Program Construction and Reasoning

Shin-Cheng Mu

Institute of Information Science, Academia Sinica, Taiwan

2010 Formosan Summer School on
Logic, Language, and Computation
June 28 – July 9, 2010

So, what is this course about?

- I am going to teach you how to write programs.
- But you program much more than I do. What about programming could I possibly teach you?

Programming Language Theory?

It has always been, and still is, hard to talk to people about my research.

- "It's called 'programming language'."

Programming Language Theory?

It has always been, and still is, hard to talk to people about my research.

- "It's called 'programming language'."
- "Like, making computers understand natural languages?"

Programming Language Theory?

It has always been, and still is, hard to talk to people about my research.

- "It's called 'programming language'."
- "Like, making computers understand natural languages?"
- "Well, no... I mean the languages we use to communicate to computers. We design better programming language concepts to make programming easier."

Programming Language Theory?

It has always been, and still is, hard to talk to people about my research.

- "It's called 'programming language'."
- "Like, making computers understand natural languages?"
- "Well, no... I mean the languages we use to communicate to computers. We design better programming language concepts to make programming easier."
- "...surely it is the easiest to program in natural languages?"

Programming Language Theory?

It has always been, and still is, hard to talk to people about my research.

- "It's called 'programming language'."
- "Like, making computers understand natural languages?"
- "Well, no... I mean the languages we use to communicate to computers. We design better programming language concepts to make programming easier."
- "...surely it is the easiest to program in natural languages?"
- "Err, no. In fact we are trying to make programming more mathematical."

Programming Language Theory?

It has always been, and still is, hard to talk to people about my research.

- "It's called 'programming language'."
- "Like, making computers understand natural languages?"
- "Well, no... I mean the languages we use to communicate to computers. We design better programming language concepts to make programming easier."
- "...surely it is the easiest to program in natural languages?"
- "Err, no. In fact we are trying to make programming more mathematical."
- "...and you call that an improvement?"

Correctness?

Or I could try to explain that our concern is about "correctness."

- "And what does that mean?"

Correctness?

Or I could try to explain that our concern is about "correctness."

- "And what does that mean?"
- "That a program meets its specification."
- (totally confused) "A program meets . . . what?"

Correctness?

Or I could try to explain that our concern is about "correctness."

- "And what does that mean?"
- "That a program meets its specification."
- (totally confused) "A program meets . . . what?"
- "Ok, I mean to ensure that a computer does what it is supposed to do."

Correctness?

Or I could try to explain that our concern is about "correctness."

- "And what does that mean?"
- "That a program meets its specification."
- (totally confused) "A program meets . . . what?"
- "Ok, I mean to ensure that a computer does what it is supposed to do."
- "Doesn't a computer always do what it is instructed to do?"

Maximum Segment Sum

- Given a list of numbers, find the maximum sum of a *consecutive* segment.
  - $[-1, 3, 3, -4, -1, 4, 2, -1] \Rightarrow 7$
  - $[-1, 3, 1, -4, -1, 4, 2, -1] \Rightarrow 6$
  - $[-1, 3, 1, -4, -1, 1, 2, -1] \Rightarrow 4$

Maximum Segment Sum

- Given a list of numbers, find the maximum sum of a *consecutive* segment.
  - $[-1, 3, 3, -4, -1, 4, 2, -1] \Rightarrow 7$
  - $[-1, 3, 1, -4, -1, 4, 2, -1] \Rightarrow 6$
  - $[-1, 3, 1, -4, -1, 1, 2, -1] \Rightarrow 4$
- Not trivial. However, there is a linear time algorithm.

Maximum Segment Sum

- Given a list of numbers, find the maximum sum of a *consecutive* segment.
  - $[-1, 3, 3, -4, -1, 4, 2, -1] \Rightarrow 7$
  - $[-1, 3, 1, -4, -1, 4, 2, -1] \Rightarrow 6$
  - $[-1, 3, 1, -4, -1, 1, 2, -1] \Rightarrow 4$

	-1	3	1	-4	-1	1	2	-1		
▶	3	4	1	0	2	3	2	0	0	$(up + right) \uparrow 0$
	4	4	3	3	3	3	2	0	0	$up \uparrow right$

A Simple Program Whose Proof is Not

- The specification: $max\ \{\ sum\ (i,j) \mid 0 \leq i \leq j \leq N\ \}$, where $sum\ (i,j) = a[i] + a[i+1] + \ldots + a[i]$.
  - *What* we want the program to do.
- The program:

```
s = 0; m = 0;
for (i=0; i<=N; i++) {
    s = max(0, a[j]+s);
    m = max(m, s);
}
```

  - *How* to do it.
- They do not look like each other at all!
- Moral: programs that appear "simple" might not be that simple after all!

Programming, and Programming Languages

- Correctness: that the behaviour of a program is allowed by the specification.
- Semantics: defining "behaviours" of a program.
- Programming: to code up a correct program!

Programming, and Programming Languages

- Correctness: that the behaviour of a program is allowed by the specification.
- Semantics: defining "behaviours" of a program.
- Programming: to code up a correct program!
- Thus the job of a programming language is to help the programmer to program,
  - either by making it easy to check that whether a program is correct,
  - or by ensuring that programmers may only construct correct programs, that is, disallowing the very construction of incorrect programs!

Verification v.s. Derivation

- Verification: given a program, prove that it is correct with respect to some specification.
- Derivation: start from the specification, and attempt to construct *only* correct programs!
- Dijkstra: *"to prove the correctness of a given program, was in a sense putting the cart before the horse. A much more promising approach turned out to be letting correctness proof and program grow hand in hand: with the choice of the structure of the correctness proof one designs a program for which this proof is applicable."*
- What happened so far is that theoretical development of one side benefits the other.
- We focus on verification today, and talk about derivation tomorrow.

Can you Implement Binary Search?

Given a sorted array of N numbers and a key to search for, either locate the position where the key resides in the array, or report that the value does not present in the array, in $O(\log N)$ time.

- You would not expect it to be a hard programming task.
- Jon Bentley, however, noted:

> *"I've assigned this problem in courses at Bell Labs and IBM. Professional programmers had a couple of hours to convert the above description into a program in the language of their choice; . . . 90% of the programmers found bugs in their programs.*
>
> *. . . Knuth points out that while the first binary search was published in 1946, the first published binary search without bugs did not appear until 1962."*

Give It a Try?

- Bentley: "The only way you'll believe this is by putting down this column right now and writing the code yourself."
- Given: an array $a[0, N)$ of N elements,
- that is sorted: $(\forall i, j : 0 \leq i < j < N : a[i] \leq a[j])$.
- Find i such that $a[i] = K$, or report that K is not in the array.

The Guarded Command Language

In this course we will talk about program construction using Dijkstra's calculus. Most of the materials are from Kaldewaij.

- A program computing the greatest common divisor:

$$
\begin{array}{l}
|[\ \textbf{con}\ A, B : \mathit{int} \\
\ ;\textbf{var}\ x, y : \mathit{int}; \\
\\
\quad x, y := A, B; \\
\quad \textbf{do}\ y < x \rightarrow x := x - y \\
\qquad [\!]\ x < y \rightarrow y := y - x \\
\quad \textbf{od} \\
\\
]|.
\end{array}
$$

- **do** denotes loops with guarded bodies.

The Guarded Command Language

In this course we will talk about program construction using Dijkstra's calculus. Most of the materials are from Kaldewaij.

- A program computing the greatest common divisor:

$$
\begin{array}{l}
|[\ \textbf{con}\ A, B : \mathit{int}\ \{0 < A \wedge 0 < B\} \\
\ ;\textbf{var}\ x, y : \mathit{int}; \\
\\
\quad x, y := A, B; \\
\quad \textbf{do}\ y < x \rightarrow x := x - y \\
\qquad [\!]\ x < y \rightarrow y := y - x \\
\quad \textbf{od} \\
\quad \{x = y = \mathit{gcd}(A, B)\} \\
]|.
\end{array}
$$

- **do** denotes loops with guarded bodies.
- Assertions delimited in curly brackets.

The Hoare Triple

- The *state space* of a program is the states of all its variables.
  - E.g. state space for the GCD program is $(int \times int)$.
- The *Hoare triple* $\{P\}\, S\, \{Q\}$, operationally, denotes that the statement S, when executed in a state satisfying P, *terminates* in a state satisfying Q.
- Perhaps the simplest statement: $\{P\}\, skip\, \{Q\}$ iff. $P \Rightarrow Q$.
  - $\{X > 0 \ \wedge \ Y > 0\}\, skip\, \{X \geq 0\}$.
  - Note that the annotations need not be "exact."

The Hoare Triple

- $\{P\}\, S\, \{true\}$ expresses that S terminates.
- $\{P\}\, S\, \{Q\}$ and $P_0 \Rightarrow P$ implies $\{P_0\}\, S\, \{Q\}$.
- $\{P\}\, S\, \{Q\}$ and $Q \Rightarrow Q_0$ implies $\{P\}\, S\, \{Q_0\}$.
- $\{P\}\, S\, \{Q\}$ and $\{P\}\, S\, \{R\}$ equivales $\{P\}\, S\, \{Q \wedge R\}$.
- $\{P\}\, S\, \{Q\}$ and $\{R\}\, S\, \{Q\}$ equivales $\{P \vee R\}\, S\, \{Q\}$.

Substitution

- $P[E/x]$: substituting *free* occurrences of x in P for E.
- We do so in mathematics all the time. A formal definition of substitution, however, is rather tedious.
- For this lecture we will only appeal to "common sense":
  - E.g. $(x \leq 3)[x-1/x] \Leftrightarrow x-1 \leq 3 \Leftrightarrow x \leq 4$.
  - $$\begin{aligned} & ((\exists y : y \in \mathbb{N} : x < y) \wedge y < x)[y+1/y] \\ \Leftrightarrow\ & (\exists y : y \in \mathbb{N} : x < y) \wedge y + 1 < x. \end{aligned}$$
  - $$\begin{aligned} & (\exists y : y \in \mathbb{N} : x < y)[y/x] \\ \Leftrightarrow\ & (\exists z : z \in \mathbb{N} : y < z). \end{aligned}$$
- The notation $[E/x]$ hints at "divide by x and multiply by E." In the refinement calculus, substitution is closely related to assignments, thus some also write $[x := E]$.

Substitution and Assignments

- Which is correct:
  1. $\{P\}\, x := E\, \{P[E/x]\}$, or
  2. $\{P[E/x]\}\, x := E\, \{P\}$?

