Programming Fundamentals

Dafny - week 7

A. Meijster

University of Groningen

university of groningen

Proof rules for a while-loop

From

- $\text{vf} \in \mathbb{Z}$
- $J \wedge B \Rightarrow \text{vf} \geq 0$
- $\{J \wedge B \wedge \text{vf} = V\}\ S\ \{J \wedge \text{vf} < V\}$

we may conclude

$$\{J\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{end}\ \{J \wedge \neg B\}$$

university of groningen

Step-by-step design of a while-program: $\{P\}\ T\{Q\}$

1. Choose an invariant J and a guard B such that $J \wedge \neg B \Rightarrow Q$
2. Initialization: Find a command T_0 such that $\{P\}\ T_0\ \{J\}$.
   Note that if $P \Rightarrow J$, then this step can be skipped.
3. Variant function: choose a $\text{vf} \in \mathbb{Z}$ and prove $J \wedge B \Rightarrow \text{vf} \geq 0$
4. Body of the loop: Find a command S such that
   $\{J \wedge B \wedge \text{vf} = V\}\ S\ \{J \wedge \text{vf} < V\}$
5. Conclude that $\{P\}\ T_0;$ **while** B **do** S **end** $\{Q\}$

university of groningen

Example: integer square root

In the previous lecture, we proved the correctness of the following program fragment that computes the integer square root of x.

const $x : \mathbb{N}$;
var $y : \mathbb{Z}$;
$y := 0$;
  $\{J : y \geq 0 \wedge y^2 \leq x\}$
    $(* \text{ vf} = x - y^2 *)$
while $(y+1)*(y+1) \leq x$ **do**
  $y := y + 1$;
end;
  $\{Q : y \geq 0 \wedge y^2 \leq x < (y+1)^2\}$

university of groningen

Binary search: integer square root

But, choosing a different invariant leads to a much more efficient program.

We start with the post-condition $y \geq 0 \ \wedge \ y^2 \leq x < (y+1)^2$.

This time, we choose the following invariant and guard:

$$\begin{array}{lcl} J & : & 0 \leq y \leq z \ \wedge \ y^2 \leq x < (z+1)^2 \\ B & : & y \neq z \end{array}$$

Clearly, $J \ \wedge \ \neg B \Rightarrow Q$.

Initialization is easy with $y := 0; \ z := x;$.
It is clear that $x \in \mathbb{N} \Rightarrow \mathcal{WP}(y := 0; z := x, \ J)$.

Variant function: We choose $\text{vf} = z - y \in \mathbb{Z}$. Clearly, $J \ \wedge \ B \Rightarrow \text{vf} \geq 0$

university of groningen

Binary search: continued

Body of the loop: $\{J \ \wedge \ B \ \wedge \ \text{vf} = V\} \ S \ \{J \ \wedge \ \text{vf} < V\}$

$\{0 \leq y \leq z \ \wedge \ y^2 \leq x < (z+1)^2 \ \wedge \ y \neq z \ \wedge \ z - y = V\}$
$(* \ (y \leq z \wedge y < z) \equiv (y + y \leq y + z \wedge y + z < z + z) \equiv 2 \cdot y \leq y + z < 2 \cdot z \ *)$
$\{0 \leq y \leq (y+z) \ \textbf{div} \ 2 < z \ \wedge \ y^2 \leq x < (z+1)^2 \ \wedge \ z - y = V\}$
$m := (y+z) \ \textbf{div} \ 2;$
$\{0 \leq y \leq m < z \ \wedge \ y^2 \leq x < (z+1)^2 \ \wedge \ z - y = V\}$
if $(m+1) * (m+1) > x$ **then**
$\{(m+1)^2 > x \ \wedge \ 0 \leq y \leq m < z \ \wedge \ y^2 \leq x < (z+1)^2 \ \wedge \ z - y = V\}$
$(*$ logic; calculus; prepare $z := m \ *)$
$\{0 \leq y \leq m \ \wedge \ y^2 \leq x < (m+1)^2 \ \wedge \ m - y < V\}$
$z := m;$
$\{0 \leq y \leq z \ \wedge \ y^2 \leq x < (z+1)^2 \ \wedge \ z - y < V\}$
else
$\{(m+1)^2 \leq x \ \wedge \ 0 \leq y \leq m < z \ \wedge \ y^2 \leq x < (z+1)^2 \ \wedge \ z - y = V\}$
$(*$ logic; calculus; prepare $y := m + 1 \ *)$
$\{0 \leq m+1 \leq z \ \wedge \ (m+1)^2 \leq x < (z+1)^2 \ \wedge \ z - (m+1) < V\}$
$y := m + 1;$
$\{0 \leq y \leq z \ \wedge \ y^2 \leq x < (z+1)^2 \ \wedge \ z - y < V\}$
end $(*$ collect branches; definitions J and vf $*)$
$\{J \ \wedge \ \text{vf} < V\}$

university of groningen

Binary search: continued

Conclusion: binary search algorithm

const $x : \mathbb{N}$;
var $y,\ z,\ m : \mathbb{N}$;
$y := 0$; $z := x$;
  $\{J : 0 \leq y \leq z \ \wedge \ y^2 \leq x < (z+1)^2\}$
    $(*\ \mathrm{vf} = z - y\ *)$
while $y \neq z$ **do**
  $m := (y + z)$ **div** 2;
  if $(m+1) * (m+1) > x$ **then**
    $z := m$;
  else
    $y := m + 1$;
  endif ;
endwhile ;
  $\{Q : y^2 \leq x < (y+1)^2\}$

university of groningen

Binary search: continued

In Dafny, the binary search algorithm looks like this:

```
method Isqrt(x: nat) returns (y: nat)
ensures y*y <= x < (y+1)*(y+1)
{
  y := 0;
  var z := x;
  while y != z
  invariant 0 <= y <= z && y*y <= x < (z+1)*(z+1)
  decreases z - y
  {
    var m := (y + z)/2;
    if (m+1)*(m+1) > x {
      z := m;
    } else {
      y := m + 1;
    }
  }
}
```

university of groningen

Binary search: efficiency

Claim: The binary search algorithm needs no more than $\log x$ iterations.

[Note: usually, in computing science we mean with log the logarithm with base 2 (i.e. $\log_2 x$).]

Proof: Let r be the smallest integer such that $x \leq 2^r$.
Initially, $j = 0$ and $k = x$, so the size of the search interval is $\#[j, k) = k - j = x \leq 2^r$.

In each iteration of the loop the interval $[j, k)$ is replaced by the interval $[1 + (j + k)$ **div** $2, k)$ or $[j, (j + k)$ **div** $2)$.

It is clear that the loop is executed at most r times (i.e. $\log x$) if we can prove that $\#[1 + (j + k)$ **div** $2, k) \leq 2^{r-1}$ and $\#[j, (j + k)$ **div** $2) \leq 2^{r-1}$.

$\#[1 + (j + k)$ **div** $2, k) \leq 2^{r-1}$
$\equiv$ { size of half-open interval}
$k - 1 - (j + k)$ **div** $2 \leq 2^{r-1}$
$\equiv$ { calculus }
$k - 1 - 2^{r-1} \leq (j + k)$ **div** 2
$\equiv$ { rule: $z \leq x$ **div** $y \equiv z \cdot y \leq x$ }
$2(k - 1 - 2^{r-1}) \leq j + k$
$\equiv$ { calculus }
$k - j - 2 \leq 2^r$
$\equiv$ { $k - j \leq 2^r$ }
true

$\#[j, (j + k)$ **div** $2) \leq 2^{r-1}$
$\equiv$ { size of half-open interval}
$(j + k)$ **div** $2 - j \leq 2^{r-1}$
$\equiv$ { calculus }
$(j + k)$ **div** $2 < 1 + j + 2^{r-1}$
$\equiv$ { rule: x **div** $y < z \equiv x < z \cdot y$ }
$j + k < 2(1 + j + 2^{r-1})$
$\equiv$ { calculus }
$k - j - 2 < 2^r$
$\equiv$ { $k - j \leq 2^r$ }
true

university of groningen

How to find a good invariant?

Let P be the pre-condition and Q the post-condition.

Rule of thumb:
Use a predicate that can easily be initialized, and is obtained by *weakening* Q.

