s|-1]; // [2, 3, 5, 7, 11, 13, 17], [3, 5, 7, 11, 13, 17,
19]
assert s == s[0 .. ] == s[ .. |s|] == s[0 .. |s|] == s[ .. ] // This
Assertion holds
```

The reverse operation of slicing is concatenateing.

```
[2, 3, 5, 7,] + [11, 13, 17, 19] + [ ]// [2, 3, 5, 7, 11, 13, 17, 19]
```

Also, one can check if an element is contained within a sequence.

Sequences

Also Sequences, that represent ordered lists, are not allocated on the heap, so they are also immutable, once created. Some examples on handling and slicing Sequences:

```
var empty := [ ]; //Empty Sequence
var s := [2, 3, 5, 7, 11, 13, 17, 19];
s[|s|-1 .. |s|]; s[|s|-1]; // [19], 19
s[1 .. ]; s[ .. |s|-1]; // [2, 3, 5, 7, 11, 13, 17], [3, 5, 7, 11, 13, 17,
19]
assert s == s[0 .. ] == s[ .. |s|] == s[0 .. |s|] == s[ .. ] // This
Assertion holds
```

The reverse operation of slicing is concatenateing.

```
[2, 3, 5, 7,] + [11, 13, 17, 19] + [ ]// [2, 3, 5, 7, 11, 13, 17, 19]
```

Also, one can check if an element is contained within a sequence.

```
assert 2 in s; assert 4 !in s; // These Assertions hold
```

Sequences

Another very useful kind of abbrevation is the „Update“ notation. One has to bear in mind though, that a Sequence, as it is immutable, cannot be updated, so instead of saying a sequence is updated, it is more precise to say, that a new sequence, identical up to one element, is created.

Sequences

Another very useful kind of abbrevation is the „Update“ notation. One has to bear in mind though, that a Sequence, as it is immutable, cannot be updated, so instead of saying a sequence is updated, it is more precise to say, that a new sequence, identical up to one element, is created.

```
var s:= [1, 1, 2, 3, 4, 5];
var t:=s[0:=0]  // [0, 1, 2, 3, 4, 5]
```

Sequences

Another very useful kind of abbrevation is the „Update“ notation. One has to bear in mind though, that a Sequence, as it is immutable, cannot be updated, so instead of saying a sequence is updated, it is more precise to say, that a new sequence, identical up to one element, is created.

```
var s:= [1, 1, 2, 3, 4, 5];
var t:=s[0:=0]  // [0, 1, 2, 3, 4, 5]
```

All these operations work for Arrays as well (when they are transferred into sequences with the slicing operator).

```
var a:= new int[3];
a[0], a[1], a[2]:=3, 2, 1;
assert 2 in a[ .. ]  // This Assertion holds
```

(In)Finite Maps

Maps are associative arrays, transforming a Key into a Value, that need not have the same type. The difference between Finite and Infinite Maps, is that Infinite Maps are allowed to have an infinite domain.

(In)Finite Maps

Maps are associative arrays, transforming a Key into a Value, that need not have the same type. The difference between Finite and Infinite Maps, is that Infinite Maps are allowed to have an infinite domain.

```
var m1:= map[0:=' ',3:='c',5:='e',8:='h',9:='i'];
  // This is a map from int to char
```

(In)Finite Maps

Maps are associative arrays, transforming a Key into a Value, that need not have the same type. The difference between Finite and Infinite Maps, is that Infinite Maps are allowed to have an infinite domain. The domain of a Map is the set of keys for which that the Map has values.

```
var m1:= map[0:=' ',3:='c',5:='e',8:='h',9:='i'];
  // This is a map from int to char
```

(In)Finite Maps

Maps are associative arrays, transforming a Key into a Value, that need not have the same type. The difference between Finite and Infinite Maps, is that Infinite Maps are allowed to have an infinite domain. The domain of a Map is the set of keys for which that the Map has values.