Substitution and Assignments

- Which is correct:
  1. $\{P\}\, x := E\, \{P[E/x]\}$, or
  2. $\{P[E/x]\}\, x := E\, \{P\}$?
- Answer: 2! For example:

$$\begin{aligned} & \{(x \leq 3)[x+1/x]\}\, x := x+1\, \{x \leq 3\} \\ \Leftrightarrow\ & \{x+1 \leq 3\}\, x := x+1\, \{x \leq 3\} \\ \Leftrightarrow\ & \{x \leq 2\}\, x := x+1\, \{x \leq 3\}. \end{aligned}$$

Catenation

- $\{P\}\, S; T\, \{Q\}$ equivals that there exists R such that $\{P\}\, S\, \{R\}$ and $\{R\}\, T\, \{Q\}$.
- Verify:

$$
\begin{array}{l}
|[\ \textbf{var}\ x, y : \mathit{int};\\
\quad \{x = A\ \wedge\ y = B\}\\
\quad x := x - y;\\
\\
\quad y := x + y;\\
\\
\quad x := y - x;\\
\quad \{x = B\ \wedge\ y = A\}\\
]|.
\end{array}
$$

Catenation

- $\{P\}\, S; T\, \{Q\}$ equivals that there exists R such that $\{P\}\, S\, \{R\}$ and $\{R\}\, T\, \{Q\}$.
- Verify:

$$
\begin{array}{l}
|[\ \textbf{var}\ x, y : \mathit{int};\\
\quad \{x = A\ \wedge\ y = B\}\\
\quad x := x - y;\\
\\
\quad y := x + y;\\
\quad \{y - x = B\ \wedge\ y = A\}\\
\quad x := y - x;\\
\quad \{x = B\ \wedge\ y = A\}\\
]|.
\end{array}
$$

Catenation

- $\{P\}\, S; T\, \{Q\}$ equivals that there exists R such that $\{P\}\, S\, \{R\}$ and $\{R\}\, T\, \{Q\}$.
- Verify:

$$
\begin{array}{l}
|[\ \textbf{var}\ x, y : \mathit{int};\\
\quad \{x = A\ \wedge\ y = B\}\\
\quad x := x - y;\\
\qquad\qquad\qquad\qquad \{x + y - x = B\ \wedge\ x + y = A\}\\
\quad y := x + y;\\
\quad \{y - x = B\ \wedge\ y = A\}\\
\quad x := y - x;\\
\quad \{x = B\ \wedge\ y = A\}\\
]|.
\end{array}
$$

Catenation

- $\{P\}\, S; T\, \{Q\}$ equivals that there exists R such that $\{P\}\, S\, \{R\}$ and $\{R\}\, T\, \{Q\}$.
- Verify:

$$
\begin{array}{l}
|[\ \textbf{var}\ x, y : \mathit{int};\\
\quad \{x = A\ \wedge\ y = B\}\\
\quad x := x - y;\\
\quad \{y = B\ \wedge\ x + y = A\} \Rightarrow \{x + y - x = B\ \wedge\ x + y = A\}\\
\quad y := x + y;\\
\quad \{y - x = B\ \wedge\ y = A\}\\
\quad x := y - x;\\
\quad \{x = B\ \wedge\ y = A\}\\
]|.
\end{array}
$$

Catenation

- $\{P\}\, S; T\, \{Q\}$ equivals that there exists R such that $\{P\}\, S\, \{R\}$ and $\{R\}\, T\, \{Q\}$.
- Verify:

$$
\begin{array}{l}
|[\ \textbf{var}\ x, y : \mathit{int};\\
\quad \{x = A\ \wedge\ y = B\} \Rightarrow \{y = B\ \wedge\ x - y + y = A\}\\
\quad x := x - y;\\
\quad \{y = B\ \wedge\ x + y = A\}\\
\quad y := x + y;\\
\quad \{y - x = B\ \wedge\ y = A\}\\
\quad x := y - x;\\
\quad \{x = B\ \wedge\ y = A\}\\
]|.
\end{array}
$$

Catenation

- $\{P\}\, S; T\, \{Q\}$ equivals that there exists R such that $\{P\}\, S\, \{R\}$ and $\{R\}\, T\, \{Q\}$.
- Verify:

$$
\begin{array}{l}
|[\ \textbf{var}\ x, y : \mathit{int};\\
\quad \{x = A\ \wedge\ y = B\}\\
\quad x := x - y;\\
\quad \{y = B\ \wedge\ x + y = A\}\\
\quad y := x + y;\\
\quad \{y - x = B\ \wedge\ y = A\}\\
\quad x := y - x;\\
\quad \{x = B\ \wedge\ y = A\}\\
]|.
\end{array}
$$

If-Conditionals

- Selection takes the form **if** $B_0 \to S_0 \,[\!]\, \ldots \,[\!]\, B_n \to S_n$ **fi**.
- Each B_i is called a *guard*; $B_i \to S_i$ is a *guarded command*.
- If none of the guards $B_0 \ldots B_n$ evaluate to true, the program aborts. Otherwise, one of the command with a true guard is chosen *non-deterministically* and executed.

If-Conditionals

- Selection takes the form **if** $B_0 \to S_0 \,[\!]\, \ldots \,[\!]\, B_n \to S_n$ **fi**.
- Each B_i is called a *guard*; $B_i \to S_i$ is a *guarded command*.
- If none of the guards $B_0 \ldots B_n$ evaluate to true, the program aborts. Otherwise, one of the command with a true guard is chosen *non-deterministically* and executed.
- To annotate an **if** statement:

$$
\begin{array}{l}
\{P\} \\
\textbf{if } B_0 \to \{P \wedge B_0\}\, S_0\, \{Q\} \\
\quad [\!]\ B_1 \to \{P \wedge B_1\}\, S_1\, \{Q\} \\
\textbf{fi} \\
\{Q, Pf\},
\end{array}
$$

where $Pf : P \Rightarrow B_0 \vee B_1$.

Binary Maximum

- Goal: to assign $x \uparrow y$ to z. By definition,
  $z = x \uparrow y \quad \leftrightarrow \quad (z = x \ \vee \ z = y) \ \wedge \ x \leq z \ \wedge \ y \leq z.$

Binary Maximum

- Goal: to assign $x \uparrow y$ to z. By definition,
  $z = x \uparrow y \quad \leftrightarrow \quad (z = x \ \vee \ z = y) \ \wedge \ x \leq z \ \wedge \ y \leq z.$
- Try $z := x$. We reason:

$$\begin{aligned} & ((z = x \ \vee \ z = y) \ \wedge \ x \leq z \ \wedge \ y \leq z)[x/z] \\ \Leftrightarrow \ & (x = x \ \vee \ x = y) \ \wedge \ x \leq x \ \wedge \ y \leq x \\ \Leftrightarrow \ & y \leq x, \end{aligned}$$

  which hinted at using a guarded command: $y \leq x \rightarrow z := x$.

Binary Maximum

- Goal: to assign $x \uparrow y$ to z. By definition,
  $z = x \uparrow y \quad \leftrightarrow \quad (z = x \ \vee \ z = y) \ \wedge \ x \leq z \ \wedge \ y \leq z.$
- Try $z := x$. We reason:

$$\begin{aligned} & ((z = x \ \vee \ z = y) \ \wedge \ x \leq z \ \wedge \ y \leq z)[x/z] \\ \Leftrightarrow \ & (x = x \ \vee \ x = y) \ \wedge \ x \leq x \ \wedge \ y \leq x \\ \Leftrightarrow \ & y \leq x, \end{aligned}$$

  which hinted at using a guarded command: $y \leq x \rightarrow z := x$.
- Indeed:

$$\begin{array}{l} \{true\} \\ \textbf{if } y \leq x \rightarrow \{y \leq x\}\, z := x\, \{z = x \uparrow y\} \\ \ [\!] \ x \leq y \rightarrow \{x \leq y\}\, z := y\, \{z = x \uparrow y\} \\ \textbf{fi} \\ \{z = x \uparrow y\}. \end{array}$$

On Understanding Programs

- There are two ways to understand the program below:

$$\begin{array}{l} \textbf{if } B_{00} \to S_{00} \,[\!]\, B_{01} \to S_{01} \textbf{ fi}; \\ \textbf{if } B_{10} \to S_{10} \,[\!]\, B_{11} \to S_{11} \textbf{ fi}; \\ \qquad \vdots \\ \textbf{if } B_{n0} \to S_{n0} \,[\!]\, B_{n1} \to S_{n1} \textbf{ fi}. \end{array}$$

- One takes effort exponential to n; the other is linear.
- Dijkstra: "... if we ever want to be able to compose really large programs reliably, we need a programming discipline such that the intellectual effort needed to understand a program does not grow more rapidly than in proportion to the program length."

Loops

- Repetition takes the form **do** $B_0 \to S_0 \; [\!] \; \ldots \; [\!] \; B_n \to S_n$ **od**.
- If none of the guards $B_0 \ldots B_n$ evaluate to true, the loop terminates. Otherwise one of the commands is chosen non-deterministically, before the next iteration.

Loops

- Repetition takes the form **do** $B_0 \to S_0 \; [\!] \; \ldots \; [\!] \; B_n \to S_n$ **od**.
- If none of the guards $B_0 \ldots B_n$ evaluate to true, the loop terminates. Otherwise one of the commands is chosen non-deterministically, before the next iteration.
- To annotate a loop (for partial correctness):

$$
\begin{array}{l}
\{P\} \\
\textbf{do } B_0 \to \{P \wedge B_0\}\, S_0\, \{P\} \\
\quad [\!] \; B_1 \to \{P \wedge B_1\}\, S_1\, \{P\} \\
\textbf{od} \\
\{Q, Pf\},
\end{array}
$$

where $Pf : P \wedge \neg B_0 \wedge \neg B_1 \Rightarrow Q$.

Loops

- Repetition takes the form **do** $B_0 \to S_0 \; [\!] \; \ldots \; [\!] \; B_n \to S_n$ **od**.
- If none of the guards $B_0 \ldots B_n$ evaluate to true, the loop terminates. Otherwise one of the commands is chosen non-deterministically, before the next iteration.
- To annotate a loop (for partial correctness):

$$
\begin{array}{l}
\{P\} \\
\textbf{do } B_0 \to \{P \wedge B_0\}\, S_0\, \{P\} \\
\quad [\!] \; B_1 \to \{P \wedge B_1\}\, S_1\, \{P\} \\
\textbf{od} \\
\{Q, Pf\},
\end{array}
$$

  where $Pf : P \wedge \neg B_0 \wedge \neg B_1 \Rightarrow Q$.
- P is called the *loop invariant*. Every loop should be constructed with an invariant in mind!

Linear-Time Exponentiation

$$
\begin{array}{l}
|[\ \textbf{con}\ N\ \{0 \leq N\};\ \textbf{var}\ x, n : \mathit{int};\\
\quad x, n := 1, 0\\
\quad ;\textbf{do}\ n \neq N \rightarrow\\
\qquad x, n := x + x, n + 1\\
\quad \textbf{od}\\
\quad \{x = 2^N \qquad\}\\
]|
\end{array}
$$

Linear-Time Exponentiation

$$
\begin{array}{l}
|[\ \textbf{con}\ N\ \{0 \leq N\};\ \textbf{var}\ x, n : \mathit{int};\\
\\
\quad x, n := 1, 0\\
\quad \{x = 2^n \wedge n \leq N\}\\
\ ;\textbf{do}\ n \neq N \rightarrow\\
\\
\qquad x, n := x + x, n + 1\\
\\
\quad \textbf{od}\\
\quad \{x = 2^N \qquad\}\\
]|
\end{array}
$$

Linear-Time Exponentiation

$|[$ **con** N $\{0 \leq N\}$; **var** $x, n : int$;

$x, n := 1, 0$
$\{x = 2^n \wedge n \leq N\}$
; **do** $n \neq N \rightarrow$

$x, n := x + x, n + 1$

od
$\{x = 2^N, Pf2\}$
$]|$

Pf2:

$$x = 2^n \wedge n \leq N \wedge \neg(n \neq N)$$
$$\Rightarrow x = 2^N$$

Linear-Time Exponentiation

$|[$ **con** N $\{0 \leq N\}$; **var** $x, n : int$;

$x, n := 1, 0$
$\{x = 2^n \wedge n \leq N\}$
; **do** $n \neq N \rightarrow$

$x, n := x + x, n + 1$
$\{x = 2^n \wedge n \leq N, Pf1\}$
od
$\{x = 2^N, Pf2\}$
$]|$

Pf2:

$$x = 2^n \wedge n \leq N \wedge \neg(n \neq N) \Rightarrow x = 2^N$$

Linear-Time Exponentiation

$$
\begin{array}{l}
|[\ \textbf{con}\ N\ \{0 \le N\};\ \textbf{var}\ x, n : \mathit{int};\\
\quad x, n := 1, 0\\
\quad \{x = 2^n \wedge n \le N\}\\
\ ;\textbf{do}\ n \neq N \rightarrow\\
\qquad \{x = 2^n \wedge n \le N \wedge n \neq N\}\\
\qquad x, n := x + x, n + 1\\
\qquad \{x = 2^n \wedge n \le N, \mathit{Pf1}\}\\
\quad \textbf{od}\\
\quad \{x = 2^N, \mathit{Pf2}\}\\
]|
\end{array}
$$

Pf2:

$$x = 2^n \wedge n \le N \wedge \neg(n \neq N)$$
$$\Rightarrow x = 2^N$$

Linear-Time Exponentiation

$$
\begin{array}{l}
|[\ \textbf{con}\ N\ \{0 \le N\};\ \textbf{var}\ x, n : \mathit{int};\\
\quad x, n := 1, 0\\
\quad \{x = 2^n \wedge n \le N\}\\
\ ;\textbf{do}\ n \neq N \rightarrow\\
\qquad \{x = 2^n \wedge n \le N \wedge n \neq N\}\\
\qquad x, n := x + x, n + 1\\
\qquad \{x = 2^n \wedge n \le N, \mathit{Pf1}\}\\
\quad \textbf{od}\\
\quad \{x = 2^N, \mathit{Pf2}\}\\
]|
\end{array}
$$

Pf1:

$$
\begin{array}{ll}
 & (x = 2^n \wedge n \le N)[x + x, n + 1/x, n]\\
\Leftrightarrow & x + x = 2^{n+1} \wedge n + 1 \le N\\
\Leftrightarrow & x = 2^n \wedge n < N
\end{array}
$$

Pf2:

$$
\begin{array}{l}
x = 2^n \wedge n \le N \wedge \neg(n \neq N)\\
\qquad \Rightarrow x = 2^N
\end{array}
$$

Greatest Common Divisor

- Known: $gcd(x, x) = x$; $gcd(x, y) = gcd(y, x - y)$ if $x > y$.

