CS-135, Lab 10
Program Verification in Dafny

To be ticked off:

- on **4.4.2025** or
- a week later, on **11.4.2025 between 11am and 12 noon.**

Please work in teams of two.
We recommend using the university computers as they are all ready and set up for these labs.
There are Computer Instructions at the beginning of this Lab Sheet.

- You can obtain two marks by solving this sheet.
- Each completed task gives you one mark.
- All group participants need to be present to be ticked off.
- Check your marks on Lab-Tracker after having been ticked off, marks cannot be changed after.

Learning Outcomes

- Understanding of Program Verificaion
- Computing of Weakest Preconditions
- Coming out with Loop Invariants

Required Resources

- Files: `hello.dfy`, `Task1.dfy`, `Task2.dfy`, `TimeMeasurement.java`

Where to Look Up Things

- Lecture Slides on Program Verification, weeks 9 & 10
- Dafny Cheat Sheet: `https://dafny.org/latest/DafnyCheatsheet.pdf`

Computer Instructions

→ sign in to OneDrive in order to have your files synched to the cloud

→ Under Teaching Software
Faculty
choose VScode [Do not choose Visual Studio!!!]
start Visual Studio Code (it might be the case that it still needs to be installed)
note: the window is often hidden

→ In VScode choose extensions (left sidebar)
enter Dafny
select Dafny
click install
create a new file with extension .dfy, say hugo.dfy
click into this file
if there are migration instruction [likely], accept the update to version 4.6.0.0, and you are ready to go

→ In case there is no migration pop-up
click on Dafny in the extensions field
click uninstall
click install
(possibly repeat this cycle)
until you are offered restart extension
click restart extension
a pop-up upgrade version appears accept the upgrade to 4.6.0

→ click on hugo.dfy
check for Dafny 4.6.0.0 in the bottom right corner - then things are correctly installed

→ Seeing Dafy in Action:

Download the file hello.dfy
Open it in VScode
Press F5: this executes hello.dfy
– you should see "hello" and the number "34" on the screen
Change the assertion "assert $42 > 0$;" to "assert $42 < 0$;"
– you see a right marker on the left of this line

Introduction to the Lab

Information hiding is an important principle in Software Engineering. Applied to methods, this principle means: each method comes with a contract, which in Dafny is expressed through two assertions following the keywords `requires` and `ensures`:

```
method M (x:X) returns (y:Y)
    requires P
    ensures Q
{
    body
}
```

For method `M` holds: if called such that predicate `P` holds, then `M` will produce an output such that predicate `Q` holds.

In the context of Software Engineering, such a contract allows the method author to exchange the `body` of `M`. Such exchange is legit, as long as also the new body fulfills the contract. For example, one might be interested to exchange an algorithm, say, insert sort against quick sort. Both these algorithms fulfill the same contract.

Dafny is a tool that tries to check automatically, if the `body` of a method fulfills a contract. In case Dafny's check is successful, the contract is fulfilled. In case the Dafny fails, we just know that – within the given resources of time and memory – Dafny was not able to provide a proof that the contract holds, i.e., we do not know if the contract holds or not.

Explanation - Method Correctness

To prove that a method implementation is correct with respect to its specification, we show here[a] that the given precondition implies the weakest precondition of the method body with respect to the postcondition.

> We use the following notion concerning such conditions: Suppose `S` is a statement and `Q` is a predicate on the poststate of `S`, then `WP [[S, Q]]` is the weakest precondition of `S` with respect to `Q`.
>
> In symbols, to prove that the following method is correct:
>
> ```
> method M (x:X) returns (y:Y)
>     requires P
>     ensures Q
> {
>     body
> }
> ```
>
> we need to prove
>
> ```
> P ==> WP [[ Body, Q]]
> ```
>
> [from Leino: Program Proofs, MIT Press 2023]

Often, we simply choose `WP [[ Body, Q]]` for `P`.

[a] As usual, there are several proof techniques that one could apply.

Task 1 - Deciding Correctness for Assignment, Sequential Composition, and Branching Structures

Introduction to the Task: If the `body` of a method[1] consists only of assignment, sequential composition, and branching structures, it is *decidable* if it fulfills a contract, i.e., we can expect Dafny to tell us if a contract is correct or not.

Explanation - Weakest Precondition for Assignment

The rule to compute the weakest precondition for assignment is as follows:

```
WP [[ x := E, Q]] = Q[x := E]
```

The substitution operator `[x := E]` after `Q` means: replace each occurrence of `x` in `Q` by `E`.