Informally, we could say that J is a predicate that is sort of 'in between' P and Q.

Next, choose a guard B such that $J \wedge \neg B \Rightarrow Q$.

In general, choosing a proper invariant requires experience and training.
But, there are a number of heuristics that can be helpful.

university of groningen

Heuristic invariant finding: Split conjuncts

If Q is of the form $Q_0 \ \wedge \ Q_1$ then we could try $J \equiv Q_0$ and $B \equiv \neg Q_1$ (or vice versa).

Clearly, we must be able to initialize J, and B must be a valid test (i.e. without specification constants).

Sometimes, Q is a single conjunct while it still can be expressed as two conjuncts. For example $x < y$ can be expressed as $x \leq y \ \wedge \ x \neq y$.

We actually used this heuristic already twice: in `Isqrt` and in `GTpow2`.

university of groningen

Example split conjuncts: integer square root

Recall the following specification:

const $x : \mathbb{N}$;
var $y : \mathbb{Z}$;
$\{P : \ \textbf{true}\}$
T
$\{Q : \ y \geq 0 \ \wedge \ y^2 \leq x < (y+1)^2\}$

We found the invariant (and guard) by conjunct-splitting:

$$\begin{aligned} J : &\quad y \geq 0 \ \wedge \ y^2 \leq x \\ B : &\quad (y+1)^2 \leq x \end{aligned}$$

university of groningen

Example split conjuncts: Powers of 2

Recall the following specification:

const $x : \mathbb{Z}$;
var $i, y : \mathbb{Z}$;
$\{P : x > 0\}$
T
$\{Q : x < y \leq 2 \cdot x \ \wedge \ y = 2^i\}$

We found the invariant (and guard) by conjunct-splitting:

$$J : \ y \leq 2 \cdot x \ \wedge \ y = 2^i$$

$$B : \ x \geq y$$

university of groningen

Heuristic invariant finding: Replace an expression by a variable

If Q contains some expression E then we could try to replace some (or all) occurrences of E in Q by a new variable i, such that $J \ \wedge \ i = E \Rightarrow Q$.

Next, we choose the guard $B : \ i \neq E$. Of course, B should not contain any specification constants.

Usually, it is a good practice to augment J with some conjunct that indicates which range of values i may attain. We call such a conjunct a *domain predicate*.

university of groningen

Heuristic invariant finding: Replace a constant by a variable

A special case of the heuristic "*replace an expression by a variable*" is the heuristic "*replace a constant by a variable*".

If Q contains some constant n then we could try to replace some (or all) occurrences of n in Q by a new variable i, such that $J \wedge i = n \Rightarrow Q$.

Clearly, we choose the guard $i \neq n$.

Again, it is good practice to augment J with a domain predicate.

We actually used this heuristic when we computed x^n using:

```
method Pow(x: int, n: nat) returns (y: int)
ensures y == exp(x,n)
{
  var i := 0;
  y := 1;
  while i != n
  invariant 0 <= i <= n && y == exp(x,i)
  {
    y := x*y;
    i := i + 1;
  }
}
```

university of groningen

Heuristic invariant finding: Split a variable

The heuristic called "*split a variable*" is actually also a special case of the heuristic "*replace an expression by a variable*".

If Q contains some variable k multiple times, then we could try to replace some (but not all) occurrences of k in Q by a new variable i, such that $J \wedge i = k \Rightarrow Q$.

Clearly, again we choose the guard $i \neq k$.

Again, it is good practice to augment J with a domain predicate.

We applied this heuristic in the binary search algorithm for integer square roots:

$$
\begin{array}{ll}
Q: & y \geq 0 \wedge y^2 \leq x < (y+1)^2 \\
J: & 0 \leq y \leq z \wedge y^2 \leq x < (z+1)^2 \\
B: & y \neq z
\end{array}
$$

university of groningen

Heuristic invariant finding: Generalization

Let P be of the form $P:\ X = E$, and Q of the form $Q:\ x = X$.

It is often the case that we can find a suitable invariant by generalizing E in the pre-condition by a more general expression E'.

- Example 1: $J:\ X = x + E$ where $\neg B \Rightarrow E = 0$.
- Example 2: $J:\ X = x \cdot E$ where $\neg B \Rightarrow E = 1$.

One could argue that this is actually the heuristic "Replace a constant by a variable" applied to the following (rewritten) pre-conditions:

- Example 1: $P:\ X = 0 + E$
- Example 2: $P:\ X = 1 \cdot E$

university of groningen

Examples: Generalization (Exponentiation)

const $x:\ \mathbb{Z}$;
var $y:\ \mathbb{Z},\ n:\ \mathbb{Z}$;
$\{P:\ n \geq 0\ \wedge\ x^n = Y\}$
S
$\{Q:\ y = Y\}$

We found the invariant (and guard) using generalization:

$$J:\ n \geq 0\ \wedge\ y \cdot x^n = Y$$
$$B:\ n \neq 0$$

```
method Pow(x: int, e: nat) returns (y: int)
ensures y == exp(x,n)
{
  var n := e;   // e is an in-parameter, so e := e - 1 is invalid
  y := 1;
  while n > 0
  invariant n >= 0 && y*exp(x,n) == exp(x,e)
  {
    y := y*x;
    n := n - 1;
  }
}
```

university of groningen

Examples: Generalization (Factorial)

We aim for a command S that satisfies:

$$
\begin{array}{l}
\textbf{var } x,\ n : \mathbb{Z}; \\
\quad \{P :\ n \geq 0 \ \wedge\ X = n!\} \\
S \\
\quad \{Q :\ x = X\}
\end{array}
$$

1. Choose an invariant J, and guard B such that $J \ \wedge\ \neg B \Rightarrow Q$.
   We use the heuristic *generalization*.

$$
\begin{array}{ll}
J : & x \cdot n! = X \ \wedge\ n \geq 0 \\
B : & n \neq 0
\end{array}
$$

   Clearly, J and $n = 0$ implies Q, since by definition $0! = 1$.

Examples: Generalization (Factorial)

2. Initialization: Find a command T_0 such that $\{P\}\ T_0\ \{J\}$

$$
\begin{array}{l}
\quad \{P : X = n! \ \wedge\ n \geq 0\} \\
\quad\quad (* \text{ calculus } *) \\
\quad \{X = 1 \cdot n! \ \wedge\ n \geq 0\} \\
x := 1; \\
\quad \{J :\ x \cdot n! = X \ \wedge\ n \geq 0\}
\end{array}
$$

3. Variant function: $\text{vf} \in \mathbb{Z}$ and $J \ \wedge\ B \Rightarrow \text{vf} \geq 0$
   Clearly, we must decrease n until $n = 0$. So, we simply choose $\text{vf} = n \in \mathbb{N}$. The invariant contains the conjunct $n \geq 0$, so trivially $J \ \wedge\ B \Rightarrow \text{vf} \geq 0$.

Examples: Generalization (Factorial)

4 Body of the loop: $\{J \wedge B \wedge \text{vf} = V\}\ S\ \{J \wedge \text{vf} < V\}$

$\{J \wedge B \wedge \text{vf} = V\}$
$\{x \cdot n! = X \wedge n \geq 0 \wedge n \neq 0 \wedge n = V\}$
$(* \ n = V > 0 \Rightarrow n! = n \cdot (n-1)! \wedge n - 1 \geq 0 \wedge n - 1 < V \ *)$
$\{x \cdot n \cdot (n-1)! = X \wedge n - 1 \geq 0 \wedge n - 1 < V\}$
$x := x * n;$
$\{x \cdot (n-1)! = X \wedge n - 1 \geq 0 \wedge n - 1 < V\}$
$n := n - 1;$
$\{x \cdot n! = X \wedge n \geq 0 \wedge n < V\}$
$\{J \wedge \text{vf} < V\}$

university of groningen

Examples: Generalization (Factorial)