Greatest Common Divisor

- Known: $gcd(x, x) = x$; $gcd(x, y) = gcd(y, x - y)$ if $x > y$.
-

$$
\begin{array}{l}
[\![\ \textbf{con}\ A, B : int\ \{0 < A \wedge 0 < B\}; \\
\quad \textbf{var}\ x, y : int; \\
\\
\quad x, y := A, B \\
\quad \{0 < x \wedge 0 < y \wedge gcd(x, y) = gcd(A, B)\} \\
\quad ; \textbf{do}\ y < x \rightarrow x := x - y \\
\qquad [\!]\ x < y \rightarrow y := y - x \\
\quad \textbf{od} \\
\quad \{x = gcd(A, B) \wedge y = gcd(A, B)\} \\
]\!]
\end{array}
$$

Greatest Common Divisor

- Known: $gcd(x,x) = x$; $gcd(x,y) = gcd(y, x-y)$ if $x > y$.
- $$
\begin{array}{l}
[\![\ \textbf{con}\ A, B : int\ \{0 < A \wedge 0 < B\}; \\
\quad \textbf{var}\ x, y : int; \\
\\
\quad x, y := A, B \\
\quad \{0 < x \wedge 0 < y \wedge gcd(x,y) = gcd(A,B)\} \\
\quad ;\textbf{do}\ y < x \rightarrow x := x - y \\
\qquad [\!]\ x < y \rightarrow y := y - x \\
\quad \textbf{od} \\
\quad \{x = gcd(A,B) \wedge y = gcd(A,B)\} \\
]\!]
\end{array}
$$
- $$
\begin{array}{ll}
 & (0 < x \wedge 0 < y \wedge gcd(x,y) = gcd(A,B))[x - y/x] \\
\leftrightarrow & 0 < x - y \wedge 0 < y \wedge gcd(x-y, y) = gcd(A,B) \\
\Leftarrow & 0 < x \wedge 0 < y \wedge gcd(x,y) = gcd(A,B) \wedge y < x
\end{array}
$$

A Weird Equilibrium

- Consider the following program:

$$
\begin{array}{l}
|[\ \textbf{var}\ x, y, z : int \\
\quad \{true \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\} \\
\quad ;\textbf{do}\ x < y \rightarrow x := x + 1 \\
\qquad [\!]\ y < z \rightarrow y := y + 1 \\
\qquad [\!]\ z < x \rightarrow z := z + 1 \\
\quad \textbf{od} \\
\quad \{x = y = z\} \\
]|.
\end{array}
$$

- If it terminates at all, we do have $x = y = z$. But why does it terminate?

A Weird Equilibrium

- Consider the following program:

$$
\begin{array}{l}
|[\ \textbf{var}\ x, y, z : \mathit{int} \\
\quad \{\mathit{true}, \mathit{bnd} : 3 \times (x \uparrow y \uparrow z) - (x + y + z)\} \\
\quad ;\textbf{do}\ x < y \to x := x + 1 \\
\qquad [\!] \ y < z \to y := y + 1 \\
\qquad [\!] \ z < x \to z := z + 1 \\
\quad \textbf{od} \\
\quad \{x = y = z\} \\
]|.
\end{array}
$$

- If it terminates at all, we do have $x = y = z$. But why does it terminate?
  1. $\mathit{bnd} \geq 0$, and $\mathit{bnd} = 0$ implies none of the guards are true.
  2. $\{x < y \wedge \mathit{bnd} = t\}\, x := x + 1\, \{\mathit{bnd} < t\}$.

Repetition

To annotate a loop for *total correctness*:

$$
\begin{array}{l}
\{P, bnd : t\} \\
\textbf{do } B_0 \to \{P \wedge B_0\}\, S_0\, \{P\} \\
\quad [\!] \; B_1 \to \{P \wedge B_1\}\, S_1\, \{P\} \\
\textbf{od} \\
\{Q\},
\end{array}
$$

we have got a list of things to prove:

Repetition

To annotate a loop for *total correctness*:

$$
\begin{array}{l}
\{P, bnd : t\} \\
\textbf{do } B_0 \to \{P \wedge B_0\}\, S_0\, \{P\} \\
\quad [\!] \; B_1 \to \{P \wedge B_1\}\, S_1\, \{P\} \\
\textbf{od} \\
\{Q\},
\end{array}
$$

we have got a list of things to prove:

1. $B \wedge \neg B_0 \wedge \neg B_1 \Rightarrow Q$,

Repetition

To annotate a loop for *total correctness*:

$$
\begin{array}{l}
\{P, bnd : t\} \\
\textbf{do } B_0 \to \{P \wedge B_0\}\, S_0\, \{P\} \\
\quad [\!] \; B_1 \to \{P \wedge B_1\}\, S_1\, \{P\} \\
\textbf{od} \\
\{Q\},
\end{array}
$$

we have got a list of things to prove:

1. $B \wedge \neg B_0 \wedge \neg B_1 \Rightarrow Q$,
2. for all i, $\{P \wedge B_i\}\, S_i\, \{P\}$,

Repetition

To annotate a loop for *total correctness*:

$$
\begin{array}{l}
\{P, bnd : t\} \\
\textbf{do } B_0 \to \{P \wedge B_0\}\, S_0\, \{P\} \\
\quad [\!] \; B_1 \to \{P \wedge B_1\}\, S_1\, \{P\} \\
\textbf{od} \\
\{Q\},
\end{array}
$$

we have got a list of things to prove:

1. $B \wedge \neg B_0 \wedge \neg B_1 \Rightarrow Q$,
2. for all i, $\{P \wedge B_i\}\, S_i\, \{P\}$,
3. $P \wedge (B_1 \vee B_2) \Rightarrow t \geq 0$,

Repetition

To annotate a loop for *total correctness*:

$$
\begin{array}{l}
\{P, bnd : t\} \\
\textbf{do } B_0 \to \{P \wedge B_0\}\, S_0\, \{P\} \\
\quad [\!] \; B_1 \to \{P \wedge B_1\}\, S_1\, \{P\} \\
\textbf{od} \\
\{Q\},
\end{array}
$$

we have got a list of things to prove:

1. $B \wedge \neg B_0 \wedge \neg B_1 \Rightarrow Q$,
2. for all i, $\{P \wedge B_i\}\, S_i\, \{P\}$,
3. $P \wedge (B_1 \vee B_2) \Rightarrow t \geq 0$,
4. for all i, $\{P \wedge B_i \wedge t = C\}\, S_i\, \{t < C\}$.

E.g. Linear-Time Exponentiation

- What is the bound function?

$$
\begin{array}{l}
[\![\ \textbf{con}\ N\ \{0 \le N\};\ \textbf{var}\ x, n : \mathit{int};\\
\\
\quad x, n := 1, 0\\
\quad \{x = 2^n \wedge n \le N \qquad\qquad\qquad \}\\
\quad ;\textbf{do}\ n \neq N \rightarrow\\
\qquad x, n := x + x, n + 1\\
\quad \textbf{od}\\
\quad \{x = 2^N\}\\
]\!]
\end{array}
$$

E.g. Linear-Time Exponentiation

- What is the bound function?

$$
\begin{array}{l}
[\![\ \textbf{con}\ N\ \{0 \le N\};\ \textbf{var}\ x, n : \mathit{int};\\
\\
\quad x, n := 1, 0\\
\quad \{x = 2^n \wedge n \le N, \mathit{bnd} : N - n\}\\
\quad ;\textbf{do}\ n \neq N \rightarrow\\
\qquad x, n := x + x, n + 1\\
\quad \textbf{od}\\
\quad \{x = 2^N\}\\
]\!]
\end{array}
$$

- $x = 2^n \wedge n \wedge n \neq N \Rightarrow N - n \geq 0$,
- $\{\ldots \wedge N - n = t\}\, x, n := x + x, n - 1\, \{N - n < t\}$.

E.g. Greatest Common Divisor

- What is the bound function?

$$
\begin{array}{l}
|[\ \textbf{con}\ A, B : \mathit{int}\ \{0 < A \wedge 0 < B\}; \\
\quad \textbf{var}\ x, y : \mathit{int}; \\
\\
\quad x, y := A, B \\
\quad \{0 < x \wedge 0 < y \wedge \mathit{gcd}(x, y) = \mathit{gcd}(A, B) \qquad \} \\
\quad ; \textbf{do}\ y < x \rightarrow x := x - y \\
\qquad [\!] \ x < y \rightarrow y := y - x \\
\quad \textbf{od} \\
\quad \{x = \mathit{gcd}(A, B) \wedge y = \mathit{gcd}(A, B)\} \\
]|
\end{array}
$$

E.g. Greatest Common Divisor

- What is the bound function?

$$
\begin{array}{l}
|[\ \textbf{con}\ A, B : \mathit{int}\ \{0 < A \wedge 0 < B\}; \\
\quad \textbf{var}\ x, y : \mathit{int}; \\
\\
\quad x, y := A, B \\
\quad \{0 < x \wedge 0 < y \wedge \mathit{gcd}(x, y) = \mathit{gcd}(A, B), \mathit{bnd} : |x - y|\} \\
\quad ; \textbf{do}\ y < x \rightarrow x := x - y \\
\qquad [\!] \ x < y \rightarrow y := y - x \\
\quad \textbf{od} \\
\quad \{x = \mathit{gcd}(A, B) \wedge y = \mathit{gcd}(A, B)\} \\
]|
\end{array}
$$

- $\ldots \Rightarrow |x - y| \geq 0$,
- $\{\ldots 0 < y \wedge y < x \wedge |x - y| = t\}\, x := x - y\, \{|x - y| < t\}$.

The van Gasteren-Feijen Approach

- Van Gasteren and Feijen pointed a surprising fact: binary search does not apply only to sorted lists!
- In fact, they believe that comparing binary search to searching for a word in a dictionary is a major educational blunder.
- Their binary search: let Φ be a predicate on $(int \times int)$, with some additional constraints to be given later:

$$
\begin{array}{l}
|[\ \textbf{con}\ M, N : int\ \{M < N\ \wedge\ \Phi(M, N) \ldots\}; \\
\quad \textbf{var}\ l, r : int; \\
\quad bsearch \\
\quad \{M \leq l < N\ \wedge\ \Phi(l, l+1)\} \\
]|
\end{array}
$$

The Program

$$
\begin{array}{l}
\{M < N \wedge \Phi(M, N)\} \\
l, r := M, N \\
\{M \leq l < r \leq N \ \wedge \ \Phi(l, r), bnd : r - l\} \\
;\textbf{do } l + 1 \neq r \rightarrow \\
\quad \{\ldots \wedge \ l + 2 \leq r\} \\
\quad m := (l + r)/2 \\
\quad \{\ldots \wedge \ l < m < r\} \\
\quad ;\textbf{if } \Phi(m, r) \rightarrow l := m \\
\quad \ [\!] \ \Phi(l, m) \rightarrow r := m \\
\quad \textbf{fi} \\
\textbf{od} \\
\{M \leq l < N \ \wedge \ \Phi(l, l + 1)\}
\end{array}
$$

Proof of Correctness

Let's start with verifying the easier bits.