Example (Substitution operator)

```
                    x * y >= 50 [x := 2 * x]
=                   2 * x * y >= 50
```

Example (Assignment)

```
    WP [[ x := 2 * x, (x * y >= 50)]]
=   (x * y >= 50)[x := 2 * x]
=   ( 2 * x * y >= 50)
```

Consider the Dafny method `M1`:

```
method M1 (x: int, y:int) returns (z:int)
    requires ...
    ensures  z == 2 * (x - y)
{
   z := 3 * x;
}
```

Complete the contract of `M1` and check with Dafny if your completion is correct.

To this end, compute the weakest precondition `P` of the method body with respect to the postcondition stated after the keyword `ensures`. Enter your precondition `P` after the keyword `requires`.

Consider the Dafny method `M2`:

```
method M2 (x: int, y:int) returns (z:int)
    requires ...
    ensures  (x >= z) && (y == z - 2)
{
   z := 3 * x;
}
```

Complete the contract of `M2` and check with Dafny if your completion is correct.

[1] that does not call any methods and whose contract lies in the decidable fragment of a theory

Explanation - Weakest Precondition for Sequential Composition

Suppose `S` and `T` are statements, and `R` is a predicate on the post-state of `T`. The rule to compute the weakest precondition for sequential composition is as follows:

```
WP [[ S;T, R]] = WP [[ S, WP[[T, R] ]]
```

I.e., in order to find the weakest precondition of `S;T` with regards to `R`, we first compute the weakest precondition `WP[[T, R]]`of `T` with regards to `R`, and then compute the weakest precondition of `S` with regards to `WP[[T, R]]`.

Example (Sequential Composition)

```
  WP [[ y := 2 * x; z := y + 2, z > 0 ]]
= WP [[ y := 2 * x, WP [[ z := y + 2, z > 0  ]] ]]
= WP [[ y := 2 * x, y + 2 > 0 ]]
= (2* x) + 2 > 0
```

Consider the Dafny method `M3`:

```
method M3 (x: int) returns (z:int)
   requires ...
   ensures z >= 20
{
   z := x - 5;
   z := 3 * z;
}
```

Complete the contract of `M3` and check with Dafny if your completion is correct.

Consider the Dafny method `M4`:

```
method M4 (x: int, y: int) returns (z:int)
   requires ...
   ensures z >= 2 * x + y
{
   z := x - 5;
   z := 3 * z;
}
```

Complete the contract of `M4` and check with Dafny if your completion is correct.

Explanation - Weakest Precondition for Branching Structures

Suppose `B` is a condition, and `S` and `T` are statements. Consider the branching structure

```
if B {S} else {T}
```

and a poststate `Z` of it.
The rule to compute the weakest precondition for Branching Structures is:

```
   WP [[ if B {S} else {T}, Z ]]
=   B ==> WP[[S, Z]]
   &&
   !B ==> WP[[T, Z]]
```

I.e., in order to determine the weakest precondition of a branching structure, we separately compute the weakest preconditions for each of the branches, namely `WP[[S, Z]]` and `WP[[T, Z]]`. Then we combine these with the condition `B`. If `B` is true, statements `S` are executed, if `B` is false, i.e., `!B` is true, statements `T` are executed. In both cases, the preconditions shall hold.

Example (Branching Structure)

```
   WP [[  if (x>2) {x := x+1} else {x := 7}, x > 0 ]]
=      ((x > 2)  ==> WP [[x:=x+1, x > 0]])
   &&  ((x <= 2) ==> WP [[x:=7  , x > 0]])
=      ((x > 2)  ==> x + 1 > 0)
   &&  ((x <= 2) ==> 7 > 0)
=  true && (x <=2 )
=  x <= 2
```

Consider the Dafny method `M5`:

```
method M5 (x: int, y: int) returns (z:int)
   requires ...
   ensures z >= 2 * x + y
{
   if (x > 0)
   {
       z := 17;
   }
   else
   {
       z := 2 * x;
   }
}
```

Complete the contract of `M5` and check with Dafny if your completion is correct.

Consider the Dafny method `M6`:

```
method M6 (x: int, y: int) returns (z:int)
requires ...
ensures z >= 2 * x + y
{
   if (x * y > 27)
   {
       z := y + 4 - x;
   }
   else
   {
       z := y - 4;
       z := z + 1;
   }
}
```

Complete the contract of `M6` and check with Dafny if your completion is correct.

Assessment: The file Task1.dfy in Visual Studio with a green bar from top to bottom.

Task 2 - Proving Correctness for Loops with the Help of Invariants

Introduction to the Task:

Programm verification becomes undecidable the moment loops are involved. Only with some additional 'help' in form of so-called (loop) invariants we can expect Dafny to tell us if a contract is correct. We will explore this on the example of verifying different methods for computing the integer square root.