5 Conclude that $\{P\}\ T_0;$ **while** B **do** S **end** $\{Q\}$ solves the problem.

var $x,\ n : \mathbb{Z};$
$\{P : X = n! \wedge n \geq 0\}$
$x := 1;$
$\{J : x \cdot n! = X \wedge n \geq 0\}$
$(* \ \text{vf} = n \ *)$
while $n \neq 0$ **do**
$x := x * n;$
$n := n - 1;$
end;
$\{Q : x = X\}$

university of groningen

Factorial (in Dafny)

```
include "io.dfy"

function factorial(n: nat): nat {
  if n == 0 then 1 else n*factorial(n-1)
}

method Factorial(m: nat) returns (x: nat)
ensures  x == factorial(m)
{
  var n := m;
  x := 1;
  while n != 0
  invariant 0<=n && factorial(m)==x*factorial(n)
  {
    x := x*n;
    n := n - 1;
  }
}

method Main()
{
  var n := IO.ReadNat();
  var f := Factorial(n);
  print n, "! = ", f, "\n";
}
```

Dafny reports

```
Dafny program verifier finished with 3 verified, 0 errors
```


Example: recurrence to iteration

The function f is defined by the recurrence:

$$\begin{aligned} f(0) &= 0 \\ n > 0 \Rightarrow f(n) &= 5 \cdot f(n \textbf{ div } 3) + n \textbf{ mod } 4 \end{aligned}$$

Find a command S that satisfies the following specification:

$$\begin{array}{l} \textbf{var } n,\ x : \mathbb{Z}; \\ \quad \{P :\ n \geq 0\ \wedge\ Z = f(n)\} \\ S \\ \quad \{Q :\ Z = x\} \end{array}$$

Invariant and guard: We introduce a variable y.

$$J :\ Z = y \cdot f(n) + x\ \wedge\ n \geq 0$$

$$B :\ n \neq 0$$

$$\begin{array}{ll} & J\ \wedge\ \neg B \\ \equiv & \\ & Z = y \cdot f(n) + x\ \wedge\ n = 0 \\ \Rightarrow & \{\ f(0) = 0;\ \text{logic}\ \} \\ & Z = y \cdot 0 + x \\ \equiv & \{\ \text{calculus}\ \} \\ & Q : Z = x \end{array}$$

university of groningen

Recurrence to iteration (Continued)

$$J: \ Z = y \cdot f(n) + x \ \wedge \ n \geq 0$$

Initialization of the invariant is easy:

$$\begin{array}{l} \{P: Z = f(n) \ \wedge \ n \geq 0\} \\ \quad (* \text{ calculus } *) \\ \quad \{Z = 1 \cdot f(n) + 0 \ \wedge \ n \geq 0\} \\ x, y := 0, 1; \\ \quad \{J: Z = y \cdot f(n) + x \ \wedge \ n \geq 0\} \end{array}$$

Next, we need to choose a vf $\in \mathbb{Z}$. The invariant ensures that $n \geq 0$, and we chose $B: n \neq 0$. We will decrease n until $n = 0$. So, we choose vf $= n \in \mathbb{Z}$. Clearly, $J \ \wedge \ B \Rightarrow$ vf ≥ 0, since $vf \geq 0$ is a conjunct of J.

university of groningen

Recurrence to iteration (Continued)

$$\begin{array}{rcl} f(0) & = & 0 \\ n > 0 \Rightarrow f(n) & = & 5 \cdot f(n \ \textbf{div} \ 3) + n \ \textbf{mod} \ 4 \end{array}$$

For the body of the loop we find:

$$\begin{array}{l} \{J \ \wedge \ B \ \wedge \ \text{vf} = V\} \\ \{Z = y \cdot f(n) + x \ \wedge \ n \geq 0 \ \wedge \ n \neq 0 \ \wedge \ n = V\} \\ \quad (* \ n > 0 \Rightarrow f(n) = 5 \cdot f(n \ \textbf{div} \ 3) + n \ \textbf{mod} \ 4 \ \wedge \ 0 \leq n \ \textbf{div} \ 3 < n \ *) \\ \{Z = y \cdot (5 \cdot f(n \ \textbf{div} \ 3) + n \ \textbf{mod} \ 4) + x \ \wedge \ 0 \leq n \ \textbf{div} \ 3 < V\} \\ \quad (* \text{ calculus } *) \\ \{Z = 5 \cdot y \cdot f(n \ \textbf{div} \ 3) + y \cdot (n \ \textbf{mod} \ 4) + x \ \wedge \ 0 \leq n \ \textbf{div} \ 3 < V\} \\ x := y * (n \ \textbf{mod} \ 4) + x; \\ \{Z = 5 \cdot y \cdot f(n \ \textbf{div} \ 3) + x \ \wedge \ 0 \leq n \ \textbf{div} \ 3 < V\} \\ y := 5 * y; \\ \{Z = y \cdot f(n \ \textbf{div} \ 3) + x \ \wedge \ 0 \leq n \ \textbf{div} \ 3 < V\} \\ n := n \ \textbf{div} \ 3; \\ \{Z = y \cdot f(n) + x \ \wedge \ 0 \leq n < V\} \\ \{J \ \wedge \ \text{vf} < V\} \end{array}$$

university of groningen

Recurrence to iteration (Continued)

In conclusion, we derived the following program fragment:

$$
\begin{array}{l}
\textbf{var } x,\ y,\ n:\ \mathbb{Z};\\
\quad \{P:\ Z = f(n)\ \wedge\ n \geq 0\}\\
x, y := 0, 1;\\
\quad \{J:\ Z = y \cdot f(n) + x\ \wedge\ n \geq 0\}\\
\qquad (* \text{ vf} = n *)\\
\textbf{while } n \neq 0 \textbf{ do}\\
\quad x := y * (n \textbf{ mod } 4) + x;\\
\quad y := 5 * y;\\
\quad n := n \textbf{ div } 3;\\
\textbf{end};\\
\quad \{Q:\ x = Z\}
\end{array}
$$

university of groningen

Recurrence to iteration (Dafny version)

```
function f(n: nat): nat {
  if n == 0 then 0 else 5*f(n/3) + n%4
}

method computeF(m: nat) returns (x: nat)
ensures  x == f(m)
{
  var n, y;
  n,x,y := m,0,1;
  while n != 0
  invariant 0<=n && f(m)==y*f(n) + x
  {
    x := x + y*(n%4);
    y := 5*y;
    n := n/3;
  }
}
```

university of groningen

Another example: From recurrence to a while-loop

The function f is defined by the recurrence:

$$\begin{array}{lcl} y \leq 0 & \Rightarrow & f(y,z) = z \\ y > 0 & \Rightarrow & f(y,z) = 10 \cdot f(y \textbf{ div } 10, z) + y \textbf{ mod } 10 \end{array}$$

Find a command S that satisfies the specification:

$$\begin{array}{l} \textbf{var } y,\ z :\ \mathbb{Z}; \\ \quad \{P :\ Z = f(y,z)\} \\ S \\ \quad \{Q :\ Z = z\} \end{array}$$

We introduce auxiliary variables m and n, and $J :\ Z = m \cdot f(y,z) + n$
Initialization is easy:

$$\begin{array}{l} \quad \{P : Z = f(y,z)\} \\ \quad\quad (* \text{ calculus } *) \\ \quad \{Z = 1 \cdot f(y,z) + 0\} \\ m := 1;\ n := 0; \\ \quad \{J :\ Z = m \cdot f(y,z) + n\} \end{array}$$

university of groningen

Example: From recurrence to a while-loop (Continued)

$$\begin{array}{lcl} y \leq 0 & \Rightarrow & f(y,z) = z \\ y > 0 & \Rightarrow & f(y,z) = 10 \cdot f(y \textbf{ div } 10, z) + y \textbf{ mod } 10 \\ J :\ Z & = & m \cdot f(y,z) + n \\ Q :\ Z & = & z \end{array}$$

The next step is to choose the guard B.

From the definition of f we see that we can determine $f(y,z)$ directly if $y \leq 0$.
Therefore, we choose the guard $B : y > 0$.

We need (active) finalization to establish Q:

$$\begin{array}{l} \quad \{J \ \wedge\ \neg B\} \\ \quad \{Z = m \cdot f(y,z) + n \ \wedge\ y \leq 0\} \\ \quad\quad (* \text{ definition } f *) \\ \quad \{Z = m \cdot z + n\} \\ z := m * z + n; \\ \quad \{Z = z\} \end{array}$$

university of groningen

Example: From recurrence to a while-loop (Continued)

variant function: The guard is $y > 0$, so we need to decrease y until $y \leq 0$.
We choose vf $= y \in \mathbb{Z}$.
Since $B \equiv \text{vf} > 0$, it is clear that $J \wedge B \Rightarrow \text{vf} \geq 0$.