- When the loop exits:

$$\begin{aligned} & M \le l < r \le N \ \wedge \ \Phi(l, r) \ \wedge \ \neg(l + 1 \neq r) \\ \Rightarrow \ & M \le l < l + 1 \le N \ \wedge \ \Phi(l, l + 1) \\ \Leftrightarrow \ & M \le l < N \ \wedge \ \Phi(l, l + 1). \end{aligned}$$

- Termination: exercise.
- To verify $\{\ldots l + 2 \le r\}\, m := (l + r)/2\, \{\ldots l < m < r\}$:

$$\begin{aligned} & (l < m < r)[((l + r)/2)/m] \\ \Leftrightarrow \ & l < (l + r)/2 \ \Leftarrow \ l + 2 \le r. \end{aligned}$$

Proof of Correctness

- To verify that the loop body maintains the invariant, check the first branch in **if**:

$$
\begin{aligned}
& (M \leq l < r \leq N \ \wedge \ \Phi(l, r))[m/l] \\
\Leftrightarrow \ & M \leq m < r \leq N \ \wedge \ \Phi(m, r) \\
\Leftarrow \ & M \leq l < r \leq N \ \wedge \ \Phi(l, r) \ \wedge \ l < m < r \ \wedge \ \Phi(m, r).
\end{aligned}
$$

- Similarly with the other branch.
- However, we still need to be sure that at least one of the guards in **if** holds! Thus we need this property from Φ:

$$\Phi(l, r) \ \wedge \ l < m < r \ \Rightarrow \ \Phi(l, m) \ \vee \ \Phi(m, r). \qquad (1)$$

Instantiations

Some Φ that satisfies (1):

- $\Phi(i, j) = a[i] \neq a[j]$ for some array a. Van Gasteren and Feijen suggested using this as the example when introducing binary search.
- $\Phi(i, j) = a[i] < a[j]$,
- $\Phi(i, j) = a[i] \times a[j] \leq 0$,
- $\Phi(i, j) = a[i] \vee a[j]$, etc.

Searching for a Key

- To search for a key K in an ascending-sorted array a, it seems that we could just pick:

$$\Phi(i,j) = a[i] \leq K < a[j],$$

  and check whether $a[i] = K$ after the loop.
- However, we are not sure we can establish the precondition $a[l] \leq K < a[r]$!
- For a possibly empty array $a[0..N)$, imagine two elements $a[-1]$ and $a[N]$ such that $a[-1] \leq x$ and $x < a[N]$ for all x.
- Equivalently, pick:

$$\Phi(i,j) = (i = -1 \ \vee \ a[i] \leq K) \ \ \wedge \ \ (K < a[j] \ \vee \ j = N).$$

The Program

Recall $\Phi(i,j) = (i = -1 \ \vee \ a[i] \leq K) \ \ \wedge \ \ (K < a[j] \ \vee \ j = N)$.

$\{0 \geq N \wedge \Phi(-1, N)\}$
$l, r := -1, N$
$\{-1 \leq l < r \leq N \ \wedge \ \Phi(l, r), bnd : r - l\}$
; **do** $l + 1 \neq r \rightarrow$
  $\{\ldots \wedge \ l + 2 \leq r\}$
  $m := (l + r)/2$
  ; **if** $a[m] \leq K \rightarrow l := m$
  [] $K < a[m] \rightarrow r := m$
  fi
od
$\{-1 \leq l < N \ \wedge \ \Phi(l, l + 1)\}$
; **if** $l > -1 \rightarrow found := a[l] = k$
 [] $l = -1 \rightarrow found := false$
fi

Discussions

- "Adding" elements to a?
  - The invariant implies that $-1 < m < N$, thus $a[-1]$ and $a[N]$ are never accessed.
  - No actual alteration necessary.
  - It also enables us to handle possibly empty arrays
- Is the program different from the usual binary search you've seen?

A More Common Program

Bentley's program can be rephrased below:

$$
\begin{array}{l}
l, r := 0, N-1; \quad found := false; \\
\textbf{do } l \leq r \rightarrow \\
\quad m := (l + r)/2; \\
\quad \textbf{if } a[m] < K \rightarrow l := m + 1 \\
\quad \quad [\!] \; a[m] = K \rightarrow found := true; break \\
\quad \quad [\!] \; K < a[m] \rightarrow r := m - 1 \\
\quad \textbf{fi} \\
\textbf{od}.
\end{array}
$$

I'd like to derive it, but

- it is harder to formally deal with *break*
  - but Bentley also employed a semi-formal reasoning using a loop invariant to argue for the correctness of the program;
- to relate the test $a[m] < K$ to $l := m + 1$ we have to bring in the fact that a is sorted earlier.

Comparison

- The two programs do not solve exactly the same problem (e.g. when there are multiple Ks in a).
- Is the second program quicker because it assigns l and r to $m+1$ and $m-1$ rather than m?
  - $l := m+1$ because $a[m]$ is covered in another case;
  - $r := m-1$ because a range is represented differently.
- Is it quicker to perform an extra test to *return* early?
  - When K is not in a, the test is wasted.
  - Rolfe claimed that single comparison is quicker in average.
  - Knuth: single comparison needs $17.5 \lg N + 17$ instructions, double comparison needs $18 \lg N - 16$ instructions.

Exercise: Unimodel Search

- Let array $a[0, N)$, with $0 < N$, be the concatenation of a strictly increasing and a strictly decreasing array. Formally:

$$
\begin{aligned}
&(\exists k : 0 \leq k < N : \\
&\quad (\forall i : 0 < i \leq k : a[i-1] < a[i]) \wedge \\
&\quad (\forall j : k \leq j < N : a[j-1] > a[j])).
\end{aligned}
$$

  Use binary search to find the maximum element.
- What invariant to use?

Correct by Construction

Dijkstra: "The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness. But one should not first make the program and then prove its correctness, because then the requirement of providing the proof would only increase the poor programmer's burden. On the contrary: the programmer should . . . "
". . . [let] correctness proof and program grow hand in hand: with the choice of the structure of the correctness proof one designs a program for which this proof is applicable."

Program Derivation

- Wikipedia: *program derivation* is the derivation of a program from its specification, by mathematical means.
- To write a formal specification (which could be non-executable), and then apply mathematically correct rules in order to obtain an executable program.
- The program thus obtained is correct by construction.

But What is a Proof, Anyway?

Xavier Leroy, "How to prove it":

Proof by example Prove the case n = 2 and suggests that it contains most of the ideas of the general proof.

Proof by intimidation 'Trivial'.

Proof by cumbersome notation Best done with access to at least four alphabets and special symbols.

Proof by reference to inaccessible literature a simple corollary of a theorem to be found in a privately circulated memoir of the Slovenian Philological Society, 1883.

Proof by personal communication 'Eight-dimensional colored cycle stripping is NP-complete [Karp, personal communication] (in the elevator).'

Proof by appeal to intuition Cloud-shaped drawings.

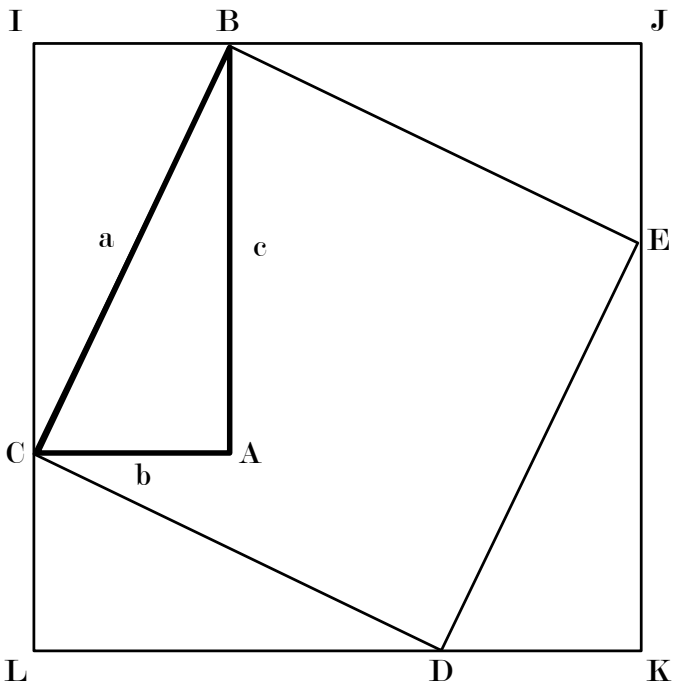
A semantic proof

A map of London is place on the ground of Trafalgar Square. There is a point on the map that is directly above the point on the ground that it represents.

Proof.

The map is directly above a part of London. Thus the entire map is directly above the part of the area which it represents. Now, the smaller area of the map representing Central London is also above the part of the area which it represents. Within the area representing Central London, Trafalgar Square is marked, and this yet smaller part of the map is directly above the part it represents. Continuing this way, we can find smaller and smaller areas of the map each of which is directly above the part of the area which it represents. In the limit we reduce the area on the map to a single point. □

Proof of Pythagoras's Theorem



Let ABC be a triangle with $\widehat{BAC} = 90^o$. Let the lengths of BC, AC, AB be, respectively, a, b, and c. We wish to prove that $a^2 = b^2 + c^2$. Construct a square $IJKL$, of side $b + c$, and a square $BCDE$, of side a. Clearly, $area(IJKL) = (b + c)^2$. But

$$\begin{aligned} area(IJKL) = & \ area(BCDE) + \\ & 4 \times area(ABC) \\ = & \ a^2 + abc. \end{aligned}$$

That is, $(b + c)^2 = a^2 + 2bc$, whence $b^2 + c^2 = a^2$.

Informal v.s. Formal Proofs

- To read an informal proof, we are expected to have a good understanding of the problem domain, the meaning of the natural language statements, and the language of mathematics.
- A *formal* proof shifts some of the burdens to the "form": the symbols, the syntax, and rules manipulating them. "Let the symbols do the work!"
- Our proof of the swapping program is formal:

$$\{x = A \ \wedge \ y = B\}$$
$$x := x - y; \ y := x + y; \ x := y - x$$
$$\{x = B \ \wedge \ y = A\}.$$

Tsuru-Kame Zan

The Tsuru-Kame Problem

Some cranes (tsuru) and tortoises (kame) are mixed in a cage. Known is that there are 5 heads and 14 legs. Find out the numbers of cranes and tortoises.

Tsuru-Kame Zan

The Tsuru-Kame Problem

Some cranes (tsuru) and tortoises (kame) are mixed in a cage. Known is that there are 5 heads and 14 legs. Find out the numbers of cranes and tortoises.

- The kindergarten approach: plain simple enumeration!
  - Crane 0, Tortoise 5 . . . No.
  - Crane 1, Tortoise 4 . . . No.
  - Crane 2, Tortoise 3 . . . No.
  - Crane 3, Tortoise 2 . . . Yes!
  - Crane 4, Tortoise 1 . . . No.
  - Crane 5, Tortoise 0 . . . No.

Tsuru-Kame Zan

The Tsuru-Kame Problem

Some cranes (tsuru) and tortoises (kame) are mixed in a cage. Known is that there are 5 heads and 14 legs. Find out the numbers of cranes and tortoises.

- Elementary school: let's do some reasoning ...
  - If all 5 animals were cranes, there ought to be $5 \times 2 = 10$ legs.
  - However, there are in fact 14 legs. The extra 4 legs must belong to some tortoises. There must be $(14 - 10)/2 = 2$ tortoises.
  - So there must be $5 - 2 = 3$ cranes.
- It generalises to larger numbers of heads and legs.
- Given a different problem, we have to come up with another different way to solve it.

Tsuru-Kame Zan

The Tsuru-Kame Problem

Some cranes (tsuru) and tortoises (kame) are mixed in a cage. Known is that there are 5 heads and 14 legs. Find out the numbers of cranes and tortoises.

- Junior high school: algebra!

$$\begin{aligned} x + y &= 5 \\ 2x + 4y &= 14. \end{aligned}$$

- It's a general approach applicable to many other problems ...
- ... and perhaps easier.
- However, it takes efforts to learn!

Another Formal Proof

The calculational logic proofs we have seen were formal:

$$
\begin{array}{ll}
 & \neg(P \leftrightarrow Q) \\
\Leftrightarrow & \quad \{ \text{ unfolding } \neg \ \} \\
 & (P \leftrightarrow Q) \leftrightarrow \bot \\
\Leftrightarrow & \quad \{ \leftrightarrow \text{ associative } \} \\
 & P \leftrightarrow (Q \leftrightarrow \bot) \\
\Leftrightarrow & \quad \{ \text{ folding } \neg \ \} \\
 & P \leftrightarrow \neg Q.
\end{array}
$$

Rather than relying on intuition on truth tables, we try to develop intuition on calculational rules.