Explanation - What Invariants are About

> The word *invariant* means "not changing". In the context of program verification, an invariant is a predicate concerning a loop. The invariant holds both, *before* the body of the loop is executed and *after* the body of the loop is executed: the validity of the invariant does not change.
>
> In Dafny, we write invariants as follows:
>
> ```
> while B
>    invariant J
> {
>    Body
> }
> ```
>
> For such an annotation, the Dafny verifier tries to prove:
>
> ```
> (B && J) ==> WP [[ Body, J]]
> ```
>
> When entering the loop, we know that `B` holds. Suppose further that also `J` holds. In order for `J` to be an invariant, `(B && J)` must imply the weakest precondition of `Body` with respect to `J`.

The *integer square root of a non-negative integer* y is defined to be the natural number x with $x * x \leq y < (x + 1) * (x + 1)$. We write

$$x = isqrt(y).$$

For example, $isqrt(27) = 5$, as $5 * 5 \leq 27 < 6 * 6$.

Another way to think about this is that we are first looking for the square root of a number y. This root is usually a real number. The number x is then y rounded down to the next natural number. In the example $\sqrt{27} = 5.196\ldots$, we then round down to get $x = 5$. If we consider $\sqrt{35} = 5.916\ldots$, we would again round down to get $x = 5$. This means, 27 and 35 both have 5 the integer square root 5.

Explanation - How to Come Up with Invariants

Invariants need to be invented or found. They represent insight into the algorithm at hand. In the lectures we discussed two design principles for invariants.

Invariant design principle 1: Express relations (which value is larger than another one) as an invariant. The below invariant `i - 1 < n` is of this type, it relates the values of the variables `i` and `n` by saying which value is smaller than the other.

```
method count0_3 (n:int) returns (r: int)
    requires n >= 0;
{
    var i:int := 0;
    while (i < n)
        invariant i - 1 < n;
    {
         i:=i+1;
    }
    r := 0;
}
```

Invariant design principle 2: State the intermediate result computed in the i-th run of the loop. The below invariant `sum == i * (i+1)/2` is of this type, it uses the formula `r == n*(n+1)/2` stated after the keyword `ensures`. However, it is

- replacing variables `r` and `n` used to state the end result
- with variables `sum` and `i` used in the loop to compute an intermediate result.

```
method sum(n:int) returns(r:int)
    requires n >= 0
    ensures r == n*(n+1)/2
{
    var sum : int := 0;
    var i: int := 0;
    while (i < n)
         invariant sum == i * (i+1)/2;
     {
          i := i+1;
          sum := sum + i;
     }
     r := sum;
}
```

Consider the method `SquareRootAscendingSearch` from the file `Task2.dfy`: Without an invariant, Dafny can't establish the contract. Follow the hints in the file and prove that the method is correct.

Consider the method `SquareRootDescendingSearch` in the file `Task2.dfy`. Without an invariant, Dafny can't establish the contract. Follow the hints in the file and prove that the method is correct.

➲ Consider the method `SquareRootAscendingUsingAddition` in the file `Task2.dfy`. Astonishingly, Dafny can prove its contract without requiring an invariant. Thus, there is nothing for you to do :-)

➲ Consider the method `SquareRootBinarySearch` in the file `Task2.dfy`. Without an invariant, Dafny can't establish the contract. Follow the hints in the file and prove that the method is correct.

Assessment: The file Task2.dfy in Visual Studio with a green bar from top to bottom.

Assessment Check List

✔ Two tasks to be ticked off.

Fun Task (no marks) – Using Verified Dafny Methods in Java

Introduction to the Task: Now, that we have verified methods, let's try to call them from Java. Any code that we call is supposed to be correct w.r.t. its contract! [up to translation errors].

To do this in a nice way is possibly, but also a bit of work. Thus, we will use a shortcut here.

- After verifying everything in Task 2, we can compile things to Java. To this end:
- Choose under View the Command Pallet
- search dafny
select Dafny: Build with Custom Arguments
enter build –target:java –spill-translation
- Effect: this produces a folder and a .jar file
- ignore the .jar file
- in the folder you find produed code including the dafny runtime
- we are interested in `_System/__default.java`
- copy from `__default.java` the code for the first three meethods into IntelliJ.
- Call call the first 3 methods from within the main method of the Main class file `TimeMeasurement.java`.
- Find a number for which there is a runtime difference for the 3 methods – there ought to be one, as the integer square root is closer to 0 than to y; also, the algorithm with addition is supposed to be faster than the other two.