For the body of the loop we find:

$\{J \wedge B \wedge \text{vf} = V\}$
$\{Z = m \cdot f(y,z) + n \wedge y > 0 \wedge y = V\}$
$(* \; y = V > 0 \Rightarrow y \; \textbf{div} \; 10 < V$; definition $f \; *)$
$\{Z = m \cdot (10 \cdot f(y \; \textbf{div} \; 10, z) + y \; \textbf{mod} \; 10) + n \wedge y \; \textbf{div} \; 10 < V\}$
$(*$ calculus $*)$
$\{Z = 10 \cdot m \cdot f(y \; \textbf{div} \; 10, z) + m \cdot (y \; \textbf{mod} \; 10) + n \wedge y \; \textbf{div} \; 10 < V\}$
$n := m * (y \; \textbf{mod} \; 10) + n;$
$\{Z = 10 \cdot m \cdot f(y \; \textbf{div} \; 10, z) + n \wedge y \; \textbf{div} \; 10 < V\}$
$m := 10 * m;$
$\{Z = m \cdot f(y \; \textbf{div} \; 10, z) + n \wedge y \; \textbf{div} \; 10 < V\}$
$y := y \; \textbf{div} \; 10;$
$\{Z = m \cdot f(y,z) + n \wedge y < V\}$
$\{J \wedge \text{vf} < V\}$

university of groningen

Example: From recurrence to a while-loop (Continued)

Conclusion: we derived the following program fragment:

var $n, m, y, z : \mathbb{Z};$
$\{P : Z = f(y,z)\}$
$m := 1;$
$n := 0;$
$\{J : Z = m \cdot f(y,z) + n\}$
$(* \; \text{vf} = y \; *)$
while $y > 0$ **do**
$n := m * (y \; \textbf{mod} \; 10) + n;$
$m := 10 * m;$
$y := y \; \textbf{div} \; 10;$
end;
$z := m * z + n;$
$\{Q : z = Z\}$

university of groningen

Example: From recurrence to a while-loop (Dafny)

Let us try to convert the program fragment into a Dafny program.
A first attempt might look like this:

```
method ComputeF(y: int, z: int) returns (z: int)
ensures z == f(y,z)
{
  var m,n := 1,0;
  while y > 0
  {
    n := m*(y%10) + n;
    m := 10*m;
    y := y/10;
  }
  z := m*z + n;
}
```

Dafny reports

```
Error:  Duplicate parameter name:  z
```

university of groningen

Example: From recurrence to a while-loop (Dafny)

Ok, so we introduce a variable `r` to return the result.

```
method ComputeF(y: int, z: int) returns (r: int)
ensures z == f(y,z)
{
  var m,n := 1,0;
  while y > 0
  {
    n := m*(y%10) + n;
    m := 10*m;
    y := y/10;
  }
  r := m*z + n;
}
```

Dafny reports

```
Error:  LHS of assignment must denote a mutable variable
```

The problem is that `y` is an in-parameter, and cannot be modified in the body of the method.

university of groningen

Example: From recurrence to a while-loop (Dafny)

So, we end up with:

```
ghost function f(y: int, z: int): int
{
  if y <= 0 then z else 10*f(y/10, z) + y%10
}
method ComputeF(y: int, z: int) returns (r: int)
ensures r == f(y,z)
{
  var m,n,a := 1,0,y;
  while a > 0
  invariant f(y,z) == m*f(a,z) + n
  {
    n := m*(a%10) + n;
    m := 10*m;
    a := a/10;
  }
  r := m*z + n;
}
```

Dafny reports

```
Dafny program verifier finished with 2 verified, 0 errors
```


Exercise: Mid exam Program Correctness 2015

The function $f : \mathbb{N} \to \mathbb{N}$ is defined recursively as follows:

$$\begin{array}{rcl} n = 0 & \Rightarrow & f(n) = 1 \\ n > 0 \ \wedge \ n \ \textbf{mod}\ 2 = 0 & \Rightarrow & f(n) = 1 + 2 \cdot f(n \ \textbf{div}\ 2) \\ n > 0 \ \wedge \ n \ \textbf{mod}\ 2 = 1 & \Rightarrow & f(n) = 2 \cdot f(n \ \textbf{div}\ 2) \end{array}$$

Design a command S that satisfies $\{\ P :\ n \geq 0 \ \wedge \ X = f(n)\ \}\ S\ \{\ Q :\ X = a\ \}$

1. Invariant and guard: we choose the invariant $J :\quad X = a + b \cdot f(n) \ \wedge \ n \geq 0$.
   The recursion stops if $n = 0$, so we choose the guard $n \neq 0$.
   We will need active finalization to establish Q.

   $\{\ J \ \wedge \ \neg B\ \}$
   $\{\ n = 0 \ \wedge \ X = a + b \cdot f(n)\ \}$
   $\{\ X = a + b\ \}$
   $a := a + b;$
   $\{\ Q :\ X = a\ \}$

2. Initialization:

   $\{\ P :\ n \geq 0 \ \wedge \ X = f(n)\ \}$
   $\{\ n \geq 0 \ \wedge \ X = 0 + 1 \cdot f(n)\ \}$
   $a := 0;\ b := 1;$
   $\{\ J :\ n \geq 0 \ \wedge \ X = a + b \cdot f(n)\ \}$

3. Variant function: Choose vf $= n \in \mathbb{Z}$. Clearly, we have $J \ \wedge \ B \Rightarrow$ vf ≥ 0.

university of groningen

Exercise: Mid exam Program Correctness 2015 (Continued)

4 Body of the loop:

$\{\ X = a + b \cdot f(n)\ \wedge\ n = V > 0\ \}$
if n **mod** $2 = 0$ **then**
  $\{\ n$ **mod** $2 = 0\ \wedge\ X = a + b \cdot f(n)\ \wedge\ n = V > 0\ \}$
    $(*\ n > 0\ \wedge\ n$ **mod** $2 = 0 \Rightarrow f(n) = 1 + 2 \cdot f(n$ **div** $2)\ *)$
  $\{\ X = a + b \cdot (1 + 2 \cdot f(n$ **div** $2))\ \wedge\ n = V > 0\ \}$
  $\{\ X = a + b + 2 \cdot b \cdot f(n$ **div** $2)\ \wedge\ n = V > 0\ \}$
  $a := a + b;$
  $\{\ X = a + 2 \cdot b \cdot f(n$ **div** $2)\ \wedge\ n = V > 0\ \}$
else
  $\{\ n$ **mod** $2 = 1\ \wedge\ X = a + b \cdot f(n)\ \wedge\ n = V > 0\ \}$
    $(*\ n > 0\ \wedge\ n$ **mod** $2 = 1 \Rightarrow f(n) = 2 \cdot f(n$ **div** $2)\ *)$
  $\{\ X = a + 2 \cdot b \cdot f(n$ **div** $2)\ \wedge\ n = V > 0\ \}$
  skip;
  $\{\ X = a + 2 \cdot b \cdot f(n$ **div** $2)\ \wedge\ n = V > 0\ \}$
end; $(*$ collect branches $*)$
  $\{\ X = a + 2 \cdot b \cdot f(n$ **div** $2)\ \wedge\ n = V > 0\ \}$
$b := 2 * b;$
  $\{\ X = a + b \cdot f(n$ **div** $2)\ \wedge\ n = V > 0\ \}$
    $(*$ calculus; $n > 0 \Rightarrow 0 \leq n$ **div** $2 < n\ *)$
  $\{\ X = a + b \cdot f(n$ **div** $2)\ \wedge\ n$ **div** $2 \geq 0\ \wedge\ n$ **div** $2 < V\ \}$
$n := n$ **div** $2;$
  $\{\ X = a + b \cdot f(n)\ \wedge\ n \geq 0\ \wedge\ n < V\ \}$

university of groningen

Exercise: Mid exam Program Correctness 2015 (Continued)