Quantifications

- Let $\oplus$ be a commutative, associative operator with identity e, that is,
  - $x \oplus y = y \oplus x$,
  - $x \oplus (y \oplus z) = (x \oplus y) \oplus z$, and
  - $e \oplus x = x = x \oplus e$,

  and let f be a function defined on int.
- We denote $f\ m \oplus f\ (m+1) \oplus \ldots \oplus f\ (n-1)$ by $(\oplus i : m \leq i < n : f\ i)$.
- $(\oplus i : n \leq i < n : f\ i) = e$.
- $(\oplus i : m \leq i < n+1 : f\ i) = (\oplus i : m \leq i < n : f\ i) \oplus f\ n$ if $m \leq n$.
  - We will refer to this rule as to "split off n".

Quantifications in General

General form: $(\oplus i : R : F)$, where R specifies a range. We sometimes write $(\oplus i : R\,i : F\,i)$ to emphasise that they depend on i.

- $(\oplus i : \mathit{false} : F) = e$.
- $(\oplus i : i = x : F\,i) = F\,x$.
- $(\oplus i : R : F) \oplus (\oplus i : S : F) = (\oplus i : R \vee S : F) \oplus (\oplus i : R \wedge S : F)$.
- $(\oplus i : R : F) \oplus (\oplus i : R : G) = (\oplus i : R : F \oplus G)$.
- $(\oplus i, j : R\,i \wedge S\,i\,j : F) = (\oplus i : R\,i : (\oplus j : S\,i\,j : F))$,
- (i, j distinct, j does not occur free in R).

(Of which rule is range splitting a special case?)

Examples

- E.g.
  - $(+i : 3 \leq i < 5 : i^2) = 3^2 + 4^2 = 25$.
  - $(+i, j : 3 \leq i \leq j < 5 : i \times j) = 3 \times 3 + 3 \times 4 + 4 \times 4$.
  - $(\wedge i : 2 \leq i < 9 : \mathit{odd}\ i \Rightarrow \mathit{prime}\ i) = \mathit{true}$.
  - $(\uparrow i : 1 \leq i < 7 : -i^2 + 5i) = 6$ (when $i = 2$ or 3).
- As a convention,
  $(+i : R : F)$ is written $(\Sigma i : R : F)$,
  $(\wedge i : R : F)$ is written $(\forall i : R : F)$, and
  $(\vee i : R : F)$ is written $(\exists i : R : F)$.
- A special rule for $\uparrow$ (or $\downarrow$) and $+$:

$$x + (\uparrow i : R : F\ i) = (\uparrow i : R : x + F\ i).$$

The Number Of ...

- Define $\# : \mathit{Bool} \to \{0, 1\}$:

$$\begin{aligned} \#\ \mathit{false} &= 0 \\ \#\ \mathit{true} &= 1. \end{aligned}$$

- "The number of" quantifier is defined by:

$$(\#i : R\,i : F\,i) = (\Sigma i : R\,i : \#(F\,i)),$$

from which we may derive:

- $(\#i : \mathit{false} : F\,i) = 0$,
- $(\#i : 0 \leq i < n+1 : F\,i) = (\#i : 0 \leq i < n : F\,i) + \#(F\,n)$.

Deriving Programs from Specifications

- From such a specification:

$$
\begin{array}{l}
|[\ \ \textbf{con}\ \mathit{declarations};\\
\quad \{\mathit{preconditions}\}\\
\quad \mathit{prog}\\
\quad \{\mathit{postcondition}\}\\
]|
\end{array}
$$

  we hope to derive *prog*.
- We usually work backwards from the post condition.
- The techniques we are about to learn is mostly about constructing loops and loop invariants.

Conjunctive Postconditions

- When the post condition has the form $P \wedge Q$, one may take one of the conjuncts as the invariant and the other as the guard:
  - $\{P\}\ \mathbf{do}\ \neg Q \to S\ \mathbf{od}\ \{P \wedge Q\}$.
- E.g. consider the specficication:

$$
\begin{array}{l}
|[\ \ \mathbf{con}\ A, B : \mathit{int}; \{0 \leq A \wedge 0 \leq B\} \\
\quad \mathbf{var}\ q, r : \mathit{int}; \\
\quad \mathit{divmod} \\
\quad \{q = A\ \mathbf{div}\ B \wedge r = A\ \mathbf{mod}\ B\} \\
]|.
\end{array}
$$

- The post condition expands to $R :: A = q \times B + r \wedge 0 \leq r \wedge r < B$.

But Which Conjunct to Choose?

- $q = A \textbf{ div } B \ \wedge \ r = A \textbf{ mod } B$ expands to $R : A = q \times B + r \ \wedge \ 0 \leq r \ \wedge \ r < B$, which leads to a number of possibilities:
- $\{0 \leq r \ \wedge \ r < B\} \textbf{ do } A \neq q \times B + r \rightarrow S \textbf{ od } \{R\}$,
- $\{A = q \times B + r \ \wedge \ r < B\} \textbf{ do } 0 > r \rightarrow S \textbf{ od } \{R\}$, or
- $\{A = q \times B + r \ \wedge \ 0 \leq r\} \textbf{ do } r \geq B \rightarrow S \textbf{ od } \{R\}$, etc.

Computing the Quotient and the Remainder

Try $A = q \times B + r \wedge 0 \leq r$ as the invariant and $\neg(r < B)$ as the guard:

$\{P : A = q \times B + r \ \wedge \ 0 \leq r\}$
do $B \leq r \rightarrow$
$\quad \{P \ \wedge \ B \leq r\}$

$\quad \{P\}$
od
$\{P \wedge r < B\}$

Computing the Quotient and the Remainder

Try $A = q \times B + r \wedge 0 \leq r$ as the invariant and $\neg(r < B)$ as the guard:

- P is established by $q, r := 0, A$.

$q, r := 0, A;$
$\{P : A = q \times B + r \ \wedge \ 0 \leq r\}$
do $B \leq r \rightarrow$
$\{P \ \wedge \ B \leq r\}$

$\{P\}$
od
$\{P \wedge r < B\}$

Computing the Quotient and the Remainder

Try $A = q \times B + r \wedge 0 \leq r$ as the invariant and $\neg(r < B)$ as the guard:

- P is established by $q, r := 0, A$.
- Choose r as the bound.

$q, r := 0, A;$
$\{P : A = q \times B + r \ \wedge \ 0 \leq r\}$
do $B \leq r \rightarrow$
$\{P \ \wedge \ B \leq r\}$

$\{P\}$
od
$\{P \wedge r < B\}$

Computing the Quotient and the Remainder

Try $A = q \times B + r \wedge 0 \leq r$ as the invariant and $\neg(r < B)$ as the guard:

$q, r := 0, A;$
$\{P : A = q \times B + r \ \wedge \ 0 \leq r\}$
do $B \leq r \rightarrow$
$\quad \{P \ \wedge \ B \leq r\}$
$\quad r := r - B$
$\quad \{P\}$
od
$\{P \wedge r < B\}$

- P is established by $q, r := 0, A$.
- Choose r as the bound.
- Since $B > 0$, try $r := r - B$:

$$
\begin{aligned}
& P[r - B/r] \\
\Leftrightarrow \ & A = q \times B + r - B \ \wedge \ 0 \leq r - B \\
\Leftrightarrow \ & A = (q - 1) \times B + r \ \wedge \ B \leq r.
\end{aligned}
$$

Computing the Quotient and the Remainder

Try $A = q \times B + r \wedge 0 \leq r$ as the invariant and $\neg(r < B)$ as the guard:

$q, r := 0, A;$
$\{P : A = q \times B + r \ \wedge \ 0 \leq r\}$
do $B \leq r \rightarrow$
$\quad \{P \ \wedge \ B \leq r\}$
$\quad r := r - B$
$\quad \{P\}$
od
$\{P \wedge r < B\}$

- P is established by $q, r := 0, A$.
- Choose r as the bound.
- Since $B > 0$, try $r := r - B$:

$$
\begin{aligned}
& P[r - B/r] \\
\Leftrightarrow \ & A = q \times B + r - B \ \wedge \ 0 \leq r - B \\
\Leftrightarrow \ & A = (q - 1) \times B + r \ \wedge \ B \leq r.
\end{aligned}
$$

- $$
\begin{aligned}
& P[q + 1, r - B/q, r] \\
\Leftrightarrow \ & A = (q + 1) \times B + r - B \ \wedge \ 0 \leq r - B \\
\Leftrightarrow \ & A = q \times B + r \ \wedge \ B \leq r.
\end{aligned}
$$

Computing the Quotient and the Remainder

Try $A = q \times B + r \wedge 0 \leq r$ as the invariant and $\neg(r < B)$ as the guard:

$q, r := 0, A;$
$\{P : A = q \times B + r \ \wedge \ 0 \leq r\}$
do $B \leq r \rightarrow$
$\{P \ \wedge \ B \leq r\}$
$q, r := q + 1, r - B$
$\{P\}$
od
$\{P \wedge r < B\}$

- P is established by $q, r := 0, A$.
- Choose r as the bound.
- Since $B > 0$, try $r := r - B$:

$$\begin{aligned} & P[r - B/r] \\ \Leftrightarrow \ & A = q \times B + r - B \ \wedge \ 0 \leq r - B \\ \Leftrightarrow \ & A = (q - 1) \times B + r \ \wedge \ B \leq r. \end{aligned}$$

- $$\begin{aligned} & P[q + 1, r - B/q, r] \\ \Leftrightarrow \ & A = (q + 1) \times B + r - B \ \wedge \ 0 \leq r - B \\ \Leftrightarrow \ & A = q \times B + r \ \wedge \ B \leq r. \end{aligned}$$

Exponentiation

- Consider the problem:

$$
\begin{array}{l}
[\![\ \textbf{con}\ A, B : \mathit{int}\ \{A \geq 0\ \wedge\ B \geq 0\}; \\
\quad \textbf{var}\ r : \mathit{int}; \\
\quad \mathit{exponentiation} \\
\quad \{r = A^B\} \\
]\!].
\end{array}
$$

- There is not much we can do with a state space consisting of only one variable.
- Replacing constants by variables may yield some possible invariants.
- Again we have several choices: $r = x^B$, $r = A^x$, $r = x^y$, etc.

Exponentiation

- Use the invariant $P_0 : r = A^x$, thus $P_0 \ \wedge \ x = B$ implies the post-condition.

$\{r = A^x$ $\}$
do $x \neq B \rightarrow$

od
$\{r = A^B\}$

Exponentiation

- Use the invariant $P_0 : r = A^x$, thus $P_0 \ \wedge \ x = B$ implies the post-condition.

$$
\begin{array}{l}
\quad r, x := 1, 0 \\
\quad \{r = A^x \qquad\qquad\qquad\qquad \} \\
; \ \mathbf{do}\ x \neq B \rightarrow \\
\\
\quad \mathbf{od} \\
\quad \{r = A^B\}
\end{array}
$$

Exponentiation

- Use the invariant $P_0 : r = A^x$, thus $P_0 \ \wedge \ x = B$ implies the post-condition.
- Strategy: increment x in the loop. An upper bound $P_1 : x \leq B$.

$r, x := 1, 0$
$\{r = A^x \ \wedge \ x \leq B, bnd : B - x\}$
; **do** $x \neq B \rightarrow$

$; x := x + 1$
od
$\{r = A^B\}$

Exponentiation

- Use the invariant $P_0 : r = A^x$, thus $P_0 \ \wedge \ x = B$ implies the post-condition.
- Strategy: increment x in the loop. An upper bound $P_1 : x \leq B$.
- $(r = A^x)[x + 1/x] \Leftrightarrow r = A^{x+1}$. However, when $r = A^x$ holds, $A^{x+1} = A \times A^x = A \times r$!