```
var m1:= map[0:=' ',3:='c',5:='e',8:='h',9:='i'];
  // This is a map from int to char
var m2:= map[11:='l',14:='o',17:='r',18:='s'];
```

(In)Finite Maps

Maps are associative arrays, transforming a Key into a Value, that need not have the same type. The difference between Finite and Infinite Maps, is that Infinite Maps are allowed to have an infinite domain. The domain of a Map is the set of keys for which that the Map has values. Iff the domains of two Maps are disjoint, then the Maps are disjoint (!!)

```
var m1:= map[0:=’ ’,3:=’c’,5:=’e’,8:=’h’,9:=’i’];
  // This is a map from int to char
var m2:= map[11:=’l’,14:=’o’,17:=’r’,18:=’s’];
```

(In)Finite Maps

Maps are associative arrays, transforming a Key into a Value, that need not have the same type. The difference between Finite and Infinite Maps, is that Infinite Maps are allowed to have an infinite domain. The domain of a Map is the set of keys for which that the Map has values. Iff the domains of two Maps are disjoint, then the Maps are disjoint (!!)

```
var m1:= map[0:=' ',3:='c',5:='e',8:='h',9:='i'];
  // This is a map from int to char
var m2:= map[11:='l',14:='o',17:='r',18:='s'];
assert m1 !! m2;   // This Assertion holds
```

(In)Finite Maps

Maps are associative arrays, transforming a Key into a Value, that need not have the same type. The difference between Finite and Infinite Maps, is that Infinite Maps are allowed to have an infinite domain. The domain of a Map is the set of keys for which that the Map has values. Iff the domains of two Maps are disjoint, then the Maps are disjoint (!!) and their unification (+) is defined.[*!]

```
var m1:= map[0:=’ ’,3:=’c’,5:=’e’,8:=’h’,9:=’i’];
  // This is a map from int to char
var m2:= map[11:=’l’,14:=’o’,17:=’r’,18:=’s’];
assert m1 !! m2;   // This Assertion holds
```

(In)Finite Maps

Maps are associative arrays, transforming a Key into a Value, that need not have the same type. The difference between Finite and Infinite Maps, is that Infinite Maps are allowed to have an infinite domain. The domain of a Map is the set of keys for which that the Map has values. Iff the domains of two Maps are disjoint, then the Maps are disjoint (!!) and their unification (+) is defined.[*!]

```
var m1:= map[0:=’ ’,3:=’c’,5:=’e’,8:=’h’,9:=’i’];
  // This is a map from int to char
var m2:= map[11:=’l’,14:=’o’,17:=’r’,18:=’s’];
assert m1 !! m2;   // This Assertion holds
[*!] var m:=m1+m2;
```

(In)Finite Maps

Maps are associative arrays, transforming a Key into a Value, that need not have the same type. The difference between Finite and Infinite Maps, is that Infinite Maps are allowed to have an infinite domain. The domain of a Map is the set of keys for which that the Map has values. Iff the domains of two Maps are disjoint, then the Maps are disjoint (!!) and their unification (+) is defined.[*!] The union of the original Maps assigns keys to the same values as the original Maps.

```
var m1:= map[0:=’ ’,3:=’c’,5:=’e’,8:=’h’,9:=’i’];
  // This is a map from int to char
var m2:= map[11:=’l’,14:=’o’,17:=’r’,18:=’s’];
assert m1 !! m2;   // This Assertion holds
[*!] var m:=m1+m2;
```

(In)Finite Maps

Maps are associative arrays, transforming a Key into a Value, that need not have the same type. The difference between Finite and Infinite Maps, is that Infinite Maps are allowed to have an infinite domain. The domain of a Map is the set of keys for which that the Map has values. Iff the domains of two Maps are disjoint, then the Maps are disjoint (!!) and their unification (+) is defined.[*!] The union of the original Maps assigns keys to the same values as the original Maps.

