Programming Fundamentals

Dafny - week 7

A. Meijster

University of Groningen

university of groningen

Programming Fundamentals

- C is used as the main language for programming tasks.
- Dafny is mostly used for proving the correctness of a program.

Edsger W. Dijkstra:

"*Program testing can be used to show the presence of bugs, but never to show their absence!*"

We need a rigorous verification tool. Dafny provides this.

university of groningen

Hello world!

In almost any programming course, the first program you make prints `Hello world!` on the screen. This intro in Dafny is no exception.

You can use any editor you like, but for Dafny programs we advise to use the Visual Studio code editor, because it has a very nice plugin for Dafny.

We start the editor from a terminal using the command `code hello.dfy`.

The first time you use the editor, it may complain that it is in *restricted mode*. In that case, click `Manage` and add the directory (folder) as trusted. From that point on, you are no longer in restricted mode.

university of groningen

Hello world!

In the editor, we type the following program:

```
method Main ()
{
  print "Hello world!\n";
}
```

Save the file as `hello.dfy`.
Next we compile and run it (in a terminal):

```
$ dafny hello.dfy
Dafny program verifier finished with 0 verified, 0 errors
Compiled assembly into hello.dll
$ dotnet hello.dll
Hello world!
```

You can compile and run a program in a single command:

```
$ dafny run hello.dfy
Dafny program verifier finished with 0 verified, 0 errors
Compiled assembly into hello.dll
Hello world!
```

You can also run the program directly from the visual studio code editor (press F5).

[Live Demo]

university of groningen

42

```
method Main()
{ // note that { is on a separate line!
  var x : int;    // introduces local integer variable x
  x := 14;        // assigns 14 to x
  x := 3*x;       // assigns 3*x==42  to x
  print "The answer to the Ultimate Question of Life, ";
  print "the Universe, and Everything ";
  print "is ", x, ".\n";
}
```

Output

```
The answer to the Ultimate Question of Life, the
Universe, and Everything is 42.
```

university of groningen

print statement

You may have missed a subtlety on the previous slide.
Note the command `print "is ", x, ".\n";`

In Dafny, `print` is not a function call, but a build in keyword.
[Note: `printf` in C is a function call from the standard library.]

This may seem irrelevant, but it is not! There is a difference between
`print 21, " ", 42, "\n"` and `print (21," ",42,"\n").`

```
method Main()
{
  print 21, " ", 42, "\n";
  print (21," ",42,"\n");
}
```

output

```
21 42
(21, [' '], 42, ['\n'])
```

university of groningen

Dafny basics: methods

A *method* is like a function in C, but there are clear differences.

```
method Triple(x: int) returns (r: int)
{
    var y := 2*x;
    r := x + y;
}
```

- This method takes an *in-parameter* `x` of type integer and returns an *out-parameter* `r`.
- In the body of a method, in-parameters can be read but cannot be assigned to.
- Out-parameters act as local variables (can be read and/or written).
- When a method ends, whatever value output-parameters have will be the values returned to the caller.

university of groningen

Dafny basics: calling a method

```
method Main()
{
    var y := Triple(14);
    print y, "\n";
    y := Triple(12345678987654321234567898765432);
    y := Triple(y);
    y := y*y*y;
    print "really Huge: ", y, "\n";
}
```

Let's compile the program:

```
$ dafny triple.dfy
Dafny program verifier finished with 1 verified, 0 errors
```

What was verified? Well, that `14` and `12345678987654321234567898765432` are `int`s.

Dafny's `int` type has infinite precision (no overflow!).

Output

```
42
Huge:
13717421042524005692729766529492455624142661097393 69
```

university of groningen

No method calls in expressions

```
method Main()
{
  var y : int;

  y := 2*Triple(14);  // not allowed

  print y, "\n";
}
```

Error message

```
Error:  expression is not allowed to invoke a method (Triple)
```

university of groningen

No method calls in expressions

As a consequence, this is also not allowed:

```
method Main()
{
  var y : int;

  y := Triple(Triple(14));  // not allowed

  print y, "\n";
}
```

Error message

```
Error:  method call is not allowed to be used in an expression
context (Triple)
```

university of groningen

Assert statements

```
method Triple(x: int) returns (r: int)
{
  var y := 2*x;
  r := x + y;
  assert r == 3*x;
}
```

An `assert` is like a comment, but there is an important difference.

Comments are simply skipped by the compiler, so even if you write nonsense the compiler will not complain.

However, `assert`s are much more powerful. They are *proof obligations*!

Let us see what the compiler says:

```
$ dafny assert.dfy
Dafny program verifier finished with 2 verified, 0 errors
```

The compiler verified at compile time that `r==3*x`. No machine code was produced (zero overhead!). The `assert` statement is a so-called *ghost*-statement.

university of groningen

Assert statements

Let us see what happens if we try to compile a false assertion:

```
method Triple(x: int) returns (r: int)
{
  var y := 2*x;
  r := x + y;
  assert r == 3*x + 1;
}
```

```
$ dafny assert.dfy
assert.dfy(5,13):  Error:  assertion might not hold
Dafny program verifier finished with 1 verified, 1 error
```

Compilation failed, and no machine code was produced.

university of groningen

Assert statements

Writing down an assertion amounts to asking the verifier,
"*do you know that this condition always holds at this point?*".