$r, x := 1, 0$
$\{r = A^x \ \wedge \ x \leq B, bnd : B - x\}$
; **do** $x \neq B \rightarrow$

$\{r = A^{x+1} \ \wedge \ x + 1 \leq B\}$
$; x := x + 1$
od
$\{r = A^B\}$

Exponentiation

- Use the invariant $P_0 : r = A^x$, thus $P_0 \ \wedge \ x = B$ implies the post-condition.
- Strategy: increment x in the loop. An upper bound $P_1 : x \leq B$.
- $(r = A^x)[x + 1/x] \Leftrightarrow r = A^{x+1}$. However, when $r = A^x$ holds, $A^{x+1} = A \times A^x = A \times r$!
- Indeed, $(r = A^{x+1})[A \times r/r]$
  $\Leftrightarrow \ A \times r = A^{x+1}$
  $\Leftarrow \ r = A^x$.

$r, x := 1, 0$
$\{r = A^x \ \wedge \ x \leq B, bnd : B - x\}$
; **do** $x \neq B \rightarrow$
  $r := A \times r$
  $\{r = A^{x+1} \ \wedge \ x + 1 \leq B\}$
  $; x := x + 1$
od
$\{r = A^B\}$

Constructing Loop Body in Steps

We will see this pattern often:

- we have discovered that $(r = e)[x + 1/x] \;\Leftrightarrow\; r = e \oplus e'$.
- We want to establish:

$$\{r = e \;\wedge\; \ldots\}$$

$$\{r = e \oplus e'\}$$
$$; x := x + 1$$
$$\{r = e\}.$$

Constructing Loop Body in Steps

We will see this pattern often:

- we have discovered that $(r = e)[x + 1/x] \;\Leftrightarrow\; r = e \oplus e'$.
- We want to establish:

$$
\begin{array}{l}
\{r = e \;\wedge\; \ldots\} \\
r := r \oplus e' \\
\{r = e \oplus e'\} \\
; x := x + 1 \\
\{r = e\}.
\end{array}
$$

Constructing Loop Body in Steps

We will see this pattern often:

- we have discovered that $(r = e)[x + 1/x] \;\Leftrightarrow\; r = e \oplus e'$.
- We want to establish:

$$\begin{array}{l} \{r = e \;\wedge\; \ldots\} \\ r := r \oplus e' \\ \{r = e \oplus e'\} \\ ; x := x + 1 \\ \{r = e\}. \end{array}$$

- It works because:

$$\begin{array}{ll} & (r = e \oplus e')[r \oplus e'/r] \\ \Leftrightarrow & r \oplus e' = e \oplus e' \\ \Leftarrow & r = e. \end{array}$$

Summing Up an Array

- Another simple exercise.
- We talk about it because we need range splitting.

$$
\begin{array}{l}
|[\ \textbf{con}\ N : \mathit{int}\ \{0 \le N\};\quad f : \textbf{array}\ [0..N)\ \textbf{of}\ \mathit{int}; \\
\quad \textbf{var}\ x : \mathit{int} \\
\quad \mathit{sum} \\
\quad \{x = (\Sigma i : 0 \le i < N : f[i])\} \\
]|
\end{array}
$$

Summing Up an Array

$$
\begin{array}{l}
|[\ \textbf{con}\ N : \mathit{int}\ \{0 \le N\};\quad f : \textbf{array}\ [0..N)\ \textbf{of}\ \mathit{int};\\
\\
\quad \{P : x = (\Sigma i : 0 \le i < n : f[i]),\ \mathit{bnd} : N - n\}\\
\ ;\textbf{do}\ n \neq N \to \{P \wedge n \neq N\} \qquad\qquad \{P\}\ \textbf{od}\\
\quad \{x = (\Sigma i : 0 \le i < N : f[i])\}\\
]|
\end{array}
$$

Summing Up an Array

$$
\begin{array}{l}
|[\ \textbf{con}\ N : \mathit{int}\ \{0 \le N\};\quad f : \textbf{array}\ [0..N)\ \textbf{of}\ \mathit{int};\\
\\
\quad n, x := 0, 0\\
\quad \{P : x = (\Sigma i : 0 \le i < n : f[i]), \mathit{bnd} : N - n\}\\
\ ;\textbf{do}\ n \neq N \to \{P \wedge n \neq N\} \qquad\qquad \{P\}\ \textbf{od}\\
\quad \{x = (\Sigma i : 0 \le i < N : f[i])\}\\
]|
\end{array}
$$

Summing Up an Array

$$
\begin{array}{l}
|[\ \textbf{con}\ N : \mathit{int}\ \{0 \le N\};\quad f : \textbf{array}\ [0..N)\ \textbf{of}\ \mathit{int};\\
\\
\quad n, x := 0, 0\\
\quad \{P : x = (\Sigma i : 0 \le i < n : f[i]), \mathit{bnd} : N - n\}\\
\ ;\textbf{do}\ n \neq N \to \{P \wedge n \neq N\} \qquad\qquad\qquad\qquad n := n + 1\ \ \{P\}\ \textbf{od}\\
\quad \{x = (\Sigma i : 0 \le i < N : f[i])\}\\
]|
\end{array}
$$

- Use $N - n$ as bound, try incrementing n:

$$
\begin{array}{rl}
 & (x = (\Sigma i : 0 \le i < n : f[i]))[n + 1/n]\\
\Leftrightarrow & x = (\Sigma i : 0 \le i < n + 1 : f[i])\\
\\
\Leftrightarrow & x = (\Sigma i : 0 \le i < n : f[i]) + f[n] \qquad\qquad .
\end{array}
$$

Summing Up an Array

$$
\begin{array}{l}
|[\ \textbf{con}\ N : \mathit{int}\ \{0 \le N\};\quad f : \textbf{array}\ [0..N)\ \textbf{of}\ \mathit{int};\\
\\
\quad n, x := 0, 0\\
\quad \{P : x = (\Sigma i : 0 \le i < n : f[i])\ \wedge\ 0 \le n, \mathit{bnd} : N - n\}\\
;\textbf{do}\ n \neq N \to \{P \wedge n \neq N\} \qquad\qquad n := n + 1\ \ \{P\}\ \textbf{od}\\
\quad \{x = (\Sigma i : 0 \le i < N : f[i])\}\\
]|
\end{array}
$$

- Use $N - n$ as bound, try incrementing n:

$$
\begin{array}{rl}
& (x = (\Sigma i : 0 \le i < n : f[i])\ \wedge\ 0 \le n)[n+1/n]\\
\Leftrightarrow & x = (\Sigma i : 0 \le i < n+1 : f[i])\ \wedge\ 0 \le n+1\\
\Leftarrow & x = (\Sigma i : 0 \le i < n+1 : f[i])\ \wedge\ 0 \le n\\
\Leftrightarrow & x = (\Sigma i : 0 \le i < n : f[i]) + f[n]\ \wedge\ 0 \le n.
\end{array}
$$

Summing Up an Array

$$
\begin{array}{l}
|[\ \textbf{con}\ N : \mathit{int}\ \{0 \le N\};\quad f : \textbf{array}\ [0..N)\ \textbf{of}\ \mathit{int};\\
\\
\quad n, x := 0, 0\\
\quad \{P : x = (\Sigma i : 0 \le i < n : f[i])\ \wedge\ 0 \le n, \mathit{bnd} : N - n\}\\
;\textbf{do}\ n \neq N \to \{P \wedge n \neq N\} \qquad\qquad\qquad\qquad n := n + 1\ \ \{P\}\ \textbf{od}\\
\quad \{x = (\Sigma i : 0 \le i < N : f[i])\}\\
]|
\end{array}
$$

- $(x = (\Sigma i : 0 \le i < n : f[i]) + f[n]\ \wedge\ 0 \le n)$

$\Leftarrow\quad x = (\Sigma i : 0 \le i < n : f[i])\ \wedge\ 0 \le n.$

Summing Up an Array

$$
\begin{array}{l}
|[\ \textbf{con}\ N : \mathit{int}\ \{0 \le N\};\quad f : \textbf{array}\ [0..N)\ \textbf{of}\ \mathit{int};\\
\\
\quad n, x := 0, 0\\
\quad \{P : x = (\Sigma i : 0 \le i < n : f[i])\ \wedge\ 0 \le n, \mathit{bnd} : N - n\}\\
;\textbf{do}\ n \neq N \to \{P \wedge n \neq N\}\ \ x := x + f[n];\quad n := n + 1\ \ \{P\}\ \textbf{od}\\
\quad \{x = (\Sigma i : 0 \le i < N : f[i])\}\\
]|
\end{array}
$$

- $$
\begin{array}{ll}
 & (x = (\Sigma i : 0 \le i < n : f[i]) + f[n]\ \wedge\ 0 \le n)[x + f[n]/x]\\
\Leftrightarrow & x + f[n] = (\Sigma i : 0 \le i < n : f[i]) + f[n]\ \wedge\ 0 \le n\\
\Leftarrow & x = (\Sigma i : 0 \le i < n : f[i])\ \wedge\ 0 \le n.
\end{array}
$$

No. of Pairs in an Array

$$|[\ \textbf{con}\ N : \mathit{int}\{N \geq 0\}; a : \textbf{array}\,[0..N)\,\textbf{of}\ \mathit{int};$$
$$\quad \textbf{var}\ r : \mathit{int};$$
$$\quad S$$
$$\quad \{r = (\#i, j : 0 \leq i < j < N : a[i] \leq 0 \wedge a[j] \geq 0)\}$$
$$]|.$$

- Replace N by n:

$$P_0 :\ \ r = (\#i, j : 0 \leq i < j < n : a[i] \leq 0 \wedge a[j] \geq 0),$$
$$P_1 :\ \ 0 \leq n \leq N.$$

- Initialisation: $n, r := 0, 0$.

No. of Pairs in an Array

$$
\begin{array}{l}
|[\ \textbf{con}\ N : \mathit{int}\ \{N \geq 0\};\ \ a : \textbf{array}\,[0..N)\,\textbf{of}\ \mathit{int};\\
\quad \textbf{var}\ r : \mathit{int};\\
\quad n, r := 0, 0\\
\quad \{P_0\ \wedge\ P_1, \mathit{bnd} : N - n\}\\
\ ;\textbf{do}\ n \neq N \rightarrow \ldots n := n + 1\ \textbf{od}\\
\quad \{r = (\#i, j : 0 \leq i < j < N : a[i] \leq 0 \wedge a[j] \geq 0)\}\\
]|.
\end{array}
$$

- Replace N by n:

$$
\begin{array}{l}
P_0 :\ \ r = (\#i, j : 0 \leq i < j < n : a[i] \leq 0 \wedge a[j] \geq 0),\\
P_1 :\ \ 0 \leq n \leq N.
\end{array}
$$

- Initialisation: $n, r := 0, 0$.

No. of Pairs in an Array

To reason about $P_0[n+1/n]$, we calculate (assuming P_0, $P_1 : 0 \leq n \leq N$ and $n \neq N$):

$$
\begin{array}{ll}
 & (\#i,j : 0 \leq i < j < n+1 : a[i] \leq 0 \wedge a[j] \geq 0) \\
= & \quad \{ \text{ split off } j = n \ \} \\
 & (\#i,j : 0 \leq i < j < n : a[i] \leq 0 \wedge a[j] \geq 0) + \\
 & (\#i : 0 \leq i < n : a[i] \leq 0 \wedge a[n] \geq 0) \\
= & \quad \{ \ P_0 \ \} \\
 & r + (\#i : 0 \leq i < n : a[i] \leq 0 \wedge a[n] \geq 0) \\
= & \begin{cases} r, & \text{if } a[n] < 0; \\ r + (\#i : 0 \leq i < n : a[i] \leq 0), & \text{if } a[n] \geq 0. \end{cases}
\end{array}
$$

We could compute $(\#i : 0 \leq i < n : a[i] \leq 0)$ in a loop. . . or can we store it in another variable?