var $a,\ b,\ n\ :\ \mathbb{N};$
  $\{\ P:\ n \geq 0\ \wedge\ X = f(n)\ \}$
$a := 0;\ b := 1;$
  $\{\ J:\ X = a + b \cdot f(n)\ \wedge\ n \geq 0\ \}$
    $(*$ vf : $n\ *)$
while $n \neq 0$ **do**
  if n **mod** $2 = 0$ **then**
    $a := a + b;$
  end;
  $b := 2 * b;$
  $n := n$ **div** $2;$
end;
$a := a + b;$
  $\{\ Q:\ X = a\ \}$

university of groningen

Example: Mid exam Program Correctness 2015 (in Dafny)

```
include "io.dfy"

function f(n: nat): nat {
  if n == 0 then 1
  else if n%2 == 0 then 1 + 2*f(n/2)
       else 2*f(n/2)
}

method F(m: nat) returns (a: nat)
ensures  a == f(m)
{
  var n, b: nat;
  a,b,n := 0,1,m;
  while n != 0
  invariant 0<=n && f(m)==a + b*f(n)
  {
    if n%2==0 {
      a := a + b;
    }
    b := 2*b;
    n := n/2;
  }
  a := a + b;
}

method Main() {
  var n := IO.ReadNat();
  var f := F(n);
  print "f(", n, ")=", f, ".\n";
}
```

university of groningen

Fibonacci (once more)

The Fibonacci numbers are defined by the recurrence relation:

$$\begin{aligned} f(0) &= 0 \\ f(1) &= 1 \\ f(n) &= f(n-2) + f(n-1) \quad \text{for } n > 1 \end{aligned}$$

We aim for a command S that satisfies: $\{P : n \geq 0\}\ S\ \{Q : x = f(n)\}$

It seems natural to aim for a loop in which a variable k runs from 0 to n which keeps $x = f(k) \wedge y = f(k-1)$ invariant.
In that case $f(k+1) = f(k-1) + f(k) = y + x$.

However, this is difficult to initialize.
If we initialize with $k := 0$ then we need $f(-1)$, which is undefined.

A better choice for the invariant is $J :\ 0 \leq k \leq n\ \wedge\ x = f(k)\ \wedge\ y = f(k+1)$.

Clearly, Q holds if $J\ \wedge\ k = n$, so we choose $B : k < n$ (or $k \neq n$).

Next, we choose a variant function.
The guard suggests to choose vf $= n - k \in \mathbb{Z}$.
The requirement vf ≥ 0 follows trivially from B.

university of groningen

Fibonacci (once more)

$$J : 0 \leq k \leq n \ \wedge \ x = f(k) \ \wedge \ y = f(k+1)$$

Initialization is easy: $k := 0$; $x := 0$; $y := 1$;

For the body of the loop we find:

$\{0 \leq k < n \ \wedge \ x = f(k) \ \wedge \ y = f(k+1) \ \wedge \ n - k = V\}$
$(* \text{ prepare } k := k+1 *)$
$\{0 < k+1 \leq n \ \wedge \ x = f(k+1-1) \ \wedge \ y = f(k+1) \ \wedge \ n - (k+1) < V\}$
$k := k+1;$
$\{0 < k \leq n \ \wedge \ x = f(k-1) \ \wedge \ y = f(k) \ \wedge \ n - k < V\}$
$(* \ k > 0 \Rightarrow f(k+1) = f(k-1) + f(k) = x + y *)$
$\{0 < k \leq n \ \wedge \ x = f(k-1) \ \wedge \ x + y = f(k+1) \ \wedge \ n - k < V\}$
$y := x + y;$
$\{0 < k \leq n \ \wedge \ x = f(k-1) \ \wedge \ y = f(k+1) \ \wedge \ n - k < V\}$
$(* \ k > 0 \Rightarrow f(k) = f(k+1) - f(k-1) = y - x *)$
$\{0 \leq k \leq n \ \wedge \ y - x = f(k) \ \wedge \ y = f(k+1) \ \wedge \ n - k < V\}$
$x := y - x;$
$\{0 \leq k \leq n \ \wedge \ x = f(k) \ \wedge \ y = f(k+1) \ \wedge \ n - k < V\}$

university of groningen

Fibonacci (Continued)

In conclusion, we derived the following program fragment:

const $n : \mathbb{Z}$;
var $k, x, y : \mathbb{Z}$;
$\{P : n \geq 0\}$
$k := 0; x := 0; y := 1;$
$\{J : 0 \leq k \leq n \ \wedge \ x = f(k) \ \wedge \ y = f(k+1)\}$
$(* \text{ vf} = n - k *)$
while $k < n$ **do**
$k := k+1;$
$y := x + y;$
$x := y - x;$
end;
$\{Q : x = f(n)\}$

university of groningen

Fibonacci (in Dafny)

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

method Fib(n: nat) returns (x: nat)
ensures  x == fib(n)
{
  var k,y;
  k,x,y := 0,0,1;
  while k < n
  invariant 0<=k<=n && x==fib(k) && y==fib(k+1)
  {
    k := k + 1;
    y := x + y;
    x := y - x;
  }
}
```

Dafny reports

```
Dafny program verifier finished with 3 verified, 0 errors
```


Dijkstra's Fusc function

$$\begin{array}{rcl} f(0) = 0 & \wedge & f(1) = 1 \\ f(2 \cdot n) & = & f(n) \\ f(2 \cdot n + 1) & = & f(n) + f(n+1) \end{array}$$

We aim for a program fragment that satisfies the following specification:

var $n,\ x : \mathbb{Z}$;
$\{P : n \geq 0 \ \wedge \ Z = f(n)\}$
S;
$\{Q : x = Z\}$

1. Invariant and guard:

$$\begin{array}{ll} J: & n \geq 0 \ \wedge \ Z = y \cdot f(n) + x \cdot f(n+1) \\ B: & n \neq 0 \end{array}$$

university of groningen

Dijkstra's Fusc function (continued)

$$\begin{array}{rcl} f(0) = 0 & \wedge & f(1) = 1 \\ f(2 \cdot n) & = & f(n) \\ f(2 \cdot n + 1) & = & f(n) + f(n+1) \end{array}$$

2 Initialization: Find a command T_0 such that
$\{n \geq 0 \ \wedge \ Z = f(n)\} \ T_0 \ \{n \geq 0 \ \wedge \ Z = y \cdot f(n) + x \cdot f(n+1)\}$

$\{P : n \geq 0 \ \wedge \ Z = f(n)\}$
$(*$ calculus $*)$
$\{n \geq 0 \ \wedge \ Z = 1 \cdot f(n) + 0 \cdot f(n+1)\}$
$y := 1; \ x := 0;$
$\{J : n \geq 0 \ \wedge \ Z = y \cdot f(n) + x \cdot f(n+1)\}$

3 Variant function: vf $= n \in \mathbb{Z}$.
The invariant contains the conjunct $n \geq 0$, so $J \ \wedge \ B \Rightarrow \text{vf} \geq 0$.

university of groningen

Fusc: [4] loop body $\{J \ \wedge \ B \ \wedge \ \text{vf} = V\} \ S \ \{J \ \wedge \ \text{vf} < V\}$

$\{0 < n = V \ \wedge \ Z = y \cdot f(n) + x \cdot f(n+1)\}$
if n **mod** $2 = 0$ **then**
$\{2 \leq n = V \ \wedge \ Z = y \cdot f(2(n \ \textbf{div} \ 2)) + x \cdot f(2(n \ \textbf{div} \ 2) + 1)\}$
$(*$ logic; definition $f(n)$ for $n \geq 1 \ *)$
$\{0 < n = V \ \wedge \ Z = y \cdot f(n \ \textbf{div} \ 2) + x \cdot (f(n \ \textbf{div} \ 2) + f(n \ \textbf{div} \ 2 + 1))\}$
$\{0 < n = V \ \wedge \ Z = (x + y) \cdot f(n \ \textbf{div} \ 2) + x \cdot f(n \ \textbf{div} \ 2 + 1)\}$
$y := x + y;$
$\{0 < n = V \ \wedge \ Z = y \cdot f(n \ \textbf{div} \ 2) + x \cdot f(n \ \textbf{div} \ 2 + 1)\}$
else
$\{0 < n = V \ \wedge \ Z = y \cdot f(2(n \ \textbf{div} \ 2) + 1) + x \cdot f(2(n \ \textbf{div} \ 2) + 2)\}$
$\{0 < n = V \ \wedge \ Z = y \cdot f(2(n \ \textbf{div} \ 2) + 1) + x \cdot f(2(n \ \textbf{div} \ 2 + 1))\}$
$(*$ definition $f(n)$; Note: $f(1) = 1 = 0 + 1 = f(0) + f(0+1) = f(2 \cdot 0 + 1) \ *)$
$\{0 < n = V \ \wedge \ Z = y \cdot (f(n \ \textbf{div} \ 2) + f(n \ \textbf{div} \ 2 + 1)) + x \cdot f(n \ \textbf{div} \ 2 + 1)\}$
$\{0 < n = V \ \wedge \ Z = y \cdot f(n \ \textbf{div} \ 2) + (x + y) \cdot f(n \ \textbf{div} \ 2 + 1)\}$
$x := x + y;$
$\{0 < n = V \ \wedge \ Z = y \cdot f(n \ \textbf{div} \ 2) + x \cdot f(n \ \textbf{div} \ 2 + 1)\}$
end; $(*$ collect branches $*)$
$\{0 < n = V \ \wedge \ Z = y \cdot f(n \ \textbf{div} \ 2) + x \cdot f(n \ \textbf{div} \ 2 + 1)\}$
$(* \ 0 < n = V \Rightarrow 0 \leq n \ \textbf{div} \ 2 < V \ *)$
$\{0 \leq n \ \textbf{div} \ 2 < V \ \wedge \ Z = y \cdot f(n \ \textbf{div} \ 2) + x \cdot f(n \ \textbf{div} \ 2 + 1)\}$
$n := n \ \textbf{div} \ 2;$
$\{0 \leq n < V \ \wedge \ Z = y \cdot f(n) + x \cdot f(n+1)\}$
$\{J \ \wedge \ \text{vf} < V\}$

university of groningen

Dijkstra's Fusc function (continued)

5 Conclusion: We derived the following program:

var $n,\ x,\ y : \mathbb{Z}$;
  $\{P : n \geq 0 \ \wedge \ Z = f(n)\}$
$y := 1$;
$x := 0$;
  $\{J : \ n \geq 0 \ \wedge \ Z = y \cdot f(n) + x \cdot f(n+1)\}$
    $(* \ vf = n \ *)$
while $n \neq 0$ **do**
  if n **mod** $2 = 0$ **then**
      $y := x + y$;
  else
      $x := x + y$;
  end;
  $n := n$ **div** 2;
end;
  $\{Q : x = Z\}$

Fusc (in Dafny)

$$\begin{array}{rcl} f(0) = 0 & \wedge & f(1) = 1 \\ f(2 \cdot n) & = & f(n) \\ f(2 \cdot n + 1) & = & f(n) + f(n+1) \end{array}$$

We first need to rewrite the definition of the function `fusc`:

```
function fusc(n: nat): nat {
  if n <= 1 then n
     else if n%2 == 0
          then fusc(n/2)
          else fusc(n/2) + fusc(1 + n/2)
}
```

Fusc (in Dafny)

```
method {:timeLimit 40} Fusc(m: nat) returns (x: nat)
ensures  x == fusc(m)
{
  var n := m;
  var y := 1;
  x := 0;
  while n != 0
  invariant 0<=n && fusc(m)==y*fusc(n) + x*fusc(n+1)
  {
    if n%2 == 0 {
      y := x + y;
    } else {
      x := x + y;
    }
    n := n/2;
  }
}
```

Dafny reports

```
Dafny program verifier finished with 3 verified, 0 errors
```


Lemmas and Proofs

So far, most programs were accepted by Dafny without much effort.
However, for more complicated programs the verifier needs our help.

This can be done by writing *lemmas*, which you can use in a larger proof.

Consider the following function, which returns something larger than its argument:

```
function More(x: int): int {
  if x <= 0 then 1 else More(x - 2) + 3
}
```

Next, we introduce the lemma `Increasing`:

```
lemma Increasing(x: int)
ensures x < More(x)
{
}
```

The body of a lemma may help Dafny to complete the proof. In this particular case, Dafny is able to prove the lemma itself, so we can leave the body empty.

university of groningen

Lemmas and Proofs

Next, consider the following `method UseLemma`:

```
method UseLemma(a: int) {
  var b := More(a);
  var c := More(b);
  assert 2 <= c - a;
}
```

Clearly, the method computes `More(More(a))` into variable `c`.
Since `More(a) >= a + 1`, we must surely have `More(More(a)) >= a+2`.

Still, Dafny fails to verify the program.
The reason is that the verifier does know the definition of `More` but not its properties.

In particular, it needs the `Increasing` lemma:

```
method UseLemma(a: int) {
  var b:= More(a);
  Increasing(a);
  Increasing(b);
  var c := More(b);
  assert 2 <= c - a;
}
```

university of groningen

Lemmas and Proofs

Why did the verifier accept the latter version? Let us zoom in:

```
function More(x: int): int {
  if x <= 0 then 1 else More(x - 2) + 3
}

lemma Increasing(x: int)
ensures x < More(x)
{
}

method UseLemma(a: int) {
  var b:= More(a);  // Dafny knows that b == More(a)
  Increasing(a);    // Dafny knows that More(a) > a, so b > a
  Increasing(b);    // Dafny knows that More(b) > b > a
  var c := More(b); // Dafny knows that c == More(b) > b > a
  assert 2 <= c - a; // c > b > a, so c - a >= 2
}
```

university of groningen

Proving a lemma

Dafny was able to prove the `Increasing` lemma, because it used *automatic induction*.

You can turn automatic induction off:

```
function More(x: int): int {
  if x <= 0 then 1 else More(x - 2) + 3
}

lemma {:induction false} Increasing(x: int)
ensures x < More(x)
{
}
```

Dafny reports

```
Error:  a postcondition could not be proved on this return path
```

university of groningen

Proving a lemma

We need to help the verifier. We start with the following attempt:

```
lemma {:induction false} Increasing(x: int)
ensures x < More(x)
{
   if x <= 0 {
     // this case verifies without help (base case)
   } else {
     // this case needs our help (inductive case)
   }
}
```

Dafny reports

```
line 7:  a postcondition could not be proved on this return path
```

university of groningen

Proving a lemma

We need to help the verifier. We start with the following attempt:

```
lemma {:induction false} Increasing(x: int)
ensures x < More(x)
{
    if x <= 0 {
      // this case verifies without help (base case)
    } else {
      // x > 0  implies More(x) == More(x - 2) + 3
      // so x < More(x) is the same as x - 3 < More(x - 2)
      // that follows clearly from x - 2 < More(x - 2)
      Increasing(x-2);
    }
}
```

Dafny reports

```
Dafny program verifier finished with 0 errors
```

This completes the proof, but we need to realize that Dafny still took care of a detail.

Dafny used the `decreases x` clause to prove termination.

university of groningen

Proving a lemma (termination)

Proving termination is important, because it prevents proof attempts like this one:

```
lemma {:induction false} Increasing(x: int)
ensures x < More(x)
{
  Increasing(x); // this gives us x < More(x)
}
```

Dafny reports

```
cannot prove termination; try supplying a decreases clause
```

university of groningen

Proving a lemma (termination)

In Dafny, we can 'program' a proof:

```
lemma {:induction false} Increasing(x: int)
ensures x < More(x)
{
    if x > 0 {
      var y := x - 2;
      Increasing(y);
    }
}
```

Dafny reports

```
Dafny program verifier finished with 0 errors
```

university of groningen

Proof calculations

Dafny also supports calculations within a proof. They look like this:

```
calc {
     5*(x + 3);    // note the semicolon!
  == // distribute * over +
     5*x + 5*3;
  == // calculus
     5*x + 15;
}
```

Here is another example (`n` is of type `nat`):

```
calc {
     3*x + n + n;
  == // n + n == 2*n
     3*x + 2*n;
  <= // 2*n <= 3*n, since n >= 0
     3*x + 3*n;
   == // distribute * over +
     3*(x + n);
}
```

university of groningen

Proof calculations

A proof using `calc` looks like this:

```
lemma {:induction false} Increasing(x: int)
ensures x < More(x)
{
  if x <= 0 {
    calc {
         x;
      <= // guard says x <= 0
         0;
      <  // arithmetic
         1;
      == // definition More for x <= 0
         More(x);
     }
  } else {
    calc {
         More(x);
      == // definition More for x > 0
         More(x - 2) + 3;
      >  { Increasing(x-2); }   // hints to Dafny in curly braces
         x - 2 + 3;
      >  // calculus
         x;
     }
  }
}
```

university of groningen

Proof calculations

Here is a variation on the proof:

```
lemma {:induction false} Increasing(x: int)
ensures x < More(x)
{
  if x <= 0 {
    calc {
         x;
      <= // guard says x <= 0
         0;
      <  // arithmetic
         1;
      == // definition More for x <= 0
         More(x);
     }
  } else {
     calc {
         x < More(x);
      <== // calculus
         x + 1 < More(x);
      ==  // definition More for x > 0
         x + 1 < More(x - 2) + 3;
      ==  // calculus
         x - 2 < More(x - 2);
      <==  Increasing(x - 2);
         true;
     }
 }
```

university of groningen

Wishful thinking: Fibonacci squared (Dafny)

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

We aim for the following method:

```
method SquaredFib(N: nat) returns (x: nat)
ensures  x == fib(N)*fib(N)
{
  x := 0;
  var n := 0;
  while n != N
  invariant 0<=n<=N
  invariant x == fib(n)*fib(n)
  {
    ??????
    assert x == fib(n+1)*fib(n+1);
    n := n + 1;
    assert x == fib(n)*fib(n);
  }
}
```

Let us *wish* that we have `fib(n+1)` already computed in a variable `y`.

university of groningen

Fibonacci squared (Dafny)

```
method SquaredFib(N: nat) returns (x: nat)
ensures  x == fib(N)*fib(N)
{
  x := 0;
  var n := 0;
  while n != N
  invariant 0<=n<=N
  invariant x == fib(n)*fib(n)
  invariant y == fib(n + 1)*fib(n + 1)
  {
    x,y := y, ??;
    assert x == fib(n+1)*fib(n+1) && y == fib(n + 2)*fib(n + 2)
    n := n + 1;
    assert x == fib(n)*fib(n) && y == fib(n + 1)*fib(n + 1)
  }
}
```

university of groningen

Fibonacci squared (Dafny)

It is time for some calculation:

```
while n != N
invariant 0<=n<=N
invariant x == fib(n)*fib(n)
invariant y == fib(n + 1)*fib(n + 1)
{
  calc {
    fib(n+2)*fib(n+2);
  == // definition fib
    (fib(n)+fib(n+1))*(fib(n)+fib(n+1));
  == // calculus
    fib(n)*fib(n) + 2*fib(n)*fib(n+1) + fib(n+1)*fib(n+1);
  == // invariant
    x + 2*fib(n)*fib(n+1) + y;
  }
  x,y := y, x + ?? + y;
  assert x == fib(n+1)*fib(n+1) && y == fib(n + 2)*fib(n + 2)
  n := n + 1;
  assert x == fib(n)*fib(n) && y == fib(n + 1)*fib(n + 1)
}
```

university of groningen

Ok, we have another *wish*: `k==2*fib(n)*fib(n + 1)`.

Fibonacci squared (Dafny)

```
while n != N
invariant 0<=n<=N
invariant x == fib(n)*fib(n)
invariant y == fib(n + 1)*fib(n + 1)
invariant k == 2*fib(n)*fib(n + 1)
{
  calc {
    2*fib(n+1)*fib(n+2);
  == // definition fib
    2*fib(n+1)*(fib(n) + fib(n+1));
  == // calculus
    2*fib(n+1)*fib(n) + 2*fib(n+1)*fib(n+1);
  == // invariant
    k + 2*y;
  }
  calc {
    fib(n+2)*fib(n+2);
  == // definition fib
    (fib(n)+fib(n+1))*(fib(n)+fib(n+1));
  == // calculus
    fib(n)*fib(n) + 2*fib(n)*fib(n+1) + fib(n+1)*fib(n+1);
  == // invariant
    x + k + y;
  }
  x,y,k := y, x + k + y, k + 2*y;
  n := n + 1;
}
```

university of groningen

Fibonacci squared (Dafny)

In conclusion, we have found the following method implementation:

```
method SquaredFib(N: nat) returns (x: nat)
ensures  x == fib(N)*fib(N)
{
  x := 0;
  var n, y, k := 0, 1, 0;
  while n != N
  invariant 0<=n<=N
  invariant x == fib(n)*fib(n)
  invariant y == fib(n + 1)*fib(n + 1)
  invariant k == 2*fib(n)*fib(n + 1)
  {
    x,y,k := y, x + k + y, k + 2*y;
    n := n + 1;
  }
}
```

university of groningen

Notation

Before we continue, we need some extra notation.

This notation lacks in the book, but it makes manual proofs easier.

university of groningen

Notation: logical quantifiers

- equivalence ($\equiv$): $P \equiv Q$ is notation for $(P \Rightarrow Q) \wedge (Q \Rightarrow P)$
- $\forall x \in V : P(x) \quad \equiv \quad \forall x(x \in V \Rightarrow P(x))$
- $\exists x \in V : P(x) \quad \equiv \quad \exists x(x \in V \ \wedge \ P(x))$

university of groningen

Notation: Sets

- standard notation: $\{x \mid P(x)\}$
  For example $\{x \mid 0 \leq x^3 \leq 100\} = \{0, 1, 2, 3, 4\}$
- alternative notation: $\{E(x) \mid x : P(x)\}$, e.g.
  $\{2 \cdot x \mid x : 0 \leq x \leq 5\} = \{0, 2, 4, 6, 8, 10\}$
- notations are equivalent: $\{E(x) \mid x : P(x)\} \ = \ \{y \mid \exists x(y = E(x) \ \wedge \ P(x))\}$
- empty set: $\emptyset$
- element: $w \in V$ means "w is member of the set V"
- $\#V$: number of elements of the set V
- $\#\emptyset = 0$
- singleton rule: $\#\{a\} = 1$
- split rule: $\#V = \#(V \cap W) + \#(V \backslash W)$
- $\#\{x \in V \mid x = a\} = \text{ord}(a \in V)$, where ord(**true**) $= 1$ and ord(**false**) $= 0$

university of groningen

Notation: Segments

- Segment/interval: consecutive subset of $\mathbb{Z}$
- closed: $[m..n] \;=\; \{k \mid m \leq k \leq n\}$
- open: $(m..n) \;=\; \{k \mid m < k < n\}$
- half open: $[m..n) \;=\; \{k \mid m \leq k < n\}$
- half open: $(m..n] \;=\; \{k \mid m < k \leq n\}$
- Advice: use half open intervals!
  Why? Well, $\#[m..n) = \#(m..n] = n - m$

university of groningen

Notation: Maxima/Minima Sets/Segments

- Max V: maximum element of the set V
- Min V: minimum element of the set V
- For $V \neq \emptyset$ we have
  $a = \text{Max } V \quad \equiv \quad a \in V \;\wedge\; \forall x(x \in V \Rightarrow x \leq a)$
  $a = \text{Min } V \quad \equiv \quad a \in V \;\wedge\; \forall x(x \in V \Rightarrow a \leq x)$
- operator **max**: x **max** $y = \text{Max } \{x, y\}$
- operator **min**: x **min** $y = \text{Min } \{x, y\}$
- singleton rules: $\text{Max } \{a\} = a$ and $\text{Min } \{a\} = a$
- empty set rules: $\text{Max } \emptyset = -\infty$ and $\text{Min } \emptyset = \infty$
- split rule: $\text{Max } V = \text{Max } (V \cap W)$ **max** $\text{Max } (V \setminus W)$
- split rule: $\text{Min } V = \text{Min } (V \cap W)$ **min** $\text{Min } (V \setminus W)$

university of groningen

Notation: Families/Bags/Multi sets

- $(f(i) \mid i \in V)$ is the *family* consisting of the values $f(i)$ for all $i \in V$.
- Alternative notation: $(f(i) \mid i \in V : P(i))$
- Note that the (above) family differs from the set $\{f(i) \mid i \in V\}$
- $\{i \textbf{ div } 2 \mid i \in [0..5)\} = \{0, 1, 2\}$
- $(i \textbf{ div } 2 \mid i \in [0..5)) = (0, 0, 1, 1, 2)$
- In a family, the order of elements is relevant, so $(1, 0, 0) \neq (0, 1, 0)$, $(1, 0, 0) \neq (0, 0, 1)$, and $(0, 1, 0) \neq (0, 0, 1)$.

university of groningen

Notation: Sum operator on families

- $\Sigma(f(i) \mid i \in V)$ is the sum of the elements of the *family*.
- $\Sigma(f(i) \mid i \in \emptyset) = 0$
- $\Sigma(f(i) \mid i \in \{a\}) = f(a)$
- $\Sigma(f(i) \mid i \in V) = \Sigma(f(i) \mid i \in V \cap W) + \Sigma(f(i) \mid i \in V \setminus W)$

university of groningen

Notation: Product operator on families

- $\prod(f(i) \mid i \in V)$ is the product of the elements of the *family*.
- $\prod(f(i) \mid i \in \emptyset) = 1$
- $\prod(f(i) \mid i \in \{a\}) = f(a)$
- $\prod(f(i) \mid i \in V) = \prod(f(i) \mid i \in V \cap W) \cdot \prod(f(i) \mid i \in V \backslash W)$

university of groningen

Notation: Distributive laws

- mutiplication over addition: $\Sigma(x \cdot f(i) \mid i \in V) = x \cdot \Sigma(f(i) \mid i \in V)$
- addition over Max : $\text{Max}\ (x + f(i) \mid i \in V) = x + \text{Max}\ (f(i) \mid i \in V)$
- addition over Min : $\text{Min}\ (x + f(i) \mid i \in V) = x + \text{Min}\ (f(i) \mid i \in V)$
- multiplication over Max :
  $x \geq 0 \ \wedge \ V \neq \emptyset \Rightarrow \text{Max}\ (x \cdot f(i) \mid i \in V) = x \cdot \text{Max}\ (f(i) \mid i \in V)$
- multiplication over Min :
  $x \geq 0 \ \wedge \ V \neq \emptyset \Rightarrow \text{Min}\ (x \cdot f(i) \mid i \in V) = x \cdot \text{Min}\ (f(i) \mid i \in V)$

But,

- $x < 0 \ \wedge \ V \neq \emptyset \Rightarrow \text{Max}\ (x \cdot f(i) \mid i \in V) = x \cdot \text{Min}\ (f(i) \mid i \in V)$
- $x < 0 \ \wedge \ V \neq \emptyset \Rightarrow \text{Min}\ (x \cdot f(i) \mid i \in V) = x \cdot \text{Max}\ (f(i) \mid i \in V)$

university of groningen

Example: Summing an array

const $n : \mathbb{N},\ a :$ **array** $[0..n)$ **of** $\mathbb{Z}$;
var $x : \mathbb{Z}$;
$\{P :$ **true** $\}$
S
$\{Q : x = \Sigma(a[i] \mid i : i \in [0..n))\}$

To reduce the size of our formulas we introduce (for $0 \leq k \leq n$):

$$S(k) = \Sigma(a[i] \mid i : i \in [0..k))$$

Next, we rewrite the postcondition: $Q : x = S(n)$

university of groningen

Example: $S(k) = \Sigma(a[i] \mid i : i \in [0..k))$

It is clear that $S(0) = 0$, since the domain of the sum is empty.

For $0 \leq k < n$, we compute $S(k+1)$:

$$\begin{array}{ll} & S(k+1) \\ = & \{ \text{ definition } \} \\ & \Sigma(a[i] \mid i : i \in [0..k+1)) \\ = & \{ \text{ split domain: } i = k \ \vee \ i < k \ \} \\ & a[k] + \Sigma(a[i] \mid i : i \in [0..k)) \\ = & \{ \text{ definition } \} \\ & a[k] + S(k) \end{array}$$

So, we found the following recurrence:

$$\begin{array}{rcl} & & S(0) = 0 \\ 0 \leq k < n & \Rightarrow & S(k+1) = a[k] + S(k) \end{array}$$

university of groningen

Summing an array: step-by-step

1 Choose an invariant J, and guard B.
The postcondition is $Q: \ x = S(n)$.
We use the heuristic *replace the constant n by a variable k* and augment the invariant with the domain condition $0 \leq k \leq n$.

$$J: \ 0 \leq k \leq n \ \wedge \ x = S(k)$$
$$B: \ k \neq n$$

$J \ \wedge \ \neg B$
$\equiv$ { definition J and B}
$0 \leq k \leq n \ \wedge \ x = S(k) \ \wedge \ k = n$
$\Rightarrow$ { substitute $k = n$; logic }
$Q : x = S(n)$

university of groningen

Summing an array: step-by-step

2 Initialization: Find a command T_0 such that
$\{P: \ \textbf{true} \} \ T_0 \ \{J: \ 0 \leq k \leq n \ \wedge \ x = S(k)\}$

This is easy: since $S(0) = 0$ we choose $k := 0; \ x := 0;$

$\{P: \ \textbf{true} \}$
$\{0 \leq 0 \leq n \ \wedge \ 0 = S(0)\}$
$k, x := 0, 0;$
$\{J: \ 0 \leq k \leq n \ \wedge \ x = S(k)\}$

3 Variant function: choose a vf $\in \mathbb{Z}$ and prove $J \ \wedge \ B \Rightarrow \text{vf} \geq 0$.
Since initialliy $k = 0$ an the guard is $k \neq n$, we need to increase k.
So, we choose vf $= n - k \in \mathbb{Z}$. Clearly, $J \ \wedge \ B \Rightarrow J \Rightarrow k \leq n \equiv vf \geq 0$

4 Body of the loop: $\{J \ \wedge \ B \ \wedge \ \text{vf} = V\} \ S \ \{J \ \wedge \ \text{vf} < V\}$.

$\{0 \leq k \leq n \ \wedge \ x = S(k) \ \wedge \ k \neq n \ \wedge \ n - k = V\}$
$(*$ calculus; $k < n$; prepare $k := k + 1$; use recurrence $*)$
$\{0 \leq k + 1 \leq n \ \wedge \ x + a[k] = S(k + 1) \ \wedge \ n - (k + 1) < V\}$
$x := x + a[k];$
$\{0 \leq k + 1 \leq n \ \wedge \ x = S(k + 1) \ \wedge \ n - (k + 1) < V\}$
$k := k + 1;$
$\{0 \leq k \leq n \ \wedge \ x = S(k) \ \wedge \ n - k < V\}$

university of groningen

Summing an array: step-by-step

5 Conclusion: we derived the following program fragment:

$$
\begin{array}{l}
\textbf{const } n : \mathbb{N},\ a : \textbf{ array } [0..n) \textbf{ of } \mathbb{Z}; \\
\textbf{var } x : \mathbb{Z}; \\
\quad \{P : \ \textbf{true}\ \} \\
k := 0; \\
x := 0; \\
\quad \{J : 0 \leq k \leq n \ \wedge \ x = S(k)\} \\
\qquad (* \ \mathrm{vf} = n - k \ *) \\
\textbf{while } k \neq n \textbf{ do} \\
\quad x := x + a[k]; \\
\quad k := k + 1; \\
\textbf{end}; \\
\quad \{Q : \ x = \Sigma(a[i] \mid i : i \in [0..n))\}
\end{array}
$$

university of groningen

Summing an array (Dafny version)

```
ghost function sum(a: array<int>, upb: int := a.Length) : int
requires 0 <= upb <= a.Length
reads a
{
    if upb == 0 then 0
    else a[upb - 1] + sum(a, upb - 1)
}


method sumArray(a: array<int>) returns (x: int)
ensures x == sum(a)
{
    x := 0;
    var k: int := 0;
    while k < a.Length
        invariant 0 <= k <= a.Length
        invariant x == sum(a, k)
        decreases a.Length - k
    {
        x := x + a[k];
        k := k + 1;
    }
}
```

university of groningen