```
var m1:= map[0:=’ ’,3:=’c’,5:=’e’,8:=’h’,9:=’i’];
  // This is a map from int to char
var m2:= map[11:=’l’,14:=’o’,17:=’r’,18:=’s’];
assert m1 !! m2;   // This Assertion holds
[*!] var m:=m1+m2;
[*!] print m[8],m[5],m[11],m[11],m[14],m[0],m[17],m[9],m[18],m[3];
```

(In)Finite Maps

Maps are associative arrays, transforming a Key into a Value, that need not have the same type. The difference between Finite and Infinite Maps, is that Infinite Maps are allowed to have an infinite domain. The domain of a Map is the set of keys for which that the Map has values. Iff the domains of two Maps are disjoint, then the Maps are disjoint (!!) and their unification (+) is defined.[*!] The union of the original Maps assigns keys to the same values as the original Maps.

```
var m1:= map[0:=’ ’,3:=’c’,5:=’e’,8:=’h’,9:=’i’];
  // This is a map from int to char
var m2:= map[11:=’l’,14:=’o’,17:=’r’,18:=’s’];
assert m1 !! m2;   // This Assertion holds
[*!] var m:=m1+m2;
[*!] print m[8],m[5],m[11],m[11],m[14],m[0],m[17],m[9],m[18],m[3];
[*!]   // hello risc
```

(In)Finite Maps

Maps are associative arrays, transforming a Key into a Value, that need not have the same type. The difference between Finite and Infinite Maps, is that Infinite Maps are allowed to have an infinite domain. The domain of a Map is the set of keys for which that the Map has values. Iff the domains of two Maps are disjoint, then the Maps are disjoint (!!) and their unification (+) is defined.[*!] The union of the original Maps assigns keys to the same values as the original Maps.

```
var m1:= map[0:=’ ’,3:=’c’,5:=’e’,8:=’h’,9:=’i’];
  // This is a map from int to char
var m2:= map[11:=’l’,14:=’o’,17:=’r’,18:=’s’];
assert m1 !! m2;   // This Assertion holds
[*!] var m:=m1+m2;
[*!] print m[8],m[5],m[11],m[11],m[14],m[0],m[17],m[9],m[18],m[3];
[*!]   // hello risc
```

[*!] This is planned as future update/feature.

(In)Finite Maps

Similar to Sets, Maps can be defined using constructs like

(In)Finite Maps

Similar to Sets, Maps can be defined using constructs like

```
var qn:= map i: nat | 0<=i<=15 :: i * i;
```

(In)Finite Maps

Similar to Sets, Maps can be defined using constructs like

```
var qn:= map i: nat | 0<=i<=15 :: i * i;
```

For constructing Infinite Maps, using imap one basically has the same possibilities as for Finite Maps. Please note, that Infinite Maps can only be unbounded in a ghost context.

(In)Finite Maps

Similar to Sets, Maps can be defined using constructs like

```
var qn:= map i: nat | 0<=i<=15 :: i * i;
```

For constructing Infinite Maps, using imap one basically has the same possibilities as for Finite Maps. Please note, that Infinite Maps can only be unbounded in a ghost context. ghost: used for verification, but not needed at runtime, so ghost variables will not compiled. You will see an example in the demo I will give.

(In)Finite Maps

Similar to Sets, Maps can be defined using constructs like

```
var qn:= map i: nat | 0<=i<=15 :: i * i;
```

For constructing Infinite Maps, using imap one basically has the same possibilities as for Finite Maps. Please note, that Infinite Maps can only be unbounded in a ghost context. ghost: used for verification, but not needed at runtime, so ghost variables will not compiled. You will see an example in the demo I will give.

```
var tesseracts:= imap i: nat| 0 < i :: i*i*i*i;
```

(In)Finite Maps

Similar to Sets, Maps can be defined using constructs like

```
var qn:= map i: nat | 0<=i<=15 :: i * i;
```

For constructing Infinite Maps, using imap one basically has the same possibilities as for Finite Maps. Please note, that Infinite Maps can only be unbounded in a ghost context. ghost: used for verification, but not needed at runtime, so ghost variables will not compiled. You will see an example in the demo I will give.

```
var tesseracts:= imap i: nat| 0 < i :: i*i*i*i;
```

For Finite Maps one can also calculate the map cardinality

```
ghost var m1:= map[0:=' ',3:='c',5:='e',8:='h',9:='i'];
assert |m1| == 5;  // This Assertion holds
```

Type Synonyms and Opaque Types

For some projects it can be nice to have synonyms for certain types. Dafny allows defining such synonyms with the type keyword.

Type Synonyms and Opaque Types

For some projects it can be nice to have synonyms for certain types. Dafny allows defining such synonyms with the type keyword.

```
type audience = set<persons>
```

Type Synonyms and Opaque Types

For some projects it can be nice to have synonyms for certain types. Dafny allows defining such synonyms with the type keyword.

```
type audience = set<persons>
type complexnum = (real,real)   // ( . , . ) is a Tuple
```

Type Synonyms and Opaque Types

For some projects it can be nice to have synonyms for certain types. Dafny allows defining such synonyms with the type keyword.

```
type audience = set<persons>
type complexnum = (real,real)   // ( . , . ) is a Tuple
```

Such Type Synonyms are, as you would expect, just synonyms, so they have the exact same behavior as the original datatype.

Type Synonyms and Opaque Types

For some projects it can be nice to have synonyms for certain types. Dafny allows defining such synonyms with the type keyword.

```
type audience = set<persons>
type complexnum = (real,real)   // ( . , . ) is a Tuple
```

Such Type Synonyms are, as you would expect, just synonyms, so they have the exact same behavior as the original datatype. A special case of these synonyms are Opaque Types. They are declared, by simply leaving the right side of the type declaration empty.

Type Synonyms and Opaque Types

For some projects it can be nice to have synonyms for certain types. Dafny allows defining such synonyms with the type keyword.

```
type audience = set<persons>
type complexnum = (real,real)   // ( . , . ) is a Tuple
```

Such Type Synonyms are, as you would expect, just synonyms, so they have the exact same behavior as the original datatype. A special case of these synonyms are Opaque Types. They are declared, by simply leaving the right side of the type declaration empty.

```
type T
```

Type Synonyms and Opaque Types

For some projects it can be nice to have synonyms for certain types. Dafny allows defining such synonyms with the type keyword.

```
type audience = set<persons>
type complexnum = (real,real)   // ( . , . ) is a Tuple
```

Such Type Synonyms are, as you would expect, just synonyms, so they have the exact same behavior as the original datatype. A special case of these synonyms are Opaque Types. They are declared, by simply leaving the right side of the type declaration empty. If one wants to point out, that the opaque type is equality supporting, „(==)“ is added to the declaration.

```
type T
```

Type Synonyms and Opaque Types

For some projects it can be nice to have synonyms for certain types. Dafny allows defining such synonyms with the type keyword.

```
type audience = set<persons>
type complexnum = (real,real)   // ( . , . ) is a Tuple
```

Such Type Synonyms are, as you would expect, just synonyms, so they have the exact same behavior as the original datatype. A special case of these synonyms are Opaque Types. They are declared, by simply leaving the right side of the type declaration empty. If one wants to point out, that the opaque type is equality supporting, „(==)“ is added to the declaration.

```
type T
type ES(==)
```

Type Synonyms and Opaque Types

For some projects it can be nice to have synonyms for certain types. Dafny allows defining such synonyms with the type keyword.

```
type audience = set<persons>
type complexnum = (real,real)   // ( . , . ) is a Tuple
```

Such Type Synonyms are, as you would expect, just synonyms, so they have the exact same behavior as the original datatype. A special case of these synonyms are Opaque Types. They are declared, by simply leaving the right side of the type declaration empty. If one wants to point out, that the opaque type is equality supporting, „(==)“ is added to the declaration.

```
type T
type ES(==)
```

The reason for declaring such Opaque Types is, that they can be revealed in a module. Using Opaque Types is using higher level of abstractness.

Newtypes and Conversion

In Dafny it is also possible to define new numeric types, using the newtype keyword.

Newtypes and Conversion

In Dafny it is also possible to define new numeric types, using the newtype keyword.

```
newtype interval = x: real | -1.0 <= x < 1.0
```

Newtypes and Conversion

In Dafny it is also possible to define new numeric types, using the newtype keyword.

```
newtype interval = x: real | -1.0 <= x < 1.0
```

When using newtype one has to bear in mind, that the declared type is not compatible with the base type, when using comparison operators for example.

Newtypes and Conversion

In Dafny it is also possible to define new numeric types, using the newtype keyword.

```
newtype interval = x: real | -1.0 <= x < 1.0
```

When using newtype one has to bear in mind, that the declared type is not compatible with the base type, when using comparison operators for example.

```
var x: interval := 0.5;
assert -3.0 < x;
```

Newtypes and Conversion

In Dafny it is also possible to define new numeric types, using the newtype keyword.

```
newtype interval = x: real | -1.0 <= x < 1.0
```

When using newtype one has to bear in mind, that the declared type is not compatible with the base type, when using comparison operators for example.

```
var x: interval := 0.5;
~~assert -3.0 < x;~~
```

Newtypes and Conversion

In Dafny it is also possible to define new numeric types, using the newtype keyword.

```
newtype interval = x: real | -1.0 <= x < 1.0
```

When using newtype one has to bear in mind, that the declared type is not compatible with the base type, when using comparison operators for example. As „−3.0“ is not from type interval, the compiler throws an error. We are able to correct this by converting x to real again.

```
var x: interval := 0.5;
~~assert -3.0 < x;~~
```

Newtypes and Conversion

In Dafny it is also possible to define new numeric types, using the newtype keyword.

```
newtype interval = x: real | -1.0 <= x < 1.0
```

When using newtype one has to bear in mind, that the declared type is not compatible with the base type, when using comparison operators for example. As „−3.0“ is not from type interval, the compiler throws an error. We are able to correct this by converting x to real again.

```
var x: interval := 0.5;
assert -3.0 < real(x);
```

Newtypes and Conversion

In Dafny it is also possible to define new numeric types, using the newtype keyword.

```
newtype interval = x: real | -1.0 <= x < 1.0
```

When using newtype one has to bear in mind, that the declared type is not compatible with the base type, when using comparison operators for example. As „−3.0“ is not from type interval, the compiler throws an error. We are able to correct this by converting x to real again.

```
var x: interval := 0.5;
assert -3.0 < real(x);   // This assertion holds
```

Newtypes and Conversion

In Dafny it is also possible to define new numeric types, using the newtype keyword.

```
newtype interval = x: real | -1.0 <= x < 1.0
```

When using newtype one has to bear in mind, that the declared type is not compatible with the base type, when using comparison operators for example. As „−3.0“ is not from type interval, the compiler throws an error. We are able to correct this by converting x to real again.

```
var x: interval := 0.5;
assert -3.0 < real(x);   // This assertion holds
```

Like the function real(.) you see above, every numeric type has a corresponding conversion function with the same name. To convert from real to int, using the „trunc“ function is mandatory:

Newtypes and Conversion

In Dafny it is also possible to define new numeric types, using the newtype keyword.

```
newtype interval = x: real | -1.0 <= x < 1.0
```

When using newtype one has to bear in mind, that the declared type is not compatible with the base type, when using comparison operators for example. As „−3.0“ is not from type interval, the compiler throws an error. We are able to correct this by converting x to real again.

```
var x: interval := 0.5;
assert -3.0 < real(x);   // This assertion holds
```

Like the function real(.) you see above, every numeric type has a corresponding conversion function with the same name. To convert from real to int, using the „trunc“ function is mandatory:

```
var euler:= 2.71828;
assert euler.Trunc == 2;   // This assertion holds
```

Other Types

Besides the base types (Boolean, Characters and Numerics), the Collection Types and the other presented Types, there are various other Types and Concepts that are known from other programming languages, such as:

Other Types

Besides the base types (Boolean, Characters and Numerics), the Collection Types and the other presented Types, there are various other Types and Concepts that are known from other programming languages, such as:

- Classes `class`

Other Types

Besides the base types (Boolean, Characters and Numerics), the Collection Types and the other presented Types, there are various other Types and Concepts that are known from other programming languages, such as:

- Classes `class`
- Traits `trait`

Other Types

Besides the base types (Boolean, Characters and Numerics), the Collection Types and the other presented Types, there are various other Types and Concepts that are known from other programming languages, such as:

- Classes `class`
- Traits `trait` ≅ Interfaces/Abstract Superclasses

Other Types

Besides the base types (Boolean, Characters and Numerics), the Collection Types and the other presented Types, there are various other Types and Concepts that are known from other programming languages, such as:

- Classes `class`
- Traits `trait` ≅ Interfaces/Abstract Superclasses
- Inductive Datatypes `datatype`

Other Types

Besides the base types (Boolean, Characters and Numerics), the Collection Types and the other presented Types, there are various other Types and Concepts that are known from other programming languages, such as:

- Classes `class`
- Traits `trait` ≅ Interfaces/Abstract Superclasses
- Inductive Datatypes `datatype` ≅ Finite Trees

Other Types

Besides the base types (Boolean, Characters and Numerics), the Collection Types and the other presented Types, there are various other Types and Concepts that are known from other programming languages, such as:

- Classes class
- Traits trait ≅ Interfaces/Abstract Superclasses
- Inductive Datatypes datatype ≅ Finite Trees
- Co-inducive Datatypes codatatype

Other Types

Besides the base types (Boolean, Characters and Numerics), the Collection Types and the other presented Types, there are various other Types and Concepts that are known from other programming languages, such as:

- Classes class
- Traits trait ≅ Interfaces/Abstract Superclasses
- Inductive Datatypes datatype ≅ Finite Trees
- Co-inducive Datatypes codatatype ≅ Infinite Trees or Streams

Other Types

Besides the base types (Boolean, Characters and Numerics), the Collection Types and the other presented Types, there are various other Types and Concepts that are known from other programming languages, such as:

- Classes class
- Traits trait ≅ Interfaces/Abstract Superclasses
- Inductive Datatypes datatype ≅ Finite Trees
- Co-inducive Datatypes codatatype ≅ Infinite Trees or Streams

Also, the Functions, I already presented, are a seen as Types. The reads and requires of a Function are Functions themselves.

Table of Contents

Lemmas

Lemmas are used, when the steps to prove a programm are to complex for Dafny to discover them on its own.
Lemmas are ghost methods (can be replaced for better readability with lemma) or function Lemmas (usually a certain predicate).

Lemmas

Lemmas are used, when the steps to prove a programm are to complex for Dafny to discover them on its own.

Lemmas are ghost methods (can be replaced for better readability with lemma) or function Lemmas (usually a certain predicate).

The correctness of the step must follow from the postcondition of the Lemma (or chain of Lemmas), but there are two possible ways to verify the correctness of a programm with Lemmas.

Lemmas

Lemmas are used, when the steps to prove a programm are to complex for Dafny to discover them on its own.

Lemmas are ghost methods (can be replaced for better readability with lemma) or function Lemmas (usually a certain predicate).

The correctness of the step must follow from the postcondition of the Lemma (or chain of Lemmas), but there are two possible ways to verify the correctness of a programm with Lemmas.

- Leave the Method body empty and prove it for example mathematically.

Lemmas

Lemmas are used, when the steps to prove a programm are to complex for Dafny to discover them on its own.

Lemmas are ghost methods (can be replaced for better readability with lemma) or function Lemmas (usually a certain predicate).

The correctness of the step must follow from the postcondition of the Lemma (or chain of Lemmas), but there are two possible ways to verify the correctness of a programm with Lemmas.

- Leave the Method body empty and prove it for example mathematically.

Leaving the body empty is always good for a first approach, as one can see if the provided postcondition of the Lemma is strong enough.

Lemmas

Lemmas are used, when the steps to prove a programm are to complex for Dafny to discover them on its own.
Lemmas are ghost methods (can be replaced for better readability with lemma) or function Lemmas (usually a certain predicate).
The correctness of the step must follow from the postcondition of the Lemma (or chain of Lemmas), but there are two possible ways to verify the correctness of a programm with Lemmas.

- Leave the Method body empty and prove it for example mathematically.

Leaving the body empty is always good for a first approach, as one can see if the provided postcondition of the Lemma is strong enough. Sometimes this way also seems to be the only possible one.

Lemmas

Lemmas are used, when the steps to prove a programm are to complex for Dafny to discover them on its own.

Lemmas are ghost methods (can be replaced for better readability with lemma) or function Lemmas (usually a certain predicate).

The correctness of the step must follow from the postcondition of the Lemma (or chain of Lemmas), but there are two possible ways to verify the correctness of a programm with Lemmas.

- ▷ Leave the Method body empty and prove it for example mathematically.

Leaving the body empty is always good for a first approach, as one can see if the provided postcondition of the Lemma is strong enough. Sometimes this way also seems to be the only possible one.

- ▷ Prove the code by code, i.e. fill the Method body with the needed code.

Lemmas

Lemmas are used, when the steps to prove a programm are to complex for Dafny to discover them on its own.

Lemmas are ghost methods (can be replaced for better readability with lemma) or function Lemmas (usually a certain predicate).

The correctness of the step must follow from the postcondition of the Lemma (or chain of Lemmas), but there are two possible ways to verify the correctness of a programm with Lemmas.

- Leave the Method body empty and prove it for example mathematically.

Leaving the body empty is always good for a first approach, as one can see if the provided postcondition of the Lemma is strong enough. Sometimes this way also seems to be the only possible one.

- Prove the code by code, i.e. fill the Method body with the needed code.

This means, one has to „poke“ Dafny into the right direction, to make it consider things it knows but does not consider.

Lemmas

Lemmas are used, when the steps to prove a programm are to complex for Dafny to discover them on its own.

Lemmas are ghost methods (can be replaced for better readability with lemma) or function Lemmas (usually a certain predicate).

The correctness of the step must follow from the postcondition of the Lemma (or chain of Lemmas), but there are two possible ways to verify the correctness of a programm with Lemmas.

- ▷ Leave the Method body empty and prove it for example mathematically.

Leaving the body empty is always good for a first approach, as one can see if the provided postcondition of the Lemma is strong enough. Sometimes this way also seems to be the only possible one.

- ▷ Prove the code by code, i.e. fill the Method body with the needed code.

This means, one has to „poke“ Dafny into the right direction, to make it consider things it knows but does not consider. This is usually done by a chain of Assertions, Case distinction,...

References, Image and Code Sources I

[1] Jason Koenig and Rustan Leino. Getting Started with Dafny: A Guide. `http://research.microsoft.com/en-us/um/people/leino/papers/krml220.pdf`, 2011.

[2] Jason Koenig, Michał Moskal, Nadia Polikarpova, Valentin Wüstholz, Reza Ahmadi, Nada Amin, Maria Christakis, Chris Hawblitzel, Jay Lorch, Qunyan Mangus, Bryan Parno, Clément Pit-Claudel, Michael L. Roberts, and Dan Rosén, 2008.

[3] Rustan Leino. Schorr-Waite and other marking algorithms, written and verified in Dafny. `http://rise4fun.com/Dafny/ys`, 2008.

[4] Rustan Leino. Dafny: An Automatic Program Verifier for Functional Correctness. `http://research.microsoft.com/en-us/um/people/leino/papers/krml203.pdf`, 2010.

[5] Rustan Leino. Types in Dafny. `http://research.microsoft.com/en-us/um/people/leino/papers/krml243.html`, 2015.

References, Image and Code Sources II

[6] Rustan Leino. Terminating Functions and Extreme Predicates in Dafny: A Tutorial. `http://research.microsoft.com/en-us/um/people/leino/papers/krml250.pdf`, 2015.

[7] Microsoft Research Website. Microsoft Research. `http://research.microsoft.com/en-us/default.aspx`, 2015. especially the tabs Our Research and About Us.

[8] K.Rustan M.Leino. Dafny Website. `http://research.microsoft.com/en-us/projects/dafny/`, 2015.

[9] Tim Wood. Distributivity of Sequence Map over Function Composition in Dafny. `http://www.lexicalscope.com/blog/2015/01/21/distributivity-of-sequence-map-over-function-composition-` 2015.

[10] Youtube Icon. `http://www.youtube.com/yt/brand/using-logo.html`, 2015.