Strengthening by Using More Variables

New plan:

$$
\begin{array}{l}
|[\ \textbf{con}\ N : \mathit{int}\ \{N \geq 0\};\ \ a : \textbf{array}\,[0..N)\,\textbf{of}\ \mathit{int};\\
\quad \textbf{var}\ r, s : \mathit{int};\\
\\
\quad n, r, s := 0, 0, 0\\
\quad \{P_0\ \wedge\ P_1\ \wedge\ Q, \mathit{bnd} : N - n\}\\
\ ;\textbf{do}\ n \neq N \rightarrow \ldots n := n + 1\ \textbf{od}\\
\quad \{r = (\#i, j : 0 \leq i < j < N : a[i] \leq 0 \wedge a[j] \geq 0)\}\\
]|.
\end{array}
$$

$$
\begin{array}{l}
P_0 :\ \ r = (\#i, j : 0 \leq i < j < n : a[i] \leq 0 \wedge a[j] \geq 0),\\
P_1 :\ \ 0 \leq n \leq N,\\
Q :\ \ s = (\#i : 0 \leq i < n : a[i] \leq 0).
\end{array}
$$

Update the New Variable

$$
\begin{aligned}
& (\#i : 0 \le i < n : a[i] \le 0)[n+1/n] \\
= \; & (\#i : 0 \le i < n+1 : a[i] \le 0) \\
= \; & \quad \{ \text{ split off } i = n \text{ (assuming } 0 \le n) \ \} \\
& (\#i : 0 \le i < n : a[i] \le 0) + \#(a[i] \le 0) \\
= \; & \quad \{ \ Q \ \} \\
& s + \#(a[i] \le 0) \\
= \; & \begin{cases} s & \text{if } a[i] > 0, \\ s+1 & \text{if } a[i] \le 0. \end{cases}
\end{aligned}
$$

Resulting Program

$$
\begin{array}{l}
n, r, s := 0, 0, 0 \\
\{P_0 \wedge P_1 \wedge Q, bnd : N - n\} \\
\textbf{;do } n \neq N \rightarrow \{P_0 \wedge P_1 \wedge Q \wedge n \neq N\} \\
\quad \textbf{if } a[n] < 0 \rightarrow skip \\
\quad \quad [\!] \ a[n] \geq 0 \rightarrow r := r + s \\
\quad \textbf{fi} \\
\quad \{P_0[n+1/n] \wedge P_1 \wedge Q \wedge n \neq N\} \\
\quad \textbf{;if } a[n] > 0 \rightarrow skip \\
\quad \quad [\!] \ a[n] \leq 0 \rightarrow s := s + 1 \\
\quad \textbf{fi} \\
\quad \{(P_0 \wedge P_1 \wedge Q)[n+1/n]\} \\
\quad ; n := n + 1 \\
\textbf{od} \\
\{r = (\#i, j : 0 \leq i < j < N : a[i] \leq 0 \wedge a[j] \geq 0)\}
\end{array}
$$

Resulting Program

Since $P_0 \wedge P_1 \wedge Q \wedge n \neq N$ is a common precondition for the **if**'s (the second **if** does not use P_0), they can be combined:

$$
\begin{array}{l}
n, r, s := 0, 0, 0 \\
\{P_0 \wedge P_1 \wedge Q, bnd : N - n\} \\
;\mathbf{do}\ n \neq N \rightarrow \{P_0 \wedge P_1 \wedge Q \wedge n \neq N\} \\
\quad \mathbf{if}\ a[n] < 0 \rightarrow s := s + 1 \\
\quad\ [\!]\ a[n] = 0 \rightarrow r, s := r + s, s + 1 \\
\quad\ [\!]\ a[n] > 0 \rightarrow r := r + s \\
\quad \mathbf{fi} \\
\quad \{(P_0 \wedge P_1 \wedge Q)[n + 1/n]\} \\
\quad ; n := n + 1 \\
\mathbf{od} \\
\{r = (\#i, j : 0 \leq i < j < N : a[i] \leq 0 \wedge a[j] \geq 0)\}
\end{array}
$$

It's Easier to Do More?

- The resulting loop computes values for two variables rather than one. It appears that it does more work.
- However, we often find that a loop that does more is eaiser to construct, because more has been established in the previous iteration of the loop.
- The invariant is "stronger" because it promises more.
- It is a common phenomena: a generalised theorem is easier to prove.
- We will see another way to generalise the invariant in the next section.

Isn't It Getting A Bit Too Complicated?

- Quantifier and indexes manipulation tend to get very long and tedious.
  - Expect to see even longer expressions later!
- With long and complex expressions, one tend to make mistakes.
- To certain extent, it is a restriction of the data structure we are using. With arrays we have to manipulate the indexes.
- Is it possible to use higher-level data structures? Lists? Trees?
  - Like *map*, *filter*, *foldr*...in functional programming?
  - More on this issue later.

Fibonacci

Recall: $fib\ 0 = 0$, $fib\ 1 = 1$, and $fib\ (n+2) = fib\ n + fib\ (n+1)$.

$$
\begin{array}{l}
|[\ \textbf{con}\ N : int\ \{0 \le N\};\quad \textbf{var}\ x\quad : int;\\
\quad n, x := 0, 0\\
\quad \{P : x = fib\ n \wedge 0 \le n \le N \qquad\qquad\qquad \}\\
\quad ;\textbf{do}\ n \ne N \to\ \{P \wedge n \ne N\} \qquad\qquad\qquad n := n+1\ \ \{P\}\\
\quad \textbf{od}\\
\quad \{x = fib\ N\}\ \ ]|.
\end{array}
$$

- $$
\begin{array}{ll}
 & (x = fib\ n \wedge 0{\le}n{\le}N)[n{+}1/n]\\
\Leftrightarrow & x = fib\ (n{+}1) \wedge 0{\le}n{<}N
\end{array}
$$

Fibonacci

Recall: $\mathit{fib}\ 0 = 0$, $\mathit{fib}\ 1 = 1$, and $\mathit{fib}\ (n+2) = \mathit{fib}\ n + \mathit{fib}\ (n+1)$.

$$
\begin{array}{l}
|[\ \mathbf{con}\ N : \mathit{int}\ \{0 \leq N\};\quad \mathbf{var}\ x\ : \mathit{int}; \\
\quad n, x := 0, 0 \\
\quad \{P : x = \mathit{fib}\ n \wedge 0 \leq n \leq N \qquad\qquad \} \\
\ ;\mathbf{do}\ n \neq N \rightarrow\ \{P \wedge n \neq N\} \qquad\qquad n := n+1\ \ \{P\} \\
\quad \mathbf{od} \\
\quad \{x = \mathit{fib}\ N\}\ \ ]|.
\end{array}
$$

- $$
\begin{array}{rl}
 & (x = \mathit{fib}\ n \wedge 0 \leq n \leq N)[n{+}1/n] \\
\Leftrightarrow & x = \mathit{fib}\ (n{+}1) \wedge 0 \leq n < N
\end{array}
$$
- $$
\begin{array}{rl}
 & (x = \mathit{fib}\ (n{+}1) \wedge \ldots) \\
\Leftarrow & x = \mathit{fib}\ n \wedge \ldots.
\end{array}
$$

Fibonacci

Recall: $fib\ 0 = 0$, $fib\ 1 = 1$, and $fib\ (n+2) = fib\ n + fib\ (n+1)$.

$$
\begin{array}{l}
|[\ \textbf{con}\ N : int\ \{0 \le N\};\quad \textbf{var}\ x, y : int; \\
\quad n, x, y := 0, 0, 1 \\
\quad \{P : x = fib\ n \wedge 0 \le n \le N \wedge \quad y = fib\ (n+1)\} \\
;\textbf{do}\ n \neq N \to\ \{P \wedge n \neq N\} \qquad\qquad\qquad\qquad n := n+1\ \ \{P\} \\
\quad \textbf{od} \\
\quad \{x = fib\ N\}\ \ ]|.
\end{array}
$$

- $$
  \begin{array}{rl}
  & (x = fib\ n \wedge 0 \le n \le N \wedge y = fib\ (n+1))[n+1/n] \\
  \Leftrightarrow & x = fib\ (n+1) \wedge 0 \le n < N \wedge y = fib\ (n+2)
  \end{array}
  $$
- $$(x = fib\ (n+1) \wedge \ldots \wedge y = fib\ (n+2))$$

$$\Leftarrow \quad x = fib\ n \wedge \ldots \wedge y = fib\ (n+1).$$

Fibonacci

Recall: $fib\ 0 = 0$, $fib\ 1 = 1$, and $fib\ (n+2) = fib\ n + fib\ (n+1)$.

$$
\begin{array}{l}
|[\ \textbf{con}\ N : int\ \{0 \le N\};\quad \textbf{var}\ x, y : int; \\
\quad n, x, y := 0, 0, 1 \\
\quad \{P : x = fib\ n \wedge 0 \le n \le N \wedge \quad y = fib\ (n+1)\} \\
\quad ;\textbf{do}\ n \neq N \to\ \{P \wedge n \neq N\}\ x, y := y, x+y; \quad n := n+1\ \ \{P\} \\
\quad \textbf{od} \\
\quad \{x = fib\ N\}\ \ ]|.
\end{array}
$$

- $$
\begin{array}{rl}
 & (x = fib\ n \wedge 0 \le n \le N \wedge y = fib\ (n+1))[n+1/n] \\
\Leftrightarrow & x = fib\ (n+1) \wedge 0 \le n < N \wedge y = fib\ (n+2)
\end{array}
$$

- $$
\begin{array}{rl}
 & (x = fib\ (n+1) \wedge \ldots \wedge y = fib\ (n+2))[y, x+y/x, y] \\
\Leftrightarrow & y = fib\ (n+1) \wedge \ldots \wedge x + y = fib\ (n+2) \\
\Leftarrow & x = fib\ n \wedge \ldots \wedge y = fib\ (n+1).
\end{array}
$$

Tail Recursion

- A function f is *tail recursive* if it looks like:

$$\begin{array}{lll} f\ x & = h\ x, & \text{if } b\ x; \\ f\ x & = f\ (g\ x), & \text{if } \neg(b\ x). \end{array}$$

- The goal is to derive a program that computes $f\ X$ for given X. Plan:

$$\begin{array}{l} |[\ \mathbf{con}\ X;\ \mathbf{var}\ r, x; \\ \quad x := X \\ \quad \{f\ x = f\ X\} \\ \quad ;\mathbf{do}\ \neg(b\ x) \to x := g\ x\ \mathbf{od} \\ \quad ; r := h\ x \\ \quad \{r = f\ X\} \\ ]|, \end{array}$$

provided that the loop terminates.

Using Associativity

- Consider function k such that:

$$\begin{array}{lll} k\ x & = a, & \text{if } b\ x; \\ k\ x & = h\ x \oplus k\ (g\ x), & \text{if } \neg(b\ x). \end{array}$$

  where $\oplus$ is associative with identity e. Note that k is not tail recursive.
- Goal: establish $r = k\ X$ for given X.
- Trick: use an invariant $r \oplus k\ x = k\ X$.
  - 'computed' $\oplus$ 'to be computed' $= k\ X$.
  - Strategy: keep shifting stuffs from right hand side of $\oplus$ to the left, until the right is e.

Constructing the Loop Body

If $b\ x$ holds:

$$
\begin{array}{ll}
 & r \oplus k\ x = k\ X \\
\Leftrightarrow & \quad \{\ \ b\ x\ \ \} \\
 & r \oplus a = k\ X.
\end{array}
$$

Otherwise:

$$
\begin{array}{ll}
 & r \oplus k\ x = k\ X \\
\Leftrightarrow & \quad \{\ \ \neg(b\ x)\ \ \} \\
 & r \oplus (h\ x \oplus k\ (g\ x)) = k\ X \\
\Leftrightarrow & \quad \{\ \ \oplus \text{ associative}\ \ \} \\
 & (r \oplus h\ x) \oplus k\ (g\ x) = k\ X \\
\Leftrightarrow & (r \oplus k\ x = k\ X)[r \oplus h\ x, g\ x/r, x].
\end{array}
$$

The Program

$$
\begin{array}{l}
|[\ \textbf{con}\ X;\ \textbf{var}\ r, x; \\
\quad\\
\quad r, x := e, X \\
\quad \{r \oplus k\ x = k\ X\} \\
;\textbf{do}\ \neg(b\ x) \to r, x := r \oplus h\ x, g\ x\ \textbf{od} \\
\quad \{r \oplus a = k\ X\} \\
;r := r \oplus a \\
\quad \{r = k\ X\} \\
]|,
\end{array}
$$

if the loop terminates.