Alert

The condition may in fact always hold, but the verifier may not be "clever" enough to conclude it. In those cases, we'll need to help it along.

university of groningen

Assert statements

```
method Triple(x: int) returns (r: int)
{
  var y := 2*x;
  r := x + y;
  assert r == 3*x;      // passes
  assert r == 10*x;     // error
  assert r < 5;         // passes (!?)
  assert false;         // obviously an error
}
```

```
$ dafny assert.dfy
assert.dfy(6,13):  Error:  assertion might not hold
assert.dfy(8,11):  Error:  assertion might not hold
```

Why did `assert r<5` pass?

For traces that reach it, `r==3*x && r==10*x` holds, so `3*x==10*x`.

Hence, the verifier concludes that `x==0 && r==0`, which implies `r < 5`.

university of groningen

Method contracts

```
method Triple(x: int) returns (r: int)
{
   r := 3*x;
   assert r == 3*x;
}

method Main()
{
    var y := Triple(14);
    assert y == 42;
    print y, "\n";
}
```

```
$ dafny contract.dfy
contract.dfy(10,13): Error: assertion might not hold
```

Why did `assert y==42` fail?

Well, reason as if you do not know the body of `Triple`. We say that methods are *opaque*. Would you be able to guarantee the assertion? No!

university of groningen

Method contracts

```
method Triple(x: int) returns (r: int)
ensures r == 3*x
{
   r := 3*x;
   assert r == 3*x;
}

method Main()
{
    var y := Triple(14);
    assert y == 42;
    print y, "\n";
}
```

```
$ dafny contract.dfy
Dafny program verifier finished with 2 verified, 0 errors
```

Now the program compiles. The predicate that follows the keyword `ensures` says that the caller can rely on `r` (return value) being `3*x`.

university of groningen

Method contracts

```
method Triple(x: int) returns (r: int)
requires x >= 0
ensures r == 3*x
{
   r := 3*x;
   assert r == 3*x;
}

method Main()
{
    var y := Triple(-14);
    assert y == -42;
    print y, "\n";
}
```

```
$ dafny contract.dfy
contract.dfy(11,19): Error: a precondition for this call could not be
proved
contract.dfy(2,11): Related location: this is the precondition that
could not be proved
```

The program fails to compile, because `-14` fails the requirement that `x >= 0`.

university of groningen

Method contracts

```
method Triple(x: int) returns (r: int)
requires x >= 0
ensures r == 3*x
{
   r := 3*x;
   assert r == 3*x;
}

method Main()
{
    var y := Triple(14);
    assert y == 42;
    print y, "\n";
}
```

```
$ dafny contract.dfy
Dafny program verifier finished with 2 verified, 0 errors
Compiled assembly into contract.dll
```

Of course, this time, the verifier has no objections.

university of groningen

Method contracts

Dafny has the data type `nat`, which are natural numbers (i.e. `int`s that are `>= 0`).

```
method Triple(x: nat) returns (r:  nat)
ensures r == 3*x
{
   r := 3*x;
   assert r == 3*x;
}

method Main()
{
    var y := Triple(-14);
    assert y == -42;
    print y, "\n";
}
```

```
$ dafny contract.dfy
contract.dfy(10,20):  Error:  value does not satisfy the subset
constraints of 'nat'
Dafny program verifier finished with 1 verified, 1 error
```

By using data type `nat` for `x`, we introduced implicitly `requires x>=0`.

university of groningen

Method contracts

```
method Triple(x: nat) returns (r:  nat)
ensures r == 3*x
{
   r := 3*x;
   assert r == 3*x;
}

method Main()
{
    var y := Triple(14);
    assert y == 42;
    print y, "\n";
}
```

```
$ dafny contract.dfy
Dafny program verifier finished with 2 verified, 0 errors
Compiled assembly into contract.dll
```

Since `14` is a valid `nat`, the verifier has no objections.

university of groningen

Method contracts

```
method Triple(x: int) returns (r: int)
ensures r == 3*x
{
   var y := x/2;
   r := 6*y;
}

method Main()
{
    var y := Triple(14);
    assert y == 42;
    print y, "\n";
}
```

```
$ dafny triple.dfy
Error:  a postcondition could not be proved on this return path
Location 2:  this is the postcondition that could not be proved
```

university of groningen

Method contracts

```
method Triple(x: int) returns (r: int)
requires x%2 == 0
ensures r == 3*x
{
   var y := x/2;
   r := 6*y;
}

method Main()
{
    var y := Triple(14);
    assert y == 42;
    print y, "\n";
}
```

```
$ dafny triple.dfy
Dafny program verifier finished with 3 verified, 0 errors
```

Now the verification passes.

university of groningen

Method contracts: specifications

- A method *contract* has two fundamental parts: a *precondition* and a *postcondition*.
- Together, they form the *specification* of the method.
- The precondition says when it is legal for a caller to invoke the method.
- This precondition is a proof obligation at every call site!
- The precondition of a method is defined via the `requires` keyword.
- The postcondition can be assumed to hold upon return from the invocation at the call site.
- The postcondition is a proof obligation for the implementer of the method.
- The postcondition of a method is defined via the `ensures` keyword.

- We can omit a precondition. This is equivalent with `requires true`.
- We can omit a postcondition. This is equivalent with `ensures true`.

university of groningen