Exponentation Again

- Consider again computing A^B. Notice that:

$$\begin{array}{rcll} x^0 & = & 1 & \\ x^y & = & 1 \times (x \times x)^{y \ \mathbf{div}\ 2} & \text{if } \textit{even } y, \\ & = & x \times x^{y-1} & \text{if } \textit{odd } y. \end{array}$$

- How does it fit the pattern above? (Hint: k now has type $(\textit{int} \times \textit{int}) \to \textit{int}$.)
- To be concrete, let us look at this specialised case in more detail.

Fast Exponentiation

- To achieve $r = A^B$, choose invariant $r \times x^y = A^B$:
- To construct the loop body, we reason

for the case *even* y:

$$\begin{array}{ll} & r \times x^y \\ = & \{ \text{ assumption: } \textit{even } y \} \\ & r \times (x \times x)^{y \,\mathbf{div}\, 2} \\ = & (r \times x^y)[x \times x, y \,\mathbf{div}\, 2/x, y]. \end{array}$$

and for *odd* y:

$$\begin{array}{ll} & r \times x^y \\ = & \{ \text{ assumption: } \textit{odd } y \} \\ & r \times (x \times x^{y-1}) \\ = & \{ \times \text{ associative} \} \\ & (r \times x) \times x^{y-1} \\ = & (r \times x^y)[r \times x, y - 1/r, y]. \end{array}$$

Fast Exponentiation

The resulting program:

$r, x, y := 1, A, B;$
$\{r \times x^y = A^B \wedge 0 \leq y, bnd = y\}$
do $y \neq 0 \wedge even\ y \rightarrow x, y := x \times x, y$ **div** 2
$[\!]\ y \neq 0 \wedge odd\ y \rightarrow r, y := r \times x, y - 1$
od
$\{r \times x^y = A^B \wedge y = 0\}.$

Specification

$$|[\ \textbf{con}\ N : int\{0 \le N\};\quad f : \textbf{array}\ [0..N)\ \textbf{of}\ int;$$
$$\quad \textbf{var}\ r\quad : int;$$

$$\quad \{r = (\uparrow p, q : 0 \le p \le q \le N : sum\ p\ q)\}$$
$$]|$$

- $sum\ p\ q = (\Sigma i : p \le i < q : f[i])$.

Specification

$$
\begin{array}{l}
|[\ \textbf{con}\ N : int\{0 \le N\};\quad f : \textbf{array}\ [0..N)\ \textbf{of}\ int; \\
\quad \textbf{var}\ r, n : int; \\
\\
\quad n, r := 0, 0 \\
\quad \{r = (\uparrow p, q : 0{\le}p{\le}q{\le}n : sum\ p\ q) \wedge 0 \le n \le N\} \\
\quad ;\textbf{do}\ \ n \neq N \rightarrow \\
\qquad\qquad \ldots ; n := n + 1 \\
\quad \textbf{od} \\
\quad \{r = (\uparrow p, q : 0{\le}p{\le}q{\le}N : sum\ p\ q)\} \\
]|
\end{array}
$$

- $sum\ p\ q = (\Sigma i : p{\le}i{<}q : f[i])$.
- Replacing constant N by variable n, use an up-loop.

Strengthening the Invariant

- Let $P_0 : r = (\uparrow p, q : 0 \leq p \leq q \leq n : sum\ p\ q)$.

$$
\begin{array}{l}
n, r \quad := 0, 0 \quad ; \\
\{P_0 \wedge 0 \leq n \leq N \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \} \\
\textbf{do} \quad n \neq N - > \\
\qquad\qquad \ldots ; n := n + 1 \\
\textbf{od} \\
\{r = (\uparrow p, q : 0 \leq p \leq q \leq N : sum\ p\ q)\}
\end{array}
$$

- With assumption that $0 \leq n + 1 \leq N$:

$$
\begin{array}{ll}
 & (\uparrow p, q : 0 \leq p \leq q \leq n : sum\ p\ q)[n + 1/n] \\
= & (\uparrow p, q : 0 \leq p \leq q \leq n + 1 : sum\ p\ q) \\
= & (\uparrow p, q : 0 \leq p \leq q \leq n : sum\ p\ q) \quad \uparrow \\
 & \qquad\qquad (\uparrow p, q : 0 \leq p \leq n{+}1 : sum\ p\ (n{+}1)).
\end{array}
$$

Strengthening the Invariant

- Let $P_0 : r = (\uparrow p, q : 0 \le p \le q \le n : sum\ p\ q)$.

$$
\begin{array}{l}
n, r, s := 0, 0, 0; \\
\{P_0 \wedge 0 \le n \le N \wedge s = (\uparrow p : 0 \le p \le n : sum\ p\ n)\} \\
\textbf{do}\ \ n \neq N - > \\
\qquad \ldots ; n := n + 1 \\
\textbf{od} \\
\{r = (\uparrow p, q : 0 \le p \le q \le N : sum\ p\ q)\}
\end{array}
$$

- With assumption that $0 \le n + 1 \le N$:

$$
\begin{array}{ll}
 & (\uparrow p, q : 0 \le p \le q \le n : sum\ p\ q)[n + 1/n] \\
= & (\uparrow p, q : 0 \le p \le q \le n + 1 : sum\ p\ q) \\
= & (\uparrow p, q : 0 \le p \le q \le n : sum\ p\ q)\ \ \uparrow \\
 & \qquad (\uparrow p, q : 0 \le p \le n{+}1 : sum\ p\ (n{+}1)).
\end{array}
$$

- Let's introduce $P_1 : s = (\uparrow p : 0 \le p \le n : sum\ p\ n)$.

Constructing the Loop Body

- Known: $P_0 : r = (\uparrow p, q : 0 \leq p \leq q \leq n : sum\ p\ q)$,
- $P_1 : s = (\uparrow p : 0 \leq p \leq n : sum\ p\ n)$,
- $P_0[n+1/n] : r = (\uparrow p, q : 0 \leq p \leq q \leq n : sum\ p\ q) \uparrow (\uparrow p : 0 \leq p \leq n{+}1 : sum\ p\ (n{+}1))$.
- Therefore, a possible strategy would be:

$$
\begin{array}{l}
\{P_0 \ \wedge \ P_1 \ \wedge \ 0 \leq n \leq N \ \wedge \ n \neq N\} \\
s := \ ?; \\
\{P_0 \ \wedge \ P_1[n+1/n] \ \wedge \ 0 \leq n \leq N \ \wedge \ n \neq N\} \\
r := r \uparrow s; \\
\{(P_0 \ \wedge \ P_1 \ \wedge \ 0 \leq n \leq N)[n+1/n]\} \\
n := n+1 \\
\{P_0 \ \wedge \ P_1 \ \wedge \ 0 \leq n \leq N\}
\end{array}
$$

Updating the Prefix Sum

Recall $P_1 \equiv s = (\uparrow p : 0 \le p \le n : sum\ p\ n)$.

$$
\begin{array}{ll}
 & (\uparrow p : 0 \le p \le n : sum\ p\ n)[n+1/n] \\
= & (\uparrow p : 0 \le p \le n{+}1 : sum\ p\ (n{+}1)) \\
= & \quad \{ \text{ splitting } p = n+1 \ \} \\
 & (\uparrow p : 0 \le p \le n : sum\ p\ (n{+}1)) \uparrow \\
 & \qquad\qquad sum\ (n{+}1)\ (n{+}1) \\
= & \quad \{ \ [n+1, n+1) \text{ is an empty range } \} \\
 & (\uparrow p : 0 \le p \le n : sum\ p\ (n{+}1)) \uparrow 0 \\
= & (\uparrow p : 0 \le p \le n : sum\ p\ n + f[n]) \uparrow 0 \\
= & ((\uparrow p : 0 \le p \le n : sum\ p\ n) + f[n]) \uparrow 0.
\end{array}
$$

Thus, $\{P_1\}\, s :=? \{P_1[n+1/n]\}$ is satisfied by $s := (s + f[n]) \uparrow 0$.

Derived Program

$$
\begin{array}{l}
|[\ \textbf{con}\ N : \mathit{int}\ \{0 \le N\};\quad f : \textbf{array}\ [0..N)\ \textbf{of}\ \mathit{int};\\
\quad \textbf{var}\ r, s, n : \mathit{int};\\
\\
\quad n, r, s := 0, 0, 0\\
\quad \{P_0 \wedge P_1 \wedge 0 \le n \le N, \mathit{bnd} : N - n\}\\
\quad ;\textbf{do}\ \ n \ne N \to\\
\qquad\qquad s := (s + f[n]) \uparrow 0;\\
\qquad\qquad r := r \uparrow s;\\
\qquad\qquad n := n + 1\\
\quad \textbf{od}\\
\quad \{r = (\uparrow 0 \le p \le q \le N : \mathit{sum}\ p\ q :)\}\\
]|
\end{array}
$$

- $P_0 : r = (\uparrow p, q : 0 \le p \le q \le n : \mathit{sum}\ p\ q)$.
- $P_1 : s = (\uparrow p, q : 0 \le p \le n : \mathit{sum}\ p\ n)$.

What Have We Learnt?

How to program!

- Imperative program derivation by backwards reasoning.
- Key to imperative program derivation: every loop shall be built with an invariant and a bound in mind.
- Some techniques to construct loop invariants:
  - taking conjuncts as invariants;
  - replacing constants by variables;
  - strengthening the invariant;
  - tail invariants.

Most of the materials are from Kaldewaij.

What Have We Learnt?

And some more philosophical issues.

- What being *formal* means, and how it helps us.
- To program is to construct code that meets the specification;
- and to do so, the program must be constructed together with its proof.

Connection with other courses?

What we have learnt is *axiomatic semantics*.

- Denotational semantics: what a program *is*.
- Operational semantics: what it *does*.
- Axiomatic semantics: what it *guaratnees*.

We have not talked about Dijkstra's *weakest precondition* semantics, in which a program is seen as a *predicate transformer* – a function from predicates to predicates. See Dijkstra and Scholten.

What's Missing?

To begin with, notice the importance of purity in expressions.

- Side-effects strictly forbidden in expressions.
- $\{P[E/x]\}\, x := E\, \{P\}$ fails if E has side effects,
- which is why some programming languages have a clear separation of *expressions* and *statements*.

Fair enough, if you can design your own language. If you have to verify C, you have to somehow cope with it.

What's Missing?

- One reason making this calculus rather tedious: complex manipulation of quantifiers and array indexes.
- To certain extent it is the limitation of data structure we are using. To manipulate arrays, we tend to perform plenty of operations using indexes.
- Could we use "higher-level" data structures to avoid these messy details?

Purity and Aliasing

- *Aliasing* could cause disasters,
- which in turn makes call-by-reference dangerous.
  - Extra care must be taken when we introduce subroutines,
  - which is why procedure calls were such a big issue.
- If your interests are in program derivation, you could dismiss these problematic features. If you work on verification, however, you have to cope with them. We may see that in Frama-C.

Functional Program Derivation

In contrast, much of functional program derivation is essentially built on a theory of data structure.

$$
\begin{array}{ll}
 & max \circ map\ sum \circ segments \\
= & max \circ map\ sum \circ concat \circ map\ inits \circ tails \\
= & \quad \{\ map\ f \circ concat = concat \circ map\ (map\ f)\ \} \\
 & max \circ concat \circ map\ (map\ sum) \circ map\ inits \circ tails \\
= & \quad \{\ \text{since } max \circ concat = max \circ map\ max\ \} \\
 & \quad \vdots \\
= & max \circ scanr\ zmax\ 0.
\end{array}
$$

For an introduction, check out lectures in FLOLAC '07 and '08!

Separation Logic

- Another way out is separation logic: a logic about heap and stores.
- Advocated by John C. Reynolds.
- Facilitates reasoning about pointers and sharing.
- Separation between concurrent modules.

Where to Go from Here?

- Early issues of Science of Computer Programming have regular columns for program derivation.
- Books and papers by Dijkstra, Gries, Back, Backhouse, etc.
- You might not actually derive programs, but knowledge learnt here can be applied to program verification.
  - Plenty of tools around for program verification basing on pre/post-conditions. Some of them will be taught in this summer school.
- You might never derive any more programs for the rest of your life. But the next time you need a loop, you will know better how to construct it and